S^2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems

THÈSE N^{O} 6251 (2014)

PRÉSENTÉE LE 9 JUILLET 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DES SYSTEMES FIABLES
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Vitaly CHIPOUNOV

acceptée sur proposition du jury:

Prof. C. Koch, président du jury
Prof. G. Candea, directeur de thèse
Dr E. Bugnion, rapporteur
Prof. J. Regehr, rapporteur
Prof. M. Swift, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

Résumé

Les concepteurs de systèmes doivent régulièrement analyser le comportement de ce qu'ils construisent. Une analyse de base est de comprendre le comportement observé, par exemple pourquoi un système de fichiers se bloque ou corrompt un document donné, ou pourquoi un serveur web est lent sur un benchmark. Des analyses plus sophistiquées visent à caractériser le comportement futur dans des circonstances inédites, comme ce que serait la latence maximum et le débit minimum d'un serveur web une fois déployé sur un site client, ou s'il y a un moyen pour des pirates d'exploiter des bugs qui ne sont pas encore connus dans l'implémentation d'un système de fichiers.

Cette thèse présente S^2E, une plateforme scalable pour l'analyse des propriétés et du comportement de systèmes logiciels. La force de S^2E est sa capacité d'analyser de grands systèmes, comme une pile Windows complète, à l'aide de deux idées nouvelles : l'*exécution symbolique sélective* et les *modèles de cohérence d'exécution*. La sélectivité limite l'exploration multi-chemin au module d'intérêt (par exemple une bibliothèque) afin de minimiser la quantité de code exécuté symboliquement, ce qui évite une explosion de chemins à l'extérieur de ce module. Les modèles de cohérence d'exécution permettent de faire des compromis performance / précision lors de l'analyse. Par exemple, un modèle de cohérence détendue permet d'explorer dans un module d'intérêt tous les chemins qui pourraient être suivis par une exécution réelle, sans nécessiter de devoir chercher de tels chemins à travers l'ensemble du système.

Cette thèse montre également comment S^2E permet de concevoir deux nouvelles techniques d'analyse : un profileur de performances in-vivo multi-chemin ($PROF_S$) et un système semi-automatique de rétroingénierie de pilotes de périphériques binaires (RevNIC). $PROF_S$ permet de prédire la performance pour certaines catégories d'entrées, à l'aide de mesures telles que le nombre d'instructions ou d'échecs d'accès à la mémoire tampon. RevNIC analyse les pilotes de périphériques à source fermé afin de synthétiser de nouveaux pilotes plus fiables pour différents systèmes d'exploitation et architectures.

Nous montrons dans cette thèse comment on peut construire une telle plate-forme. S^2E est une machine virtuelle qui combine exécution symbolique avec des analyseurs de chemin modulaires. Le moteur de S^2E exécute le code binaire client de façon native dans une VM et passe à un interpréteur symbolique chaque fois qu'il rencontre une instruction qui accède à des données

symboliques. En cas de branchements conditionnels, l'interpréteur sépare le chemin d'exécution en deux de manière efficace en clonant l'état de la VM. L'exécution se divise de manière récursive pour chaque branchement, formant un arbre qui couvre le code qui serait autrement difficile à exercer manuellement. Sous le capot, S^2E étend QEMU avec un backend LLVM qui traduit le code machine en bitcode approprié pour l'interprétation symbolique, implémente la copie sur écriture pour splitter les chemins, et instrumente le MMU pour synchroniser les états symboliques et natifs de la machine.

La plateforme S^2E est open source et disponible sur `http://s2e.epfl.ch`, avec une démo prête à l'emploi, documentation et tutoriels. Trois ans après son lancement, S^2E a acquis une communauté d'utilisateurs en rapide expansion de plus de 200 membres et est activement utilisé par les chercheurs et les entreprises du monde entier dans le but de tester des réseaux distribués, d'analyser les systèmes de fichiers, de détecter les données privées leakées par les applis smartphone, effectuer des analyses de la sécurité, et plus encore.

Mots-clefs : Exécution symbolique sélective, Modèles de cohérence, Analyse multi-chemin, Virtualisation, Pilotes de périphérique, Rétroingénierie, Testing, Vérification, Profilage de performance, Analyse dynamique, Analyse statique

Abstract

System developers routinely need to analyze the behavior of what they build. One basic analysis is to understand observed behavior, such as why a file system crashes or corrupts a given document, or why a web server is slow on a given benchmark. More sophisticated analyses aim to characterize future behavior in previously unseen circumstances, such as what will a web server's maximum latency and minimum throughput be, once deployed at a customer site, or whether there will be ways for attackers to exploit the not-yet-known bugs in the file system implementation.

This thesis introduces S^2E, a scalable platform for analyzing the properties and behavior of software systems. S^2E's strength is the ability to scale to large systems, such as a full Windows stack, using two new ideas: *selective symbolic execution* and *execution consistency models*. Selectivity limits multi-path exploration to the module of interest (e.g., a library) to minimize the amount of symbolically-executed code, which avoids path explosion outside of that module. Execution consistency models allow to make principled performance/accuracy trade-offs during analysis. For example, a relaxed consistency model allows exploring all paths through a module of interest that would be followed by some concrete execution, without actually incurring the cost of finding such paths through the entire system.

This thesis also shows how S^2E enables two novel analysis techniques: an in-vivo multi-path performance profiler ($PROF_S$) and a system for semi-automatically reverse engineering binary device drivers (RevNIC). $PROF_S$ allows predicting the performance for certain classes of inputs, using metrics such as instruction count or cache misses. RevNIC analyzes closed-source device drivers to synthesize new, safer, and portable drivers for different OSes and architectures.

This thesis shows how one can build such a platform. S^2E is a virtual machine augmented with symbolic execution and modular path analyzers. S^2E's engine runs guest binaries natively in a VM and switches to a symbolic interpreter whenever an instruction accesses symbolic data. In case of conditional branches, the interpreter splits the execution path by efficiently cloning the entire VM state. Execution splits recursively for every branch, forming a tree that covers code otherwise hard to exercise manually. Under the hood, S^2E extends QEMU with an LLVM backend that translates machine code to bitcode suitable for symbolic interpretation, implements copy-on-write for path splitting, and instruments the MMU to synchronize symbolic and native machine states.

The S^2E platform is open sourced and available at `http://s2e.epfl.ch`, with a ready-to-use demo, documentation, and tutorials. Three years after release, S^2E acquired a rapidly growing user community of more than 200 members and is actively used by researchers and companies around the world in order to test distributed networks, analyze file systems, detect private data leaks in smartphone apps, perform security analysis, and more.

Keywords: Selective symbolic execution, Consistency models, Multi-path analysis, Virtualization, Device drivers, Reverse engineering, Testing, Verification, Performance profiling, Dynamic analysis, Static Analysis

Acknowledgments

I am grateful to the many people who helped me during the six years of my PhD and without whom this thesis would not have been possible.

I thank Professor George Candea, my advisor, for guiding me throughout my thesis.

I am grateful to all my lab mates, Cristian Zamfir, Horatiu Jula, Silviu Andrica, Stefan Bucur, Radu Banabic, Vova Kuznetsov, Baris Kasikci, Jonas Wagner, Johannes Kinder, and Nicoletta Isaac. I learnt a lot from them.

I would like to thank Professor Mike Swift, Professor John Regehr, Professor Christoph Koch, and Dr. Edouard Bugnion for their help and for accepting to be part of my thesis committee.

Special thanks to Volodymyr Kuznetsov for his key contributions in designing the architecture of S^{2}E and prototyping its core.

I would like to thank Willy Zwaenepoel for his early feedback on RevNIC, the ancestor of S^{2}E. Many thanks to Julia Lawall and the anonymous reviewers for giving us valuable feedback on the initial RevNIC prototype, which then gave birth to S^{2}E.

I thank Andrea Arpaci-Dusseau, Herbert Bos, Johannes Kinder, Miguel Castro, Byung-Gon Chun, Jim Larus, Petros Maniatis, Raimondas Sasnauskas, Willy Zwaenepoel, and the anonymous reviewers for their help in improving our initial paper about S^{2}E.

I would like to thank Stefan Bucur, Radu Banabic, João Carreira, Mihai Dobrescu, Laurent Fasnacht, Yoan Blanc, Roger Michoud, and Francesco Fucci for bringing S^{2}E to the next level of performance. Many thanks to Edouard Bugnion, whose insights will shape the next generation of the technology behind S^{2}E.

I thank all external S^{2}E contributors, whose outstanding efforts allowed growing the community and helped many researchers around the world. Thanks to Andreas Kirchner, who developed the first ARM and Android prototype of S^{2}E, to Jonas Zaddach and Luca Bruno for improving ARM support and enabling S^{2}E for the embedded world. Thanks to Michael Contreras who added LLVM 3.2 and 64-bit guest support. I thank Diego Biurrun for improving S^{2}E's build system. Many thanks to the S^{2}E community for reporting bugs. Congratulations to the QEMU and LLVM developers who built awesome systems. Many thanks to Daniel Dunbar, Cristian Cadar, and Dawson Engler for their work on KLEE. Without their contributions, S^{2}E would not have seen light.

I am grateful to EPFL, Google, SNF, and ERC, who generously financed my research. Many thanks to Intel for its generous award. Special thanks to EPFL for its unmatched support and outstanding infrastructures.

Finally, I would like to thank my parents, Tatiana and Victor, for supporting me throughout my education. Special thanks to my brother, Valeri, whose support and encouragement kept me sane in these past years. And to Titi, our beloved dog.

Contents

1 Introduction **13**

1.1 Problem Statement . . . 13

1.2 Thesis Objectives . . . 13

1.3 One Approach: Program Analysis . . . 15

1.4 Key Insights and Contributions . . . 16

1.5 Thesis Roadmap . . . 16

2 In-Vivo Multi-Path Analysis of Software Systems **17**

2.1 Efficient Path Exploration with Symbolic Execution . . . 19

2.2 Scaling to Large Software Stacks with Selective Symbolic Execution . . . 20

2.2.1 Symbolic → Concrete Transition . . . 22

2.2.2 Concrete → Symbolic Transition . . . 23

2.3 Execution Consistency Models . . . 24

2.3.1 Model Definitions . . . 24

2.3.1.1 Strict Consistency (SC) . . . 25

2.3.1.2 Relaxed Consistency (RC) . . . 27

2.3.2 Implementing Consistency Models . . . 30

2.3.2.1 Implementing Strict Consistency (SC) . . . 31

2.3.2.2 Implementing Relaxed Consistency (RC) . . . 32

2.3.3 Consistency Models in Existing Tools . . . 33

2.4 Summary . . . 34

3 A Platform for Developing Analyses **35**

3.1 Dynamic Binary Translation . . . 36

3.2 Dynamic Binary Translation for Symbolic Execution . . . 38

3.3 Execution Engine . . . 39

3.3.1 Sharing State Between Symbolic and Concrete Domains . . . 39

3.3.2 Symbolic Hardware . . . 42

3.3.3 Multiplexing Per-Path States . . . 44
3.4 Plugin Infrastructure . . . 45
3.5 Key Optimizations . . . 46
3.6 Summary . . . 48

4 Developing New System Analysis Tools **49**
4.1 Developing New Tools From Existing Plugins . . . 49
4.1.1 Path Selection . . . 49
4.1.2 Path Analysis . . . 51
4.1.3 Configuration Interface . . . 53
4.2 Developing New Plugins . . . 53
4.2.1 Building the *Annotations* Plugin . . . 56
4.2.2 Combining S^{2}E Plugins and In-VM Tools . . . 56
4.2.3 Summary . . . 57

5 Decompiling Program Binaries to LLVM **59**
5.1 Motivation . . . 60
5.1.1 LLVM as Intermediate Representation for Analysis of Binaries . . . 60
5.1.2 Use Cases . . . 61
5.2 Challenges . . . 63
5.3 Solution Overview . . . 63
5.4 LLVM Background . . . 63
5.5 Translating Blocks of Binary Code to LLVM . . . 64
5.6 Reconstructing the Control Flow Graph . . . 65
5.7 Obtaining Analyzable LLVM Programs . . . 66
5.7.1 Enabling Static Analysis . . . 66
5.7.2 Enabling Dynamic Analysis . . . 67
5.8 Results . . . 68
5.9 Discussion . . . 69
5.10 Summary . . . 70

6 Evaluation: Real Tools Built with S^{2}E **71**
6.1 Reverse Engineering of Closed-Source Drivers . . . 72
6.1.1 System Overview . . . 73
6.1.2 Tracing and Exercising the Original Driver . . . 76
6.1.2.1 Wiretapping the Driver . . . 76
6.1.2.2 Mechanic of Exercising the Driver . . . 76

6.1.2.3 Achieving High Coverage 77
6.1.3 Synthesizing A New Equivalent Driver 80
6.1.3.1 From Trace to a C-encoded State Machine 80
6.1.3.2 From State Machine to Complete Drivers 82
6.1.4 Evaluation 85
6.1.4.1 Effectiveness 87
6.1.4.2 Performance 88
6.1.4.3 Automation 91
6.1.4.4 Scalability 93
6.1.4.5 RevNIC vs. REV+ 94
6.1.5 Discussion and Limitations 96
6.1.6 How RevNIC's Limitations Influenced S^{2}E's Design 97
6.1.7 Related Tools 98
6.1.8 Summary 99
6.2 Deriving Performance Envelopes with Multi-Path Profiling 99
6.3 Other Tools 101
6.3.1 Automated Testing of Proprietary Device Drivers 101
6.3.2 Finding Bugs in Linux Device Drivers with SymDrive 102
6.3.3 Scalable Testing of File System Checkers with SWIFT 103
6.3.4 Prototyping Symbolic Execution Engines for Interpreted Languages with CHEF 104
6.3.5 Finding Trojan Message Vulnerabilities in Distributed Systems with Achilles 104
6.3.6 Scalable Testing of Distributed Systems with SymNet 105
6.3.7 Security Analysis of Embedded Systems' Firmware with AVATAR 106
6.3.8 Verifying Dataplanes in Software Switches and Routers 106
6.4 Performance of the S^{2}E Prototype 107
6.5 Trade-Offs in Using Execution Consistency Models 108
6.6 Summary 112

7 Related Work **113**
7.1 Accuracy vs. Performance in Analysis Tools 113
7.2 Source Code vs. Binary Analysis 115
7.2.1 Source Code Analysis 115
7.2.2 Binary Analysis 116
7.3 Choosing the Software Stack Level 118
7.4 S^{2}E in the Analysis Tools Design Space 118

8 Limitations and Future Work **121**

8.1 Limitations . . . 121

8.2 Future Work . . . 122

9 Conclusion **125**

Appendices

A Tutorial: Using the S^{2}E API to Build the *Annotations* Plugin **129**

A.1 Monitoring Module Loads . . . 129

A.2 Tracking Module Execution with *ModuleExecutionDetector* . . . 132

A.3 Monitoring Function Calls with *FunctionMonitor* . . . 133

A.4 Annotating Code with the *Annotations* plugin . . . 133

Chapter 1

Introduction

1.1 Problem Statement

System developers routinely need to analyze the behavior of what they build. One basic analysis is to *understand observed behavior*. For example, one might want to know why a given web server is slow on a given client request in order to be able to fix the problem, perhaps by tweaking the configuration or modifying the source code. Another example would be to understand why a device driver crashes in a given environment, and if it turns out to be a security flaw, prioritize the fixing of the bug. More sophisticated analyses aim to *characterize future behavior* in previously unseen circumstances, such as what will a web server's maximum latency and minimum throughput be, once deployed at a customer site. This can help with hardware provisioning. Ideally, system designers would also like to do quick *what-if analyses*, such as determining whether aligning a certain data structure on a page boundary will reduce cache misses and thus increase performance and energy efficiency. For small programs, experienced developers can often reason through some of these questions based on code alone; the goal of our work is to make it feasible to answer such questions also for large, complex, real systems.

1.2 Thesis Objectives

We introduce in this thesis a platform that enables easy construction of analysis tools (such as performance profilers, bug finders, or reverse engineering tools) that simultaneously offer the following three properties. First, they efficiently analyze entire families of execution paths. Second, they maximize realism by running the analyses within a real software stack. Third, they are able to directly analyze binaries. We explain these properties below.

First, predictive analyses often must reason about entire families of paths through the target

system, not just one path. A family of paths is a set of paths that have a specific property. For example, security analyses must check that there exist no corner cases that could violate a desired security policy; prior work has employed model checking [94] and symbolic execution [27] to find bugs in real systems—these are all multi-path analyses. One of our case studies demonstrates multi-path analysis of performance properties (§6.2): instead of profiling solely one execution path, we derive performance envelopes that characterize the performance of entire families of paths. Such analyses can check real-time requirements (e.g., that an interrupt handler will never exceed a given bound on execution time), or can help with capacity planning (e.g., determine how many web servers to provision for a web farm). In the end, properties shown to hold for *all* paths in a set constitute proofs over the corresponding set of executions; the guarantee provided by a proof is in essence the ultimate prediction of a system's behavior.

Second, an accurate estimate of program behavior often requires taking into account the whole environment surrounding the analyzed program: libraries, kernel, drivers, etc.—in other words, it requires in-vivo[1] analysis. Even small programs interact with their environment (e.g., to read/write files or send/receive network packets), so understanding program behavior requires understanding the nature of these interactions. Some tools execute the real environment, but allow calls from different execution paths to interfere inconsistently with each other [28]. Most approaches abstract away the environment behind a model [5, 27], but writing models is labor-intensive (taking in some cases multiple person-years [5]), models are rarely 100% accurate, and they tend to lose accuracy as the modeled system evolves. It is therefore preferable that target programs interact directly with their real environment during analysis in a way that keeps multi-path analysis consistent.

Third, real systems are made up of many components from various vendors; access to all corresponding source code is rarely feasible and, even when source code is available, building the code exactly as in the shipped software product is difficult [12]. Moreover, even if the source code is available, compilers can optimize it in many unpredictable ways. A security-conscious developer might want to put a buffer overflow check such as `buf + len < buf` only to find that the compiler removes it because in C, pointer overflow is undefined [125]. Thus, in order to be practical, analyses ought to operate directly on binaries.

Scalability is the key challenge of performing analyses that are in-vivo, multi-path, and operate on binaries. Going from single-path analysis to multi-path analysis turns a linear problem into an exponential one, because the number of paths through a program generally increases at least exponentially in the number of branches—the "path explosion" problem [15]. It is therefore

[1] *In vivo* is Latin for "within the living" and refers to experimenting using a whole live system; *in vitro* uses a synthetic or partial system. In life sciences, in-vivo testing—animal testing or clinical trials—is often preferred because, when organisms or tissues are disrupted (as in the case of in-vitro experiments), results can be substantially less representative. Analogously, in-vivo program analysis captures all interactions of the analyzed code with its surrounding system, not just with a simplified abstraction of that system.

not feasible today to execute fully symbolically an entire software stack (programs, libraries, OS kernel, drivers, etc.) as would be necessary if we wanted consistent in-vivo multi-path analysis.

1.3 One Approach: Program Analysis

We describe in this thesis S^{2}E, a general platform for developing multi-path in-vivo analysis tools that are practical even for large, complex systems, such as an entire Microsoft Windows or Linux software stack.

First, S^{2}E simultaneously exercises entire families of execution paths in a scalable manner by using *selective* symbolic execution and flexible *execution consistency models*. Selectivity limits multi-path exploration to the module of interest (e.g., a library) to minimize the amount of symbolically-executed code, which avoids path explosion outside of that module. Execution consistency models allow to make principled performance/accuracy trade-offs during analysis. For example, a relaxed consistency model allows exploring all paths through a module of interest that would be followed by some concrete execution, without actually incurring the cost of finding such paths through the entire system.

Second, S^{2}E employs virtualization to perform the desired analyses in vivo; this removes the need for the stubs or abstract models required by most state-of-the-art symbolic execution engines and model checkers [6, 55, 27, 112, 94].

Third, S^{2}E uses dynamic binary translation (DBT) to transparently instrument code running at any level of the software stack. The key advantage of DBT is its ability of handling a wide range of software, including proprietary systems, even if self-modifying or JITed, as well as obfuscated and packed/encrypted binaries. DBT allows exposing the running code to analysis plugins, which can both passively observe the state of the system and modify it depending on the needs of the analysis. A large body of work uses DBT to simulate entire systems [106, 9], implement virtual machine monitors [23], instrumentation code [82], etc. We give an in-depth presentation of dynamic binary translation in §3.1.

The abstraction offered by S^{2}E is that of an automated path exploration mechanism with modular path analyzers. The explorer drives in parallel the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). An analysis tool built on top of S^{2}E glues together path selectors with path analyzers. *Selectors* guide S^{2}E's path explorer by specifying the paths of interest: all paths that touch a specific memory object, paths influenced by a specific parameter, paths inside a target code module, etc. *Analyzers* can be pieced together from S^{2}E-provided default analyzers, or can be written from scratch using the S^{2}E API.

1.4 Key Insights and Contributions

This thesis makes the following contributions:

- ***Selective symbolic execution***, a new technique for automatic bidirectional symbolic–concrete state conversion that enables execution to seamlessly and correctly weave back and forth between symbolic and concrete mode;
- ***Execution consistency models***, a framework for advantageously balancing over- and under-approximation of paths in an analysis-specific way;
- A ***general platform*** for performing diverse in-vivo multi-path analyses in a way that scales to large real systems, demonstrated by building and evaluating several new analysis tools, such as a multi-path performance profiler, a tool for reverse engineering binary device drivers, finding bugs in drivers, and many others.
- A technique that uses selective symbolic execution and execution consistency models in order to semi-automatically ***reverse engineer*** device drivers.
- The first use of ***symbolic execution in performance analysis***.
- A ***wide adoption*** of the S^2E platform among the research community. Researchers at various institutions have used S^2E to build advanced bug finders for systems software [105, 30], analyze distributed systems [7, 108], test device firmware [133], verify the correctness of software routers [47], and more (see §6.3).

1.5 Thesis Roadmap

The thesis is organized as follows. Chapter 2 presents the theory behind S^2E: selective symbolic execution and execution consistency models. Chapter 3 explains how we turned the theory into a platform that can be used to build diverse analysis tools. Chapter 4 shows an overview of the S^2E SDK, including API and interfaces, and describes the various ways in which S^2E users can use them to build custom analysis tools. Chapter 5 shows how to transform the dynamic analysis core of S^2E into a static analysis tool that can operate on binaries. Chapter 6 shows how researchers, both ourselves and others, used S^2E to build powerful high-impact analysis tools. Chapter 7 describes related work and provides an overview of the analysis tools landscape. Chapters 8 presents future work ideas, and Chapter 9 concludes the thesis.

Chapter 2

In-Vivo Multi-Path Analysis of Software Systems

In this chapter, we present *selective symbolic execution* and *execution consistency models*, which allow S^2E to scale to large systems by letting developers make principled performance-accuracy trade-offs. To introduce these techniques, we first explain how one would traditionally analyze software by using manual testing, then how one can speed up the process by automating test generation, and finally how to make test generation more efficient.

A straightforward approach to analyze a system is to run one test at a time and observe the resulting behavior. A system takes certain inputs (e.g., command line arguments or request packets) and produces an output (e.g, written files or response packets). A test consists of a set of predefined inputs and one or more test predicates that check that the system produces the desired output after processing the given inputs. Each test exercises one execution path of the system, producing a certain level of code coverage. Developers can use this coverage information to write additional tests for parts of the code that were not exercised so far. Listing 2.1 shows a simple gear switching function part of a car's firmware. In order to exercise all its execution paths and achieve full line coverage, the developer would write three tests, first calling the function with *rpm = 0*, then calling it with *rpm = 1500*, and finally *rpm = 3000*.

One can further use automated test generation, alleviating the need for manually reasoning about the program. In the manual approach, developers need to reason about the inputs that would drive the execution down the paths that exercise the functionality of interest. This can be difficult for large programs that have many branch conditions. Automated test generation alleviates this problem by producing inputs that exercise many code paths without requiring large efforts from the developer. Automated test generation can use a number of techniques, such as random fuzzing, grammar-based test generation [58], concolic [55] and symbolic execution [69], and various heuristics in order to improve the quality of the generated tests.

Program analysis techniques improve path coverage by reducing the number of redundant tests. *Random fuzzing* chooses random inputs, yielding low path coverage. In the example of Listing 2.1, random fuzzing would need to guess distinct values for *rpm* such that all three paths are exercised. Assuming *rpm* is a 32-bit integer, the probability of exercising the second branch is lower than one chance in a million and most of the tests will only cover the first branch. *Grammar-based fuzzing* techniques can reach deeper paths by generating inputs that have a meaningful structure. For example, if one wanted to test a compiler's code generator, using random source files would not reach any code past the parser, because most of the random input would have invalid syntax. Grammar-based fuzzing would however generate random files with correct syntax (e.g., properly formed functions but with random arithmetic operations) that would reach the code of interest.

Concolic execution collects branch predicates along each execution path, which it then feeds into a constraint solver to derive a new set of inputs that will exercise a different execution path. In the example of Listing 2.1, the branch predicates collected when running the function with $rpm = 0$ are $rpm \leq 2500 \wedge rpm < 1000$. Concolic execution will attempt to negate $rpm \leq 2500$ and $rpm < 1000$, then use a constraint solver to obtain new values for *rpm* that would exercise different paths. In our case, concolic execution will ask the solver to give new inputs that satisfy $rpm \leq 2500 \wedge rpm \geq 1000$, for which the solver might return 1500 as a satisfying value for *rpm*. After than, concolic execution would try $rpm > 2500 \wedge rpm \geq 1000$, yielding e.g., $rpm = 3000$. Finally, $rpm > 2500 \wedge rpm < 1000$ is not satisfiable and there are no more combinations to try, stopping the exploration process after having produced 3 different execution paths.

While concolic execution is an iterative technique that reruns the program with new sets of inputs to derive additional inputs, *symbolic execution* builds an execution tree instead. Unlike concolic execution, symbolic execution does not rerun the program under analysis from the start, but from intermediate checkpoints, which are nodes in the execution tree. This makes path exploration more efficient for systems that have long prefixes (e.g., paths that need to run for a long time before reaching a location of interest). We will see next how symbolic execution works.

```
1 void autoShiftGear(unsigned &rpm, unsigned &gear) {
2   if (rpm > 2500) {
3     gear = gear + 1;
4     rpm = rpm * 0.5;
5   } else if (rpm < 1000) {
6     gear = gear - 1;
7     rpm = rpm * 2;
8   }
9 }
```

Listing 2.1 – A function with two branches and three execution paths. One can exercise all the three paths by calling the function with $rpm \in \{0, 1500, 3000\}$.

2.1 Efficient Path Exploration with Symbolic Execution

As we showed previously, a program can be treated as a superposition of possible execution paths. For example, a program that is all linear code except for one conditional statement *if* $(x>0)$ *then ... else ...* can be viewed as a superposition of two possible paths: one for $x>0$ and another one for $x \leq 0$. To exercise all paths, it is not necessary to try all possible values of x, but rather just one value greater than 0 and one value less than 0.

A symbolic execution engine unfurls this superposition of paths into a *symbolic execution tree*, in which each possible execution corresponds to a path from the root of the tree to a leaf corresponding to a terminal state. The mechanics of doing so consist of marking variables as symbolic at the beginning of the program, i.e., instead of setting a variable x to a concrete value (say, $x=5$), it is viewed as a superposition λ of all possible values x could take. Then, any time a branch instruction is conditioned on a predicate p that depends (directly or indirectly) on x, execution is split into two executions E_i and E_j, two copies of the program's state are created, and E_i's path remembers that the variables involved in p must be constrained to make p true, while E_j's path remembers that p must be false. In Figure 2.1, *rpm* is marked as symbolic, i.e., it can hold any value allowed by a 32-bit integer. When execution reaches the first branch, it is split into two executions, one of them getting $rpm > 2500$ as a constraint, the other $rpm \leq 2500$.

The process repeats recursively: E_i may further split into E_{i_i} and E_{i_j}, and so on. Every execution of a branch statement creates a new set of children, and thus what would normally be a linear execution (if concrete values were used) now turns into a tree of executions (since symbolic values are used). A node s in the tree represents a program state (a set of variables with formulae constraining the variables' values), and an edge $s_i \rightarrow s_j$ indicates that s_j is s_i's successor on any execution path satisfying the constraints in s_j. Paths in the tree can be pursued simultaneously, as the tree unfurls; since program state is copied, the paths can be explored independently. Copy-on-write is typically used to make this process efficient.

```
void autoShiftGear(unsigned &rpm,
                   unsigned &gear)
{
  if (rpm > 2500) {
    gear = gear + 1;
    rpm = rpm * 0.5;
  } else if (rpm < 1000) {
    gear = gear - 1;
    rpm = rpm * 2;
  }
}
```

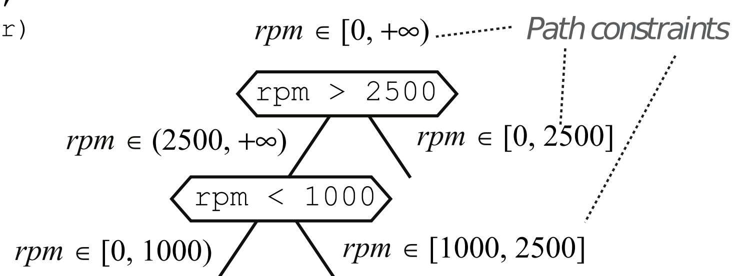

Figure 2.1 – Symbolic execution creates an execution tree with path constraints.

Symbolic execution relies on a constraint solver to decide which branches are feasible and to compute concrete input values that can be used as test cases. In the example of Figure 2.1, when execution reaches the first branch statement, the symbolic execution engine queries the constraint solver whether both outcomes are feasible. For this, the engine sends to the solver the set of path constraints $rpm \in [0, +\infty)$ as well as the query $rpm > 2500$. The solver replies that given the constraints, both outcomes are feasible. The process repeats recursively. When the second branch is reached, the solver checks that $rpm < 1000$ is feasible under the constraints $rpm \in [0, +\infty) \wedge rpm \leq 2500$. Finally, when the execution path terminates (e.g., the program exited successfully or crashed), the solver can compute concrete values for the symbolic inputs. In our example, the solver could return 0, 1500, and 3000 as solutions to the three execution paths. This can be useful to reproduce bugs, such as crashes or assertion failures.

While symbolic execution is effective at automated software testing, it suffers from path explosion. For large programs, the number of execution paths is exponential in the number of branches. It is therefore not possible to use symbolic execution effectively in order to thoroughly analyze large systems, such as an entire OS stack. In the next section, we present *selective symbolic execution*, a new approach that solves this problem.

2.2 Scaling to Large Software Stacks with Selective Symbolic Execution

Selective symbolic execution is based on the key observation that often *only some* families of paths are of interest. For example, one may want to exhaustively explore all paths through a small program, but not care about all paths through the libraries it uses or the OS kernel. This means that, when entering that program, selective symbolic execution should split executions to explore the various paths, but whenever the program calls into some other part of the system, such as a library, multi-path execution can cease and execution can revert to single-path. Then, when execution returns to the program, multi-path execution must be resumed.

Multi-path execution corresponds to *expanding* a family of paths by exploring the various side branches as they appear, while switching to single-path mode corresponds to *corseting* the family of paths. In multi-path mode, the tree grows in width and depth; in single-path mode, the tree only grows in depth. We therefore say selective symbolic execution's exploration of program paths is *elastic*. Selective symbolic execution turns multi-path mode off (i.e., do not further expand existing paths) whenever possible, to minimize the size of the execution tree and include only paths that are of interest to the target analysis.

Elasticity of multi-path exploration is key in enabling *selective symbolic execution*. Selective

symbolic execution combines virtualization with elasticity to offer the illusion of symbolically executing a full software stack, while executing symbolically only select components. For example, by concretely (i.e., non-symbolically) executing libraries and the OS kernel, it is possible to allow a program's paths to be explored efficiently without having to model its surrounding environment.

Interleaving of symbolic execution phases with concrete phases must be done carefully, to preserve the meaningfulness of each explored execution. In particular, one must handle constraints consistently to prevent inaccurate exploration that would cause analyzers to produce false positives.

For example, say we wish to analyze a program P in multi-path (symbolic) mode, but none of its libraries L_i are to be explored symbolically. If P has a symbolic variable n and calls `strncpy(dst,src,n)` in L_k, we must convert n to some concrete value and invoke `strncpy` with that value. This is straightforward: solve the current path constraints with a constraint solver and get some legal value for n (say n=5) and call `strncpy`. But what happens to n after `strncpy` returns? Variable *dst* will contain n=5 bytes, whereas n prior to the call was symbolic—can n still be treated symbolically? The answer is yes, if done carefully.

When a symbolic value is converted to concrete ($n\colon \lambda \to 5$), the family of executions is corseted. When a concrete value is converted to symbolic ($n : 5 \to \lambda$), the execution family is allowed to expand. The process of doing this back and forth is governed by the rules of an execution consistency model. For the above example, one might require that n be constrained to value 5 in all executions following the return from `strncpy`. However, doing so may exclude a large number of paths from the analysis. §2.3 describes systematic and safe relaxations of execution consistency.

We now describe the mechanics of switching back and forth between multi-path (symbolic) and single-path (concrete) execution in a way that executions stay consistent. We know of no prior symbolic execution engine that has the machinery to efficiently and flexibly cross the symbolic / concrete boundary both back and forth, while still preserving consistency of execution. Figure 2.2 provides a simplified example of using selective symbolic execution: an application *app* uses a library *lib* on top of an OS *kernel*. The target analysis requires to symbolically execute *lib*, but not *app* or *kernel*. Function *appFn* in the application calls a library function *libFn*, which eventually invokes a system call *sysFn*. Once *sysFn* returns, *libFn* does some further processing and returns to *appFn*. After the execution crosses into the symbolic domain (shaded) from the concrete domain (white), the execution tree (right side of Figure 2.2) expands. After the execution returns again to the concrete domain, the execution tree is corseted and does not add any new paths, until execution returns to the symbolic domain again. Some paths may terminate earlier than others, e.g., due to hitting a crash bug in the program.

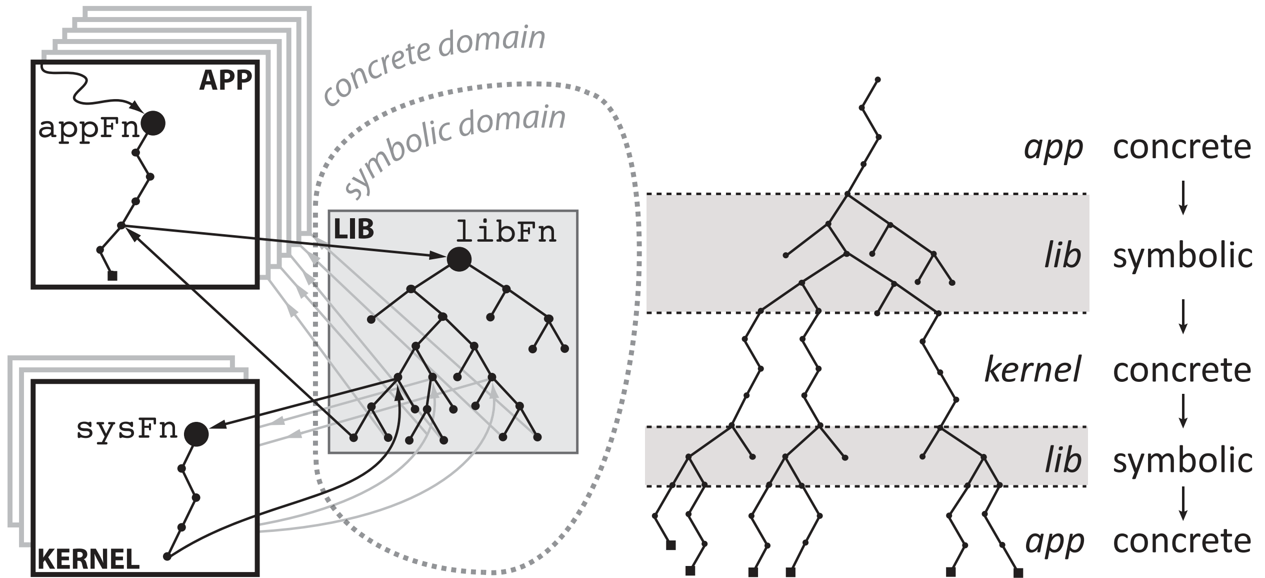

Figure 2.2 – Multi-path/single-path execution: three different modules (left) and the resulting execution tree (right). Shaded areas represent the multi-path (symbolic) execution domain, while the white areas are single-path.

2.2.1 Symbolic → Concrete Transition

In this section, we explain how selective symbolic execution handles symbolic to concrete transitions. Consider the call *libFn→sysFn* shown in Figure 2.3. Say *libFn* was called with an unconstrained symbolic value $x \in (-\infty, +\infty)$. At the first *if* branch instruction, execution forks into one path along which $x \in (-\infty, 5)$ and another path where $x \in [5, +\infty)$. These expressions are referred to as *path constraints*, as they constrain the values that x can take on a path. Along the then-branch, a call to *sysFn(x)* must be made. This requires x to be concretized, since *sysFn* is in the concrete domain. Thus, we choose a value, say x=4, that is consistent with the $x \in (-\infty, 5)$ constraint and perform the *sysFn(4)* call. The path constraints in the symbolic domain are updated to reflect that x=4. Note that a real implementation of selective symbolic execution would actually employ *lazy concretization*. As we shall see in §3.5, S^{2}E converts the value of x from symbolic to concrete on-demand, only when concretely running code is about to branch on the value of x. For the sake of clarity, in this section we assume eager (non-lazy) concretization.

Once *sysFn* completes, execution returns to *libFn* in the symbolic domain. When x was concretized prior to calling *sysFn*, the x=4 constraint was automatically added to the path constraints—*sysFn*'s return value is correct only under this constraint, because all computation in *sysFn* was done assuming x=4. Furthermore, *sysFn* may also have had side effects that are intimately tied to the x=4 constraint. With this constraint, execution of *libFn* continues, and correctness is preserved.

The problem, however, is that this constraint corsets the family of *future* paths that can be

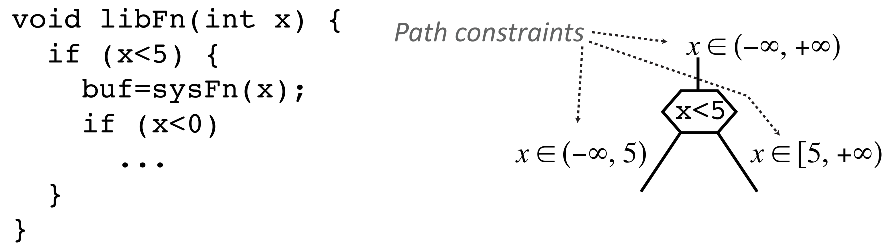

Figure 2.3 – The top level in *libFn*'s execution tree.

explored from this point on: x can no longer take on all values in $(-\infty, 5)$ so, when execution subsequently gets to the branch *if (x<0) ...*, the then-branch will no longer be feasible due to the added x=4 constraint. This is referred to as "overconstraining": the constraint is not introduced by features of *libFn*'s code, but rather as a result of concretizing x to call into the concrete domain. We think of x=4 as a soft constraint imposed by the symbolic/concrete boundary, while $x \in (-\infty, 5)$ is a hard constraint imposed by *libFn*'s code. Whenever a branch in the symbolic domain is disabled because of a soft constraint, it is possible to go back in the execution tree and pick an additional value for concretization, fork another subtree, and repeat the *sysFn* call in a way that would enable that branch.

2.2.2 Concrete $\rightarrow$ Symbolic Transition

When *appFn* calls *libFn*, it does so by using concrete arguments; the simplest concrete $\rightarrow$ symbolic conversion is to change the concrete arguments into unconstrained symbolic ones, e.g., instead of *libFn(10)* call *libFn(λ)*. One can additionally opt to constrain λ, e.g., *libFn(* $\lambda \leq 15$ *)*.

Once this transition occurs, selective symbolic execution runs *libFn* symbolically using the (potentially constrained) argument(s) and simultaneously executes *libFn* with the original concrete argument(s) as well. Once exploration of *libFn* completes, it returns to *appFn* the concrete return value resulting from the concrete execution, but *libFn* has been explored symbolically as well. In this way, the execution of *app* is consistent, while at the same time path analyzers are exposed to those paths in *lib* that are rooted at *libFn*'s entry point. All paths execute independently, and it is up to the path analyzers to decide whether, besides observing the concrete path, they also wish to look at the other paths.

Converting concrete values to symbolic ones must be done carefully. In the next section, we present execution consistency models, which define how conversions must be done in order to preserve a meaningful level of accuracy for a given analysis, without sacrificing performance.

2.3 Execution Consistency Models

The traditional assumption about system execution is that the state at any point in time is consistent, i.e., there exists a feasible concrete-execution path from the system's start state to the system's current state. However, there are many analyses for which this assumption is unnecessarily strong, and the cost of providing such strong consistency during multi-path exploration is often prohibitively high. For example, when doing unit testing, one typically exercises the unit in ways that are consistent with the unit's interface, without regard to whether all exercised paths are indeed feasible in the integrated system. This is both because testing the entire system in a way that exercises all paths through the unit is too expensive, and because exercising the unit as described above offers higher confidence in its correctness in the face of future use.

S^2E aims to be a general platform for system analyses, so it provides the ability to choose the level of execution consistency that offers the best trade-offs. In this section, we take a first step toward systematically defining alternate execution consistency models (§2.3.1), after which we explain how these different models dictate the symbolic/concrete conversions applied during the back-and-forth transition between the analyzed code and its environment (§2.3.2). In §2.3.3 we survey some of the ways in which consistency models appear in existing analysis tools.

2.3.1 Model Definitions

The key distinction between the various execution consistency models is which execution paths each model admits. Choosing an appropriate consistency model is a trade-off between how "realistic" the admitted paths are vs. the cost of enforcing the model during analysis. The appropriateness of the trade-off is determined by the nature of the analysis, i.e., by the way in which feasibility of different paths affects completeness and soundness of the analysis [1].

In the rest of the paper, we use the term *system* to denote the complete software system under analysis, including the application programs, libraries, and the operating system. We use the term *unit* to denote the part of the system that is to be analyzed. A unit could encompass different parts of multiple programs, libraries, or even parts of the operating system itself. We use the term *environment* to denote everything in the system except the unit. Thus, the system is the sum of the environment and the unit to be analyzed.

When defining a model, we think in terms of which paths it includes vs. excludes. Following the Venn diagram in Figure 2.4, an execution path can be *statically feasible*, in that there exists

[1] Execution consistency models defined here could be compared to memory consistency models in SMP systems [91]. While the former define the set of admissible execution paths, the latter specify the system's behavior in the presence of concurrent memory reads and writes and offer different trade-offs regarding cost of implementation at the micro-architecture level and programming complexity.

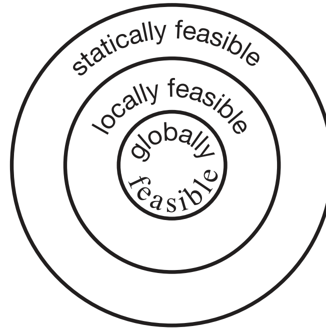

Figure 2.4 – Paths can be statically feasible, locally feasible, or globally feasible.

a path in the system's inter-procedural control flow graph (CFG) corresponding to the execution in question. A subset of the statically feasible paths are *locally feasible* in the unit, in the sense that the execution is consistent with both the system's CFG and with the restrictions on control flow imposed by the data-related constraints within the unit. Finally, a subset of locally feasible paths is *globally feasible*, in the sense that their execution is additionally consistent with control flow restrictions imposed by data-related constraints in the environment. Observing only the code executing in the unit, with no knowledge of code in the environment, it is impossible (by definition) to tell apart locally feasible from globally feasible paths.

We say a model is *complete* if exploration done under that model discovers eventually every path through the unit that corresponds to some globally feasible path through the system. A model is *consistent* if, for every path through the unit admitted by the model, there exists a corresponding globally feasible path through the system (i.e., the system can run concretely in that way).

We now define six points that we consider of particular interest in the space of possible execution consistency models, progressing from strongest to weakest consistency. They are summarized in Figure 2.5 using a representation corresponding to the Venn diagram above. Their completeness and consistency are summarized in Table 2.1. We invite the reader to follow Figure 2.5 while reading this section.

2.3.1.1 Strict Consistency (SC)

The strongest form of consistency is one that admits only the globally consistent paths. For example, the concrete execution of a program always obeys the strict consistency (SC) model. Moreover, every path admitted under the SC model can be mapped to a certain concrete execution of the system starting with certain concrete inputs. Sound analyses produce no false positives under the SC model. We define three subcategories of SC based on what information is taken into account when exploring new paths.

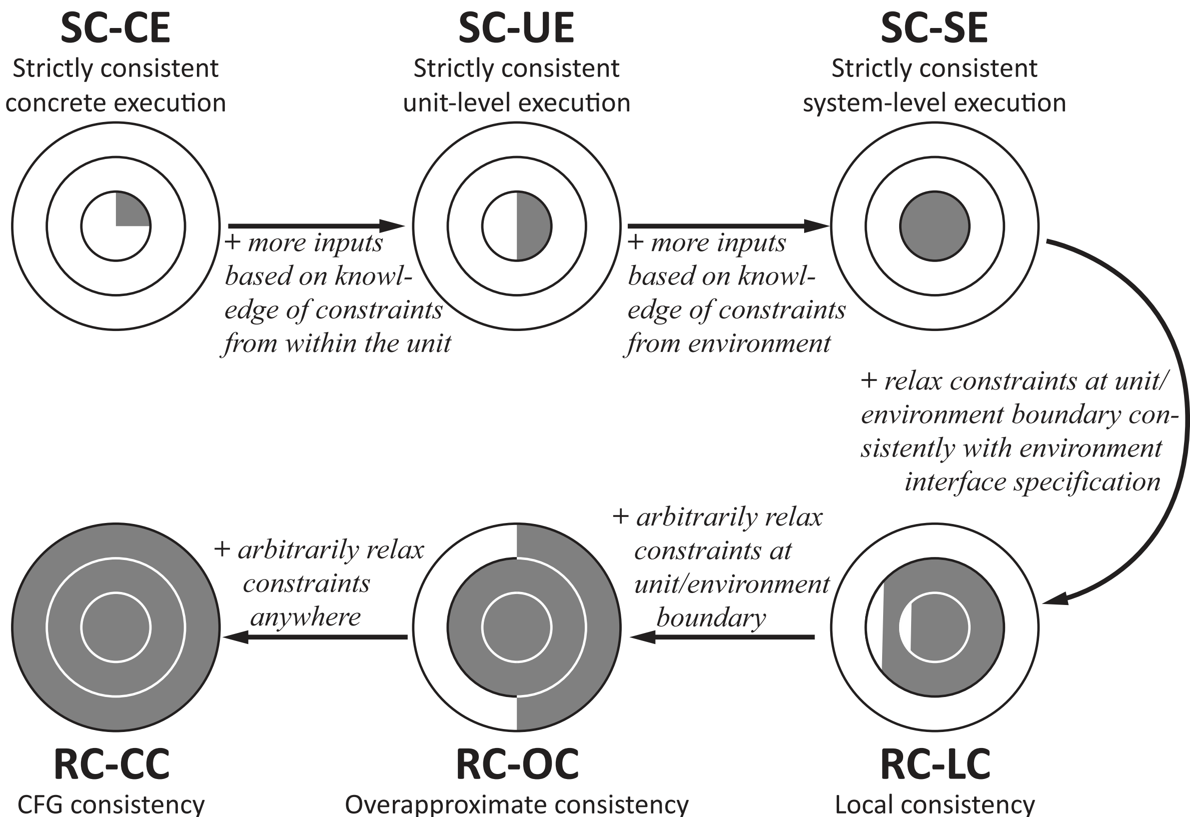

Figure 2.5 – Different execution consistency models cover different sets of feasible paths. The SC-CE model corresponds to concrete execution. The SC-UE and SC-SE models are obtained from the previous ones by using increasingly more information about the system, to explore increasingly bigger sets of concrete paths. The RC-LC, RC-OC and RC-CC models are obtained through progressive relaxation of constraints.

Strictly Consistent Concrete Execution (SC-CE) Under the SC-CE model, the entire system is treated as a black box: no knowledge of its internals is used to explore new paths. The only explored paths are the paths that the system follows when executed with the sample input provided by the analysis. New paths can only be explored by blindly guessing new inputs. Classic fuzzing (random input testing) [89] falls under this model.

Strictly Consistent Unit-Level Execution (SC-UE) Under the SC-UE model, an exploration engine is allowed to gather and use information internal to the unit (e.g., by collecting path constraints while executing the unit). The environment is still treated as a black box, i.e., path constraints generated by environment code are not tracked. Not every globally feasible path can be found with such partial information (e.g., paths that are enabled by branches in the environment can be missed). However, the exploration engine saves time by not having to analyze the environment, which is typically orders of magnitude larger than the unit.

This model is widely used by symbolic and concolic execution tools [28, 27, 55]. Such tools usually instrument only the program but not the OS code (sometimes such tools replace parts of the OS by models, effectively adding a simplified version of parts of the environment to the program). Whenever such tools see an OS call, they execute the call uninstrumented, selecting some concrete arguments for the call. Such "blind" selection of concrete arguments might cause some paths through the unit to be missed, if they depend on unexplored environment behaviors.

Strictly Consistent System-Level Execution (SC-SE) Under the SC-SE model, an exploration engine gathers and uses information about all parts of the system, to explore new paths through the unit. Such exploration is not only sound but also complete, provided that the engine can solve all constraints it encounters. In other words, every path through the unit that is possible under a concrete execution of the system will be found eventually by SC-SE exploration, making SC-SE the only model that is both strict and complete.

However, the implementation of SC-SE is limited by the path explosion problem: the number of globally feasible paths is roughly exponential in the size of the whole system. As the environment is typically orders of magnitude larger than the unit, including its code in the analysis (as would be required under SC-SE) offers an unfavorable trade-off, given today's technology.

2.3.1.2 Relaxed Consistency (RC)

Under relaxed consistency (RC), all paths through the unit are admitted, even those that are not allowed by the SC models. The RC model is therefore inconsistent in the general case.

The main advantage of RC is performance: by admitting these additional infeasible paths, one can avoid having to analyze large parts of the system that are not really targets of the analysis, thus allowing path exploration to reach the true target code sooner. However, admitting locally infeasible paths (i.e., allowing the internal state of the unit to become inconsistent) makes most analyses prone to false positives, because some of the paths these analyses are exposed to cannot be produced by any concrete run.

This might be acceptable if the analysis is itself unsound anyway, or if the analysis only relies on a subset of the state that can be easily kept consistent (in some sense, this is like RC-LC, except that the subset of the state to be kept consistent may not be the unit's state). Also note that, even though RC admits more paths, thus producing more analysis work, analyses under RC can abort early those paths that turn out to be infeasible, or the accuracy of the analysis can be decreased, thus preserving the performance advantage.

We distinguish three subcategories of the RC model, all of which are useful in practice.

Local Consistency (RC-LC) The local consistency (RC-LC) model aims to combine the performance advantages of SC-UE with the completeness advantages of SC-SE. The idea is to avoid exploring all paths through the environment, yet still explore the corresponding path segments in the unit by replacing the results of (some) calls to the environment with symbolic values that represent any possible valid result of the execution.

For example, when a unit (such as a user-mode program) invokes the `write(fd, buf, count)` system call of a POSIX operating system, the return value can be any integer between -1 and `count`, depending on the state of the system. The exploration engine can discard the actual concrete value returned by the OS and replace it with a symbolic integer between -1 and `count`. This allows exploring all paths in the unit that are enabled by different return values of `write`, without analyzing the `write` function and without having to find concrete inputs to the overall system that would enable those paths. This however introduces global inconsistency—for instance, according to the specification of the `write` system call, there exists no concrete execution in which (non-zero) `count` bytes are written to the file and `write` returns 0. However, unless the unit explicitly checks the file (e.g., by reading its content) this does not matter: the inconsistency cannot yield locally infeasible paths.

In other words, the RC-LC model allows for inconsistencies in the environment, while keeping the state of the unit internally consistent. To preserve RC-LC, an exploration engine must track the propagation of inconsistencies inside the environment and abort an execution path as soon as these inconsistencies influence the internal state of the unit on that path.

This keeps the state of the unit internally consistent on all explored paths: for each explored path, there exists some concrete execution of the system that would lead to exactly the same internal state of the unit along that path—except the engine does not incur the cost of actually finding that path. Consequently, any sound analysis that takes into account only the internal state of the unit produces no false positives under the RC-LC model. For this reason, we call the RC-LC model "locally consistent."

The set of paths explored under this model corresponds to the set of locally feasible paths, as defined earlier. However, some paths could be aborted before completion, or even be missed completely, due to the propagation of inconsistencies outside the unit. This means that the RC-LC model is not complete. In practice, the less a unit interacts with its environment, the fewer such paths are aborted or missed.

Using the RC-LC model in practice requires writing annotations for API functions called by the unit under analysis. An annotation specifies how to turn concrete values into symbolic ones, like in the case of the `write` system call described earlier. Writing such annotations is fairly straightforward, in contrast to writing environment models, where one must specify the complete behavior of the API function (and this makes both the writing of environment models and the

ensuring of their correctness and completeness in the face of code evolution hard [5]).

Overapproximate Consistency (RC-OC) In the RC-OC model, path exploration can follow paths through the unit while completely ignoring the constraints that the environment/unit API contracts impose on return values and side effects. For example, the unit may invoke `write(fd,buf, count)`, and the RC-OC model would permit the return result to be larger than `count`, which violates the specification of the `write` system call. Under the previous model (local consistency), such paths would be disallowed. Even though it is not consistent, RC-OC is complete: every environment behavior is admitted under RC-OC, so every path in the unit corresponding to some real environment behavior is admitted too.

The RC-OC model is useful, for example, for reverse engineering. It enables efficient exploration of all behaviors of the unit that are possible in a valid environment, plus some additional behaviors that are possible only when the environment behaves outside its specification. For instance, when reverse engineering a device driver, the RC-OC model allows symbolic hardware [74] to return unconstrained values; in this way, the resulting reverse engineered paths include some of those that correspond to allegedly impossible hardware behaviors. Such overapproximation improves the quality of the reverse engineering, as explained in [33].

CFG Consistency (RC-CC) The RC-CC model admits any execution paths, as long as they correspond to paths in the unit's inter-procedural control flow graph. This roughly corresponds to the consistency provided by static program analyzers that are dataflow-insensitive and analyze paths that are completely unconstrained. Being strictly weaker than the SC-SE model, though using the same information to explore new paths, the RC-CC model is complete.

The RC-CC model is useful in disassembling obfuscated and/or encrypted code: after letting the unit code decrypt itself under an RC-LC model (thus ensuring the correctness of decryption), a disassembler can switch to the RC-CC model to reach high coverage of the decrypted code and quickly disassemble as much of it as possible.

To summarize, we presented six consistency models that offer flexible trade-offs between false positives, false negatives, and performance (Table 2.1). The SC-CE model has zero false positives but yields many false negatives because it explores a tiny fraction of the paths in the system. The SC-UE model reduces false negatives and SC-SE eliminates them at the expense of high exploration cost. Relaxed consistency models alleviate this high cost by allowing inconsistencies. This allows the RC-LC model to explore paths through the unit that would be followed by some concrete execution without actually incurring the cost of finding such paths. The RC-OC model introduces further inconsistencies to guarantee zero false negatives at the expense of introducing false positives. Finally, RC-CC completely unconstrains execution to speed up path exploration.

Model	Consistency	Completeness	Use Case
SC-CE	consistent system-wide	incomplete	Single-path profiling/testing of units that have a limited number of paths
SC-UE	consistent system-wide	incomplete	Analysis of units that generate hard-to-solve constraints (e.g., cryptographic code)
SC-SE	consistent system-wide	complete	Sound and complete verification without false positives or negatives; testing of tightly coupled systems with fuzzy unit boundaries.
RC-LC	locally consistent	incomplete	Testing/profiling while avoiding false positives from the unit's perspective
RC-OC	inconsistent	complete	Reverse engineering: extract consistent path segments
RC-CC	inconsistent	complete	Dynamic disassembly of a potentially obfuscated binaries

Table 2.1 – S^2E consistency models: completeness, consistency, and use cases. Each use case is assigned to the weakest model it can be accomplished with.

2.3.2 Implementing Consistency Models

We now explain how the consistency models can be implemented by a selective symbolic execution engine (SSE), by describing the specifics of symbolic $\leftrightarrow$ concrete conversion as execution goes from the unit to the environment and then back again.

We illustrate the implementation details with an example of a kernel-mode device driver (Figure 2.6). The driver reads/writes from/to hardware I/O ports and calls the `write_usb` function, which is implemented in a kernel-mode library, as well as `alloc`, implemented by the kernel itself.

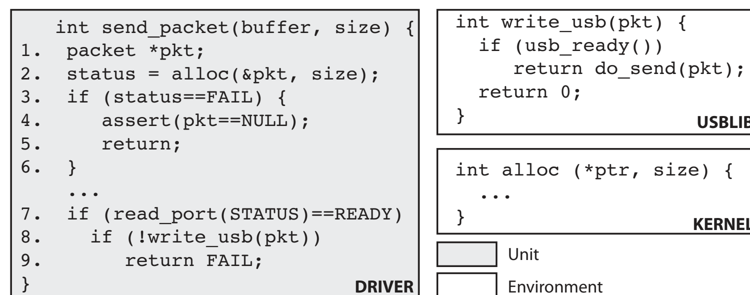

Figure 2.6 – Example of a “unit” (device driver) interacting with its “environment” (kernel-mode library and OS kernel itself). The unit is shaded.

2.3.2.1 Implementing Strict Consistency (SC)

Strictly Consistent Concrete Execution (SC-CE) For this model, an SSE allows only concrete input to enter the system. This leads to executing a single path through the unit and the environment. The SSE can execute the whole system natively without having to track or solve any constraints, because there is no symbolic data.

Strictly Consistent Unit-Level Execution (SC-UE) To implement this model, the SSE converts all symbolic data to concrete values when the unit calls the environment. The conversion is consistent with the current set of path constraints in the unit. No other conversion is performed. The environment is treated as a black box, and no symbolic data can flow into it.

In the example of Figure 2.6, the SSE concretizes the content of packet `pkt` consistently with the path constraints when calling `write_usb` and, from there on, this soft constraint (see §2.2.1) is treated as a hard constraint on the content of `pkt`. The resulting paths through the driver are globally feasible paths, but exploration is not complete, because treating the constraint as hard can curtail globally feasible paths during the exploration of the driver (e.g., paths that depend on the packet type).

Strictly-Consistent System-Level Execution (SC-SE) Under SC-SE, the SSE lets symbolic data cross the unit/environment boundary, and the entire system is executed symbolically. This preserves global execution consistency.

Consider the `write_usb` function. This function gets its external input from the USB host controller via the `usb_ready` function. Under strict consistency, the USB host controller (being "outside the system") can return a symbolic value, which in turn propagates through the USB library, eventually causing `usb_ready` to return a symbolic value as well.

Path explosion due to a large environment can make SC-SE hard to use in practice. The paths that go through the environment can substantially outnumber those that go through the unit, possibly delaying the exploration of interest. An SSE can heuristically prioritize the paths to explore, or employ *incremental symbolic execution* to execute parts of the environment as much as needed to discover interesting paths in the unit quicker. We describe this next:

The execution of `write_usb` proceeds as if it was executed symbolically, but only one globally feasible path is pursued in a depth-first manner, while all other forked paths are stored in a wait list. This simulates a concrete, single-path execution through a symbolically executing environment. After returning to `send_packet`, the path being executed carries the constraints that were accumulated in the environment, and symbolic execution continues in `send_packet` as if `write_usb` had executed symbolically. The return value x of `write_usb` is constrained according to the depth-first path pursued in the USB library, and so are the side effects. If, while executing

`send_packet`, a branch that depends on x becomes infeasible due to the constraints imposed by the call to `write_usb`, the SSE returns to the wait list and resumes execution of a wait-listed path that, for instance, is deemed to eventually execute line 9.

2.3.2.2 Implementing Relaxed Consistency (RC)

Local Consistency (RC-LC) For RC-LC, an SSE converts, at the unit/environment boundary, the concrete values generated by the environment into symbolic values that satisfy the constraints of the environment's API. This enables multi-path exploration of the unit. Referring to Figure 2.6, the SSE would turn `alloc`'s return value v into a symbolic value $\lambda_{\mathsf{ret}} \in \{v, \texttt{FAIL}\}$ and `pkt` into a symbolic pointer, while ensuring that $\lambda_{\mathsf{ret}} = \texttt{FAIL} \Rightarrow \texttt{pkt} = \mathit{null}$, so that the `alloc` API contract is satisfied.

If symbolic data is written by the unit to the environment, the SSE must track its propagation. If a branch in the environment ever depends on this data, the SSE must abort that execution path, because the unit may have derived that data based on symbolic input from the environment that subsumed values the environment could not have produced in its state at the time.

From the driver's perspective, the global state may seem inconsistent, since the driver is exploring a failure path when no failure actually occurred. However, this inconsistency has no effect on the execution, as long as the OS does not make assumptions about whether or not buffers are still allocated after the driver's failure. RC-LC would have been violated had the OS read the symbolic value of `pkt` (e.g., if the driver stored it in an OS data structure).

Overapproximate Consistency (RC-OC) In this model, the SSE converts concrete values at the unit/environment interface boundaries into unconstrained symbolic values that disregard interface contracts. For example, when returning from `alloc`, both `pkt` and `status` become completely unconstrained symbolic values.

This model brings completeness at the expense of substantial overapproximation. No feasible paths are ever excluded from the symbolic execution of `send_packet`, but since `pkt` and `status` are completely unconstrained, there could be locally infeasible paths when exploring `send_packet` after the call to `alloc`.

As an example, note that `alloc` promises to set `pkt` to *null* whenever it returns `FAIL`, so the `assert` on line 4 should normally never fail. Nevertheless, under RC-OC, both `status` on line 3 and `pkt` on line 4 are unconstrained, so both outcomes of the `assert` statement are explored, including the infeasible one. Under stronger models, like RC-LC, `pkt` must be null if `status==FAIL`.

CFG Consistency (RC-CC) An SSE can implement RC-CC by pursuing all outcomes of every branch, regardless of path constraints, thus following all edges in the unit's inter-procedural CFG. Under RC-CC, exploration is fast, because branch feasibility need not be checked with a constraint solver. As mentioned earlier, one use case is a dynamic disassembler, where running with stronger consistency models may leave uncovered (i.e., non-disassembled) code. Implementing RC-CC may require program-specific knowledge, to avoid exploring non-existing edges, as in the case of an indirect jump pointing to an unconstrained memory location.

2.3.3 Consistency Models in Existing Tools

Some of these consistency models already appear in existing tools; we survey them here as a way to further explain S^2E's consistency models.

Most dynamic analysis tools use the SC-CE model. Examples include Valgrind [123] and Eraser [109]. These tools execute and analyze programs along a single path, generated by user-specified concrete input values. Being significantly faster than multi-path exploration, analyses performed by such tools are, for instance, useful to characterize or explain program behavior on a small set of developer-specified paths (i.e., test cases). However, such tools cannot provide any confidence that results of the analyses extend beyond the concretely explored paths.

Dynamic test case generation tools usually employ either the SC-UE or the SC-SE models. For example, DART [55] uses SC-UE: it executes the program concretely, starting with random inputs, while collecting path constraints on each execution. DART uses these constraints to produce new concrete inputs that would drive the program along a different path on the next run. However, DART does not instrument the environment and hence cannot use information from it when generating new concrete inputs, thus missing feasible paths, which is characteristic of SC-UE.

As another example, KLEE [27] uses either the SC-SE or a form of the SC-UE model, depending on whether the environment is modeled or not. In the former case, both the unit and the model of the environment are executed symbolically. In the latter case, whenever the unit calls the environment, KLEE executes the environment with concrete arguments. However, KLEE does not track the side effects of executing the environment, allowing them to propagate across otherwise independent execution paths, thus making the corresponding program states inconsistent. Due to this limitation, we cannot say KLEE implements precisely SC-UE as defined here.

Static analysis tools usually implement some form of the RC model. For example, SDV [5] converts a program into a boolean form, which is an over-approximation of the original program. Consequently, every path that is feasible in the original program would be found by SDV, but the tool also finds additional infeasible paths.

2.4 Summary

In this chapter, we have shown how to combine selective symbolic execution with execution consistency models in order to achieve in-vivo analysis. Selectivity limits multi-path exploration to the module of interest (e.g., a library) to minimize the amount of symbolically-executed code, which avoids path explosion outside of that module. Execution consistency models allow to make principled performance/accuracy trade-offs during analysis. In the next chapter, we show how to actually build an analysis platform that implements selective symbolic execution and execution consistency models.

Chapter 3

A Platform for Developing Analyses

In this chapter, we show how we implemented selective symbolic execution in the S^2E prototype. This system is meant to be a platform for rapid prototyping of custom system/program analyses that employ various execution consistency models. S^2E offers two key interfaces, one for path selection and one for analysis; we describe these interfaces further in Chapter 4. S^2E explores paths by running the target system in a virtual machine and selectively executing selected parts of it symbolically. Depending on which parts are desired, some of the machine instructions of the system being analyzed are dynamically translated within the VM into an intermediate representation suitable for symbolic execution, while the rest are translated to the host instruction set. Underneath the covers, S^2E transparently converts data back and forth as execution weaves between the symbolic and concrete domains, so as to offer the illusion that the full system (OS, libraries, applications, etc.) is executing in multi-path mode.

Figure 3.1 shows the S^2E architecture. We reused parts of the QEMU virtual machine [9], the KLEE symbolic execution engine [27], and the LLVM tool chain [78]. To these, we added 30 KLOC of C++ code written from scratch, not including third party libraries[1]. We added 1 KLOC of new code to KLEE and modified 1.5 KLOC; in QEMU, we added 1.5 KLOC of new code and modified 3.5 KLOC of existing code. S^2E currently runs on Mac OS X and Linux, it can execute any guest OS that runs on x86 or ARM (e.g., Android OS), and can be easily extended to other CPU architectures, like MIPS or PowerPC.

In the rest of this section, we explain how S^2E uses dynamic binary translation (§3.1), how the execution engine handles concretely and symbolically running code (§3.3), and the details of the plugin infrastructure (§3.4). Finally, we conclude the section with some of the optimizations that are key to making the illusion of whole-system symbolic execution feasible (§3.5).

[1] All reported LOC measurements were obtained with SLOCCount [126].

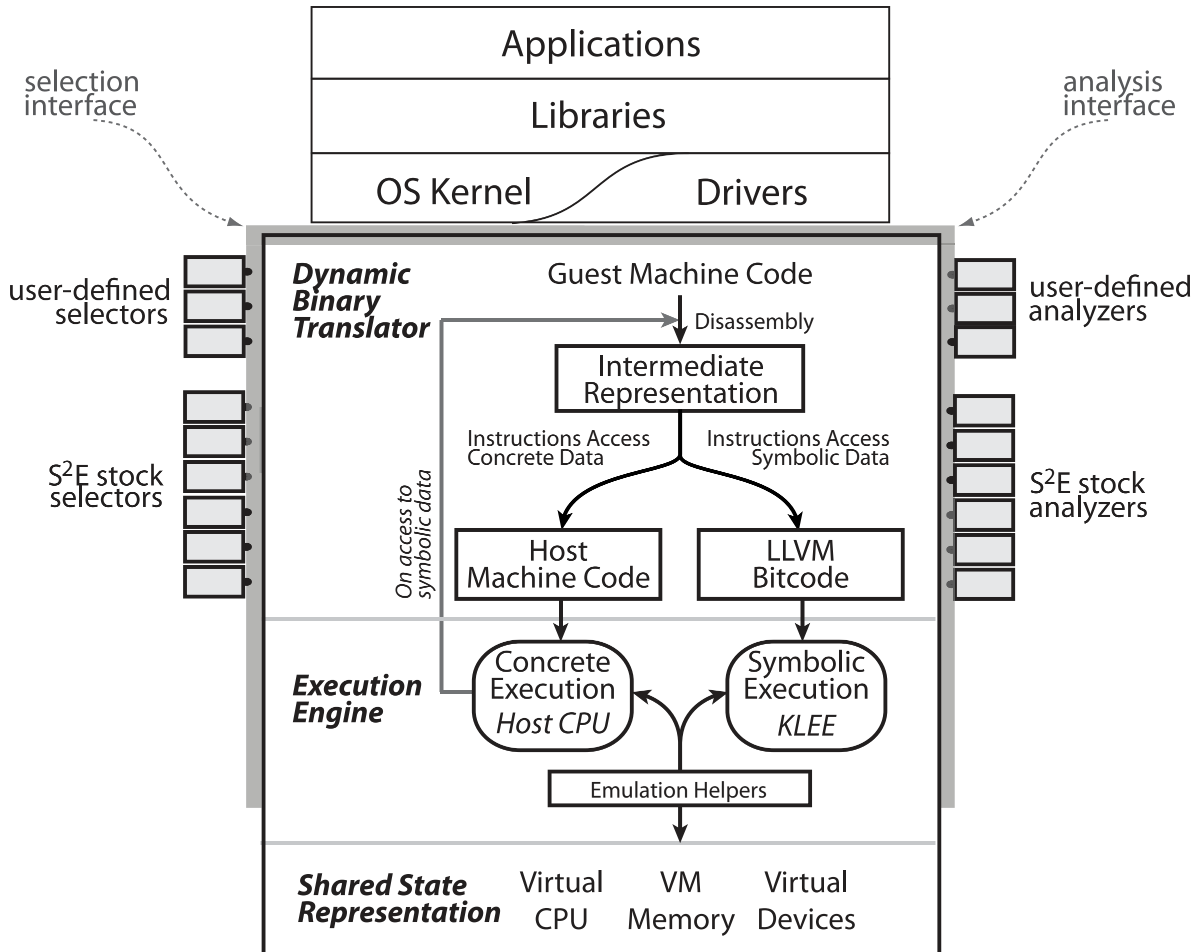

Figure 3.1 – S²E architecture, centered around a custom VM. The VM dynamically dispatches guest machine instructions to the host CPU for native execution, or to the symbolic execution engine for symbolic interpretation, depending on whether the instructions access symbolic data or not. The system state is shared between the code running natively and the code interpreted in the symbolic execution engine; this enables S²E to achieve the level of execution consistency desired by the user.

3.1 Dynamic Binary Translation

In this section, we first discuss the general idea behind dynamic binary translation and how it enables building a platform such as S²E. We then illustrate it by looking at how S²E leverages the dynamic binary translator offered by QEMU.

Binary translation in general is a powerful technique for code instrumentation and is therefore a cornerstone of many tools. Static binary translation has been originally used to run software compiled for one architecture on another, without recompiling it [116]. Dynamic binary translation has been used among others to simulate entire systems [106, 9], implement virtual machine monitors [23], and instrument code [82].

A dynamic binary translator (DBT) converts at run-time the executable code of one platform to executable code of another. A DBT works in a loop: it continuously fetches blocks of guest code, translates them to the host's instruction set, and passes the resulting translation to the execution engine. The DBT determines which code to fetch and translate by reading the state of the virtual CPU and the guest memory (i.e., the current program counter in the CPU and program code stored in memory). This state is updated as part of the execution of the translated code.

In the context of S^2E, a DBT brings several advantages. The DBT allows inserting arbitrary code in the guest instruction stream in order to create symbolic values, instrument all memory and register accesses to ensure a consistent synchronization between the symbolic and concrete domain (§3.3.1), provide symbolic hardware (§3.3.2), and to call plugin code when interesting guest instructions are executed (§3.4).

First, the DBT translates guest instructions to an intermediate representation consisting of simpler *micro-operations*. Micro-operations split complex guest instructions into simpler operations that are easier to emulate. Consider the x86 instruction `inc [eax]`, which increments the value in the memory location whose address is stored in the `eax` register. The DBT decomposes this instruction into a memory load to a temporary register, an increment of that register, and a subsequent memory store to the original location.

The DBT packages micro-operations into *translation blocks*. A translation block contains a sequence of micro-operations up to and including the first micro-operation that modifies the control flow, such as a branch, a call, or a return. The translator cannot add to the translation block the instructions past the control flow change, because the translator cannot always determine statically at which code location to continue the translation process.

The DBT transforms the micro-operations forming the translation block into machine instructions of the host instruction set by turning each micro-operation into an equivalent sequence of host instructions, using a code dictionary that maps micro-operations to host instructions. Most of the conversions consist of a one-to-one mapping from a micro-operation to the corresponding machine instruction. For more complex instructions, like those that manipulate the processor's control state or that access memory, the DBT emits a micro-operation that calls emulation helpers (which are C functions that emulate the original guest machine instruction). Some helpers have tens of lines of code and are used frequently (e.g., for memory accesses), therefore inlining them in the translated code would be prohibitively expensive in terms of generated code size.

Micro-operations simplify the translation process by abstracting away the guest's instruction set. Without such an IR, translation would require a different translator for every pair of guest and host instruction sets. For example, supporting 8 guest platforms and 4 hosts would require 32 translators. In contrast, the use of an IR requires only 8 *front-ends* (to transform the guest code to the IR) and 4 *back-ends* (to convert the IR to the host's instruction set).

S^{2}E uses QEMU for dynamic binary translation. QEMU comes with many front-ends, including Alpha, ARM, x86, Microblaze, Motorola 68K, MIPS, PowerPC, and SPARC. Back-ends include ARM, x86, MIPS, SPARC, PowerPC, and PowerPC 64. QEMU can run x86 Windows on a MIPS machine by translating the x86 code to the MIPS instruction set.

3.2 Dynamic Binary Translation for Symbolic Execution

In order to enable symbolic execution, one needs to interpret the IR generated by the DBT. For each micro-operation, the interpreter fetches the value of its operands (that are possibly symbolic), performs the operation specified by the micro-operation (possibly producing a new symbolic expression), and storing the result to the appropriate register or memory location.

In order to provide acceptable performance, the IR must be translated to native code as often as possible, using a native back-end. The native back-end allows running at full speed code that does not manipulate symbolic data. Native execution is triggered as soon as the CPU's register state becomes concrete. The native code must however be instrumented in order to switch execution to the symbolic interpreter as soon as a memory read returns a symbolic value.

In the rest of this chapter, we show how we modified QEMU's DBT in order to enable symbolic execution and switching between symbolic and native mode.

We added the LLVM back-end to QEMU, in order to interface S^{2}E with the KLEE symbolic execution engine. This back-end translates micro-operations to the LLVM intermediate representation, which is directly interpretable by KLEE (see §3.3). Neither the guest OS nor KLEE are aware of the x86-to-LLVM translation: the guest OS sees that its instructions are being executed, and KLEE only sees LLVM instructions, just as if they were coming from a program entirely compiled to LLVM. In this way, the guest thinks the "entire world" is concrete, while KLEE thinks the "entire world" is symbolic.

The DBT must translate code in a way that allows precise exception handling, given that execution could be interrupted at any time by hardware interrupts, page faults, etc. S^{2}E extends the DBT to enable precise exception handling from LLVM code. When an exception occurs, QEMU converts the address of the *translated* instruction that raised the exception to the program counter of the *guest* code. Such a conversion is possible because each guest instruction corresponds to a clearly delimited sequence of host machine instructions. However, there is no such clear correspondence in LLVM code, because LLVM applies more aggressive optimizations within each translation block. To solve this, we modified the DBT to insert micro-operations that explicitly update the program counter before each guest instruction is executed. As a result, both the LLVM code and the native code see a consistent program counter at every point during execution, allowing precise exception handling.

3.3 Execution Engine

We now present the extensions to QEMU's execution engine that enable transparent switching between concrete and symbolic execution, while preserving the consistency of the execution state.

The execution engine consists of a loop that calls the DBT to translate guest code, then runs the translated code in native mode or interprets it in the symbolic execution engine. The execution engine does not know a priori whether to ask the DBT to generate LLVM or native code. It first instructs the DBT to generate native code and, if the code reads memory locations that contain symbolic data, it invokes the DBT to retranslate the code to LLVM. The DBT stores the translations in a cache to avoid needless retranslations, such as when repeatedly executing a loop body.

S^{2}E mediates access to most of the VM state via *emulation helpers*. While simple instructions can access the CPU state directly, memory accesses, device I/O, as well as complex manipulations of the CPU state go through specific helpers, in order to reduce the size of the translated code. For example, the translated code for the software interrupt instruction triggers during execution the `do_interrupt` helper. This helper emulates the instruction's behavior by checking the current execution mode and privilege level, saving registers, taking care of potential exceptions, etc.

S^{2}E provides emulation helpers both for concrete (native) execution and symbolic (LLVM) execution. Native-mode helpers mediate access to the shared state when S^{2}E executes concrete code on the host CPU, while LLVM helpers are used in symbolic execution mode. The execution engine runs native-mode helpers on the host CPU and interprets LLVM helpers in the symbolic execution engine; LLVM helpers must sometimes call native-mode QEMU code, for example to simulate a virtual device.

LLVM emulation helpers avoid forceful unnecessary concretizations that would arise from calling native emulation helpers from within KLEE. The emulation helpers, called by the translated code, are compiled twice: to x86 and to LLVM. When running in symbolic mode, KLEE executes the LLVM version of the helper in order to let the helper manipulate symbolic data. If that version was missing, KLEE would be forced to call the native x86 version of the helper, which would then forcefully concretize the symbolic data. For example, QEMU implements bit-shift operations in helpers; if the bit-shift helper was available in x86 form only, the data it manipulates would have to be concretized when called from KLEE.

3.3.1 Sharing State Between Symbolic and Concrete Domains

S^{2}E combines concrete with symbolic execution in a controlled fashion along the same path by using a representation of machine state that is shared between the VM and the embedded symbolic execution engine. S^{2}E redirects reads and writes from QEMU and KLEE to the common machine state, which consists of VM physical memory, virtual CPU state, and virtual device state (see

Figure 3.2). In this way, S^2E can transparently convert data between concrete and symbolic, according to the desired consistency model, and provide distinct copies of the entire machine state to distinct paths. S^2E reduces the memory footprint of all these system states by several orders of magnitude through copy-on-write (§3.5).

S^2E implements transparent state sharing by using KLEE's `ObjectState` data structure for the CPU and the physical memory. This structure encapsulates an array of concrete bytes and symbolic expressions. It provides accessors to get and set concrete or symbolic bytes. To execute native code more efficiently, S^2E extends `ObjectState` to expose a direct pointer to the concrete array of bytes, bypassing getters and setters. It also exposes a pointer to a bitmap that indicates which bytes are symbolic and which are concrete.

Sharing the CPU State S^2E splits the CPU state into symbolic and concrete regions, each in a different `ObjectState` structure. The *symbolic region* contains the general purpose registers and the flags registers. These registers can store symbolic values. The *concrete region* stores the control state of the system, including segment registers, program counter, control and debug registers, etc. S^2E does not allow this state to become symbolic, because doing so would cause execution to fork inside the S^2E emulation code, thus exercising the emulator and not the target software. For example, a symbolic protection mode bit in the `CR0` register would fork the translator excessively often, since many instructions behave differently in protected mode vs. in real mode.

S^2E concretizes all symbolic data written to the concrete region. For example, S^2E concretizes symbolic addresses when they are written to the (always concrete) program counter. To avoid reducing the completeness of exploration too much, S^2E actually allows execution to fork up to some predefined number of times, and then concretizes the program counter in each of the states. This behavior can be customized by the user, via the S^2E API. Finally, S^2E assigns floating point registers to the concrete region, because KLEE and the underlying constraint solver[2] do not yet support floating point operations on symbolic data.

The translated code accesses the CPU state directly by dereferencing the pointer to the CPU state or, in the case of native helpers, indirectly: read accesses to the symbolic CPU state are prepended with checks for symbolic data.

Sharing the Memory State QEMU emulates a memory management unit (MMU) to handle all guest memory accesses. The MMU translates virtual memory addresses into physical addresses. The TLB caches the result of the translation to speed up the translation on repeated accesses to the same pages. In QEMU, the TLB is a direct-mapped cache where each entry holds an offset that,

[2]The constraint solver decides whether path constraints associated with each branch outcome are satisfiable and, if so, allows the symbolic execution engine to continue execution along that outcome.

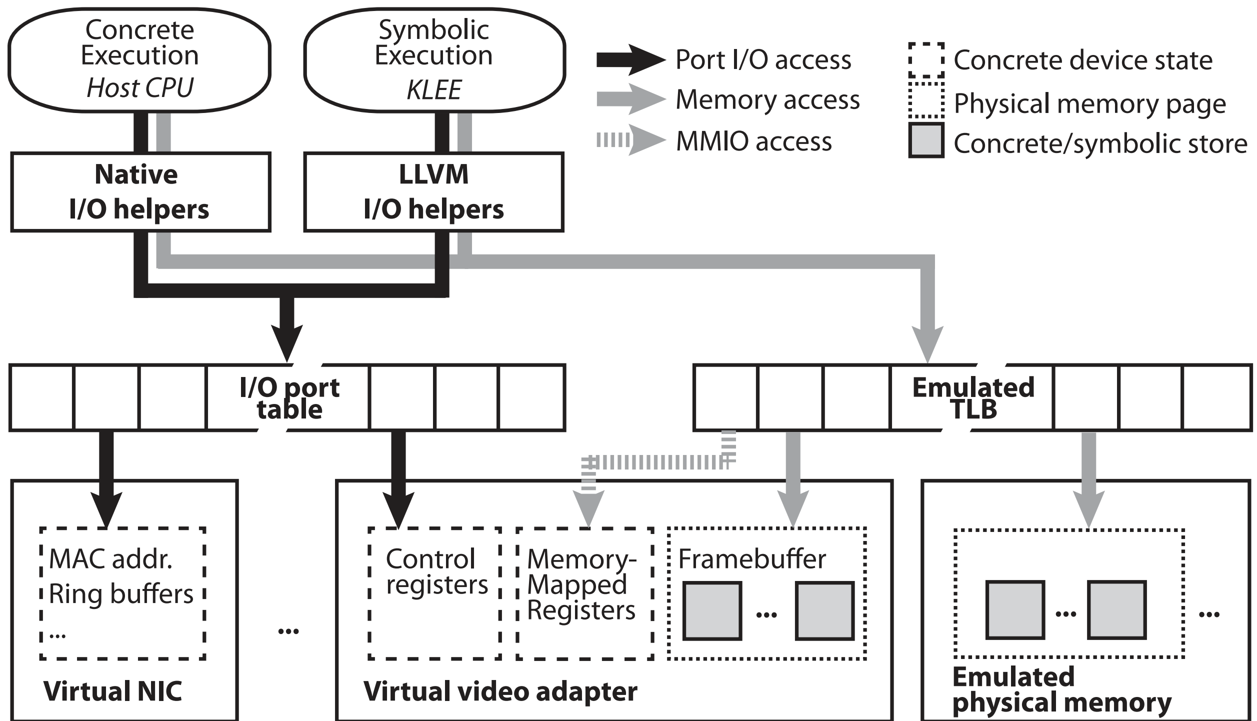

Figure 3.2 – Data paths to shared memory and device state.

when added to the virtual address of a memory page, results in the physical address inside QEMU's address space where the data for that page is stored. Each TLB entry also contains information about access permissions and whether the memory page belongs to an emulated device.

S^2E extends the TLB with pointers to `ObjectState` structures in order to support symbolic memory. `ObjectState` structures store the actual concrete and/or symbolic data of the memory pages[3]. When native code is running, the MMU checks whether memory reads would return symbolic data by looking at the `ObjectState`'s bitmap. If yes, the MMU instructs the execution engine to abort the execution of the current translation block and to re-execute the memory access in symbolic mode. In symbolic mode, the engine retrieves the `ObjectState` that corresponds to the physical address stored in the TLB entry and proceeds with the memory operation.

Sharing the Device State A device performs operations on state and produces output visible to the machine the device is attached to. In real hardware, the state consists of the contents of all internal registers (stored in flip-flops) and memory (e.g., DRAM chips). Virtual devices in QEMU emulate the behavior of the real devices: device state is kept in host memory, and the device's functionality is implemented by software running on the host CPU. QEMU supports both memory-mapped (MMIO) and port I/O-based devices.

[3]One memory page can be split in multiple `ObjectState` structures, in order to optimize access times, as shown in §3.5.

S^2E modifies the QEMU *block layer* to support consistent disk state. Virtual block devices (e.g., hard disks and CD drives) provide storage to guests, which is backed by files stored on the host. Virtual block devices access the host files via the QEMU block layer. S^2E modifies the block layer to redirect to a state-local buffer all writes to the host files. When the guest OS issues a read request, S^2E returns the latest write from the buffer. If there were no writes, S^2E forwards the request to the block layer. This ensures that all execution states see a consistent disk state and do not clobber each other's writes by writing to a shared disk file. Failing to provide consistent disk state quickly leads to file system corruption, resulting in guest OS crashes.

3.3.2 Symbolic Hardware

To support whole-system symbolic execution, S^2E extends the virtual hardware with symbolic devices (e.g., to enable analysis of low-level code such as device drivers) and introduces a per-state virtual clock, to ensure that the guest sees a coherent time.

Symbolic Devices A symbolic device is a special device that discards all writes, returns a symbolic value on every read, and triggers symbolic interrupts[4]; in other words, it does not implement specific functionality. S^2E instruments port I/O, MMIO, and physical memory accesses (for DMA-d memory) in order to determine on which read to return a symbolic value. To support symbolic reads for port I/O and MMIO, S^2E extends QEMU's emulation helpers. If a given port belongs to a symbolic device, S^2E returns a symbolic value on reads and discards writes. MMIO helpers are similar: each TLB entry contains a flag that specifies whether the memory page is mapped to physical memory or to a device, and is directed to the device emulation helpers as needed. These helpers return symbolic values on reads, exactly like for port I/O. To handle DMA, when the TLB entry of a memory page involved in a DMA transfer is loaded, S^2E modifies the flag in order to invoke MMIO emulation helpers whenever this memory page is accessed; in these helpers, if the access indeed falls inside the DMA region, a symbolic value is returned.

Supporting symbolic interrupts does not require any modification to QEMU. Triggering such interrupts consists of asserting the interrupt pin of the virtual device at the desired moment. This can be readily done by QEMU, which has different mechanisms to assert interrupts for each class of devices (e.g., for PCI, ISA, and USB devices). At which point in an execution to trigger the interrupt is decided by the S^2E plugins.

These mechanisms enable selection plugins to implement arbitrary symbolic devices. S^2E comes with a *SymbolicHardware* plugin that implements symbolic PCI and ISA devices. For ISA devices, the plugin registers port I/O ranges, MMIO, and DMA regions according to the user's

[4]A symbolic interrupt is an interrupt with a symbolic arrival time.

configuration. For PCI devices, the plugin lets the user specify the device and vendor identifiers, as well as I/O and MMIO regions, interrupt channels, and all other fields available in a PCI descriptor. The plugin uses this information to instantiate an "impostor" PCI device that will induce the guest OS to load the appropriate device driver. Then, whenever the driver accesses the device, S^2E returns symbolic data.

Enabling DMA regions and symbolic interrupts is done with support from analysis plugins: they monitor the OS kernel, catch invocations of DMA-related APIs (e.g., registration of DMA regions), and pass address ranges to selector plugins (e.g., *SymbolicHardware*) that then register these regions through the S^2E API. Likewise, an analysis plugin can help determine when to trigger symbolic interrupts. For example, DDT^+, an automated testing tool for proprietary drivers, triggers such interrupts on every call to the kernel API in order to maximize the chances of exposing concurrency bugs. REV^+, a reverse engineering tool, triggers symbolic interrupts after exercising the send entry point of a network card, in order to maximize the coverage of the interrupt handler.

Per-State Virtual Clock QEMU maintains two types of clocks: a host clock and a virtual clock. The *host clock* reflects the current time of the host machine. The host clock is used by QEMU's virtual real-time clock device in order to provide the guest OS with a time source synchronized with the host machine. The *virtual clock* stores the number of ticks elapsed from the start of the system (i.e., when the VM was turned on). Unlike the host clock, the virtual clock is periodically incremented but not synchronized with the host machine's time.

Since S^2E splits "reality" into multiple executions, it must correspondingly offer multiple timelines. For this reason, S^2E maintains a separate virtual clock for *each* system state and does not rely on the host clock. S^2E increments the virtual clock of the state of the currently running path and keeps the respective clocks frozen in all other states. This way, the guest OS is given (a sufficiently good) illusion that the execution of those paths never stopped.

S^2E slows down the per-state virtual clock when running in symbolic mode. Interpreting LLVM instructions in KLEE is slower than running native code, and frequent timer interrupts make progress even slower. In practice, a new interrupt arrives after every translation block that runs in the symbolic interpreter. It is therefore not enough to disable timer interrupts and restore them after S^2E finishes interpreting the LLVM code. Instead, S^2E applies to the host clock a time dilation factor that is equal to the slowdown caused by the LLVM interpreter. This delays the scheduling of the next timer interrupt further enough in time so that execution of the program under analysis can make sufficient progress.

3.3.3 Multiplexing Per-Path States

S^{2}E executes one path at a time and switches between paths to allow executions to progress in parallel. Since each execution path is characterized by its state, S^{2}E switches execution paths by switching states. The challenge is to save and restore QEMU-specific *concrete* state (i.e., virtual devices and concrete CPU state) as well as to properly manage the translation block cache.

S^{2}E explicitly copies the *concrete* region of the CPU state to/from QEMU's heap. Before S^{2}E is initialized, QEMU allocates a `CPUState` structure on the heap. Although S^{2}E stores the CPU state in an `ObjectState` structure, which LLVM helpers and symbolically running code access transparently, parts of QEMU also directly access the concrete region by dereferencing `CPUState` pointers (e.g., from the DBT). Finding and instrumenting all accesses to redirect them to the `ObjectState` is error-prone and unmaintainable (e.g., when upgrading QEMU versions). Therefore, S^{2}E leaves all the accesses unchanged (i.e., lets QEMU access the `CPUState` on the heap) and, during state switch, S^{2}E saves the concrete content on the heap in the `ObjectState` of the active execution state, fetches the new state, and overwrites the structure on the heap with the new `CPUState` data.

S^{2}E relies on QEMU's snapshot mechanism to automatically save and restore concrete virtual device data structures. QEMU uses snapshots to suspend and resume the virtual machine: S^{2}E redirects all writes and reads to/from the snapshot file to a per-path buffer. When S^{2}E is about to switch states, it calls QEMU to go through the list of all virtual devices and save their internal data structures. Then, S^{2}E selects the next execution state and restores the state of the virtual devices by calling `vmstate_load`.

Users can configure S^{2}E to not preserve the per-path device state upon state switching and let devices share their state between all execution paths, as done in KLEE. This causes inconsistencies, but reduces memory usage. For example, disabling state saving for the framebuffer avoids recording a separate `ObjectState` (multiple MBs) for each state and copying this data between the heap and the `ObjectState`. This makes for intriguing visual effects on-screen: multiple erratic mouse cursors and BSODs blend chaotically, providing free entertainment to the S^{2}E user.

Since different states may execute different code at the same address, stale code might end up being executed if the translation block cache is not flushed on state switches. However, since many programs do not change their code at run-time, disabling flushing makes sense, since it improves emulation speed. We plan to make the translation block cache state-local, in order to avoid unnecessary flushes.

3.4 Plugin Infrastructure

The S^{2}E plugin infrastructure connects selector and analyzer plugins via events, as will be described in Chapter 4. An S^{2}E plugin is a C++ class that subclasses the `Plugin` base class, which in turn registers the plugin with S^{2}E, automatically checks that plugin dependencies are satisfied, and provides an API to retrieve the instance of other plugins in order to communicate with them. At initialization, a plugin must subscribe to at least one core event; it can also subscribe to events exported by other plugins. A plugin can later modify its subscriptions from its event handlers.

We wrote an event library that defines *signals* (the S^{2}E events) to which it is possible to connect *callbacks* (the event subscribers). We originally used the `libsigc++` [103] library for the plugin infrastructure, but it incurs an unacceptable performance overhead, because it calls memory allocation routines during signal invocation. S^{2}E plugins can trigger signals at a high rate (up to thousands of signal invocations per second). For example, it took 250 seconds to open the Windows control panel while using the *FunctionMonitor* plugin (12 seconds without the plugin). The new implementation reduced the overhead to 25% (15 seconds).

S^{2}E instruments translated code to generate run-time events. For each guest instruction that the DBT translates, S^{2}E invokes the *onInstrTranslation* event, described in §4. One parameter of this event is a pointer to a list of callbacks. Subscribers that want to be notified every time that a guest instruction is executed append their callback to that list. After S^{2}E processes all subscribers of *onInstrTranslation*, S^{2}E saves the list of *onInstrExecution* callbacks in the translation block and inserts a micro-operation that triggers the invocation of a specific emulation helper every time that instruction is executed. This emulation helper goes through the list stored in the translation block and invokes the callbacks.

S^{2}E extends the x86 instruction set with custom instructions that trigger events. S^{2}E uses the opcode `0x0f 0x3f` for custom instructions, which is unused according to the Intel instruction set manual [61]. In S^{2}E, this opcode is followed by an 8-bytes operand that is freely definable by the plugins. The DBT translates this opcode into a call to the S^{2}E custom instruction emulation helper and passes the operand as a parameter. At run-time, the helper invokes all the callbacks registered by the subscribers of the *onCustomInstruction* event, the subscribers check the operand and perform whatever action is appropriate. Note that executing on a normal machine a program instrumented with S^{2}E opcodes would trigger an invalid instruction exception.

S^{2}E triggers all other events without requiring the translated code to be instrumented. For example, S^{2}E triggers the *onTimer* event from QEMU's timer handler in order to allow plugins to process periodic events. Likewise, S^{2}E triggers *onException*, *onExecutionFork*, and *onTlbMiss* from the exception emulation helpers, KLEE, and the MMU, respectively.

3.5 Key Optimizations

In this section, we describe five optimizations that have brought the greatest improvement in S^2E's performance: pervasive use of copy-on-write, lazy concretization, aggressive simplification of symbolic expressions, optimized handling of symbolic pointers, and multi-core parallelization.

Copy-on-Write Copy-on-write (COW) minimizes memory usage by sharing as much data as possible between execution states. When a state is copied upon path splitting, the child states share the data stored in the parent. When a write occurs, S^2E copies the data from the parent to the child that initiated the write. S^2E splits the physical memory into multiple `ObjectState` structures and then reuses KLEE's COW mechanisms. For all other devices, S^2E does not use COW because device state is small (a few KBs per state) and in practice, every execution path modifies the state of virtually every device.

Lazy Concretization S^2E employs *lazy concretization*: it concretizes a symbolic value x on-demand, only when code that runs in the concrete domain is about to branch on the value of x. This is an important optimization when doing in-vivo symbolic execution, because a lot of data can be carried through the layers of the software stack without conversion. For example, when a program writes a buffer of symbolic data to the filesystem, there are usually no branches in the kernel or the disk device driver that depend on this data. The buffer can therefore pass through unconcretized and be written in symbolic form to the virtual disk, from where it will eventually be read back in its symbolic form.

Expression Simplification Conversion from x86 to LLVM gives rise to complex symbolic expressions. S^2E "sees" a lower level representation of the programs than what would be obtained by compiling source code to LLVM (as done in KLEE): it actually sees the code that *simulates* the execution of the original program on the target CPU architecture. Such code typically contains many bitfield operations (such as `and`/`or`, `shift`) that manipulate bits in the `eflags` register.

To optimize these expressions, we built a bitfield expression simplifier that, if parts of a symbolic variable are masked away by bit operations, removes those bits from the corresponding expressions. First, the simplifier starts from the bottom of the expression's tree representation and propagates information about individual bits whose value is known. If all bits in an expression are known, S^2E replaces the expression with the corresponding constant. Second, the simplifier propagates top-down information about bits that are ignored by the upper parts of the expression—when an operator only modifies bits that upper parts ignore, the simplifier removes that entire operation.

We say a bit in an expression is known to be one (respectively zero), when that bit is not symbolic and has the value one (respectively zero). For example, if x is a 4-bit symbolic value,

the expression $x \mid 1000$ has its most significant bit (MSb) known to be one, because the result of an `or` of a concrete bit set to one and of a symbolic bit is always one. Moreover, this expression has no bits known to be zero, because the MSb is always one and symbolic bits `or`-ed with a zero remain symbolic. Finally, the ignore mask specifies which bits are ignored by the upper part of an expression. For example, in $1000 \,\&\, (x \mid 1010)$, the ignore mask at the top-level expression is 0111 because the `and` operator cancels the three lower bits of the entire expression.

To illustrate, consider the 4-bit wide expression $0001 \,\&\, (x \mid 0010)$. The simplifier starts from the bottom (i.e., $x \mid 0010$) and propagates up the expression tree the value $k_{11} = 0010$ for the known-one bits as well as $k_{10} = 0000$ for the known-zero bits. This means that the simplifier knows that bit 1 is set but none of the bits are zero for sure (because x is symbolic). At the top level, the `and` operation produces $k_{21} = 0000$ for the known-one bits ($k_{11} \,\&\, 0001$) and $k_{20} = 1110$ for the known-zero bits ($k_{10} \mid 1110$). The simplifier now knows that only the least significant bit matters and propagates the ignore mask $m = 1110$ top down. There, the simplifier notices that 0010 is redundant and removes it, because $1101 \mid m$ yields 1111, meaning that all bits are ignored. The final result is thus $1 \,\&\, x$.

We implemented this simplification in the early stage of expression creation rather than in the constraint solver. This way, we do not have to re-simplify the same expressions again when they are sent to the constraint solver several times (for example, as part of path constraints). This is an example of applying domain-specific logic to reduce constraint solving time; we expect our simplifier to be directly useful for KLEE as well, when testing programs that use bitfields heavily.

Symbolic Pointers A symbolic pointer is a pointer whose value depends on symbolic inputs, therefore referring to a range of memory locations (as opposed to a concrete pointer, which refers to only one particular address). Symbolic pointers commonly occur when indexing arrays, like in jump tables generated by compilers for switch statements. When a symbolic pointer is dereferenced, S^{2}E determines the pages referenced by the pointer and passes their contents to the constraint solver. Alas, large page sizes can bottleneck the solver, so S^{2}E splits the default 4KB-sized pages into smaller pages of configurable size (e.g., 128 bytes), so that the constraint solver need not reason about large areas of symbolic memory. In §6.4, we show how much this helps in practice.

S^{2}E can also concretize symbolic pointers to further reduce overhead. This is most useful in the case of switch statements and symbolic writes to the program counter register (which is always concrete in S^{2}E). S^{2}E uses binary search to determine to which interval the symbolic pointer belongs, and forks n states, each state having one concrete address that satisfies the path constraints. n is usually bounded, since the path constraints often limit the interval (e.g., switch statements have a limited number of cases). n can be user-configurable to avoid path explosion in case the symbolic pointer references a large memory range.

Multi-core S^{2}E S^{2}E explores different paths concurrently by running multiple S^{2}E instances in parallel. Whenever an execution path splits due to a symbolic condition, S^{2}E assigns the exploration of the newly created path to a new S^{2}E instance that runs on a different core. If all cores are already busy exploring paths, then the S^{2}E instance behaves like in single-instance mode: each split path is added to the local queue of the instance that split it. An S^{2}E instance terminates when it has explored all the paths in its queue, leaving the core available for new instances.

The simple parallelization algorithm used by S^{2}E does not address the issues of redundant exploration (i.e., two cores exploring identical states) and load balancing (i.e., moving a subset of states from one instance to another). This can be solved by combining S^{2}E with the Cloud9 [21] parallel symbolic execution engine. §6.1.4.5 analyzes the impact of this multi-core design on S^{2}E's performance.

S^{2}E uses the `fork` system call to run instances on multiple processors/cores. This system call maps naturally to the concept of execution path splitting in symbolic execution. Consider an execution path p that is explored by an S^{2}E process q. When p splits on a branch that depends on a symbolic value, S^{2}E creates a path p' and forks a child process q', which is an identical copy of the S^{2}E process q. The child process q' receives the execution path p', and the parent process q continues the execution of p. After the fork system call completes, each instance starts exploring an independent subtree. A similar approach is used by EXE [28] to implement symbolic execution.

S^{2}E plugins can be kept aware of the various running instances: S^{2}E triggers *onInstanceFork* whenever it creates a new instance. For example, the *Logger* plugin listens to this event to create a fresh execution trace file each time a new instance is created; this avoids expensive synchronization, yet writing traces to separate files does not burden offline processing tools: each file contains an independent subtree, and recreating the full tree through trace concatenation is straightforward.

3.6 Summary

In this chapter, we showed how S^{2}E combines virtualization, dynamic binary translation, native execution, and symbolic interpretation to give the illusion of whole-system symbolic execution. We explained how S^{2}E shares CPU, memory, and device state between native and symbolic execution, described how to efficiently implement the plugin infrastructure, and presented some of the key optimizations that make the S^{2}E approach feasible. We describe next how S^{2}E can be used to write new analysis tools.

Chapter 4

Developing New System Analysis Tools

S^{2}E is a platform for rapid prototyping of custom system analyses. It offers two key interfaces: the *selection* interface, used to guide the exploration of execution paths (and thus implement arbitrary consistency models), and the *analysis* interface, used to collect events or check properties of execution paths. Both interfaces accept modular selection and analysis plugins. Underneath the covers, S^{2}E consists of a customized virtual machine, a dynamic binary translator (DBT), and an embedded symbolic execution engine, as was described in the previous section. The DBT decides which guest machine instructions to execute concretely vs. which ones to interpret symbolically using the embedded symbolic execution engine.

S^{2}E provides many plugins out of the box for building custom analysis tools—we describe these plugins in §4.1. One can also extend S^{2}E with new plugins, using S^{2}E's developer API (§4.2). Figure 4.1 shows a snapshot of the S^{2}E plugins that are part of S^{2}E today.

4.1 Developing New Tools From Existing Plugins

In this section, we show how a developer can combine the various S^{2}E plugins in order to construct custom analysis tools, without writing any additional plugins. For instance, a developer could be a device driver tester performing quality checks during driver development. The developer would combine here various path selectors to limit multi-path exploration to the driver under test (§4.1.1) with path analysis plugins to check the driver for the presence of bugs (§4.1.2).

4.1.1 Path Selection

The first step in using S^{2}E is deciding on a policy for which part of a program to execute in multi-path (symbolic) mode vs. single-path (concrete) mode; this policy is encoded in a selector. S^{2}E provides a set of selectors for the most common types of selection, which fall into three categories:

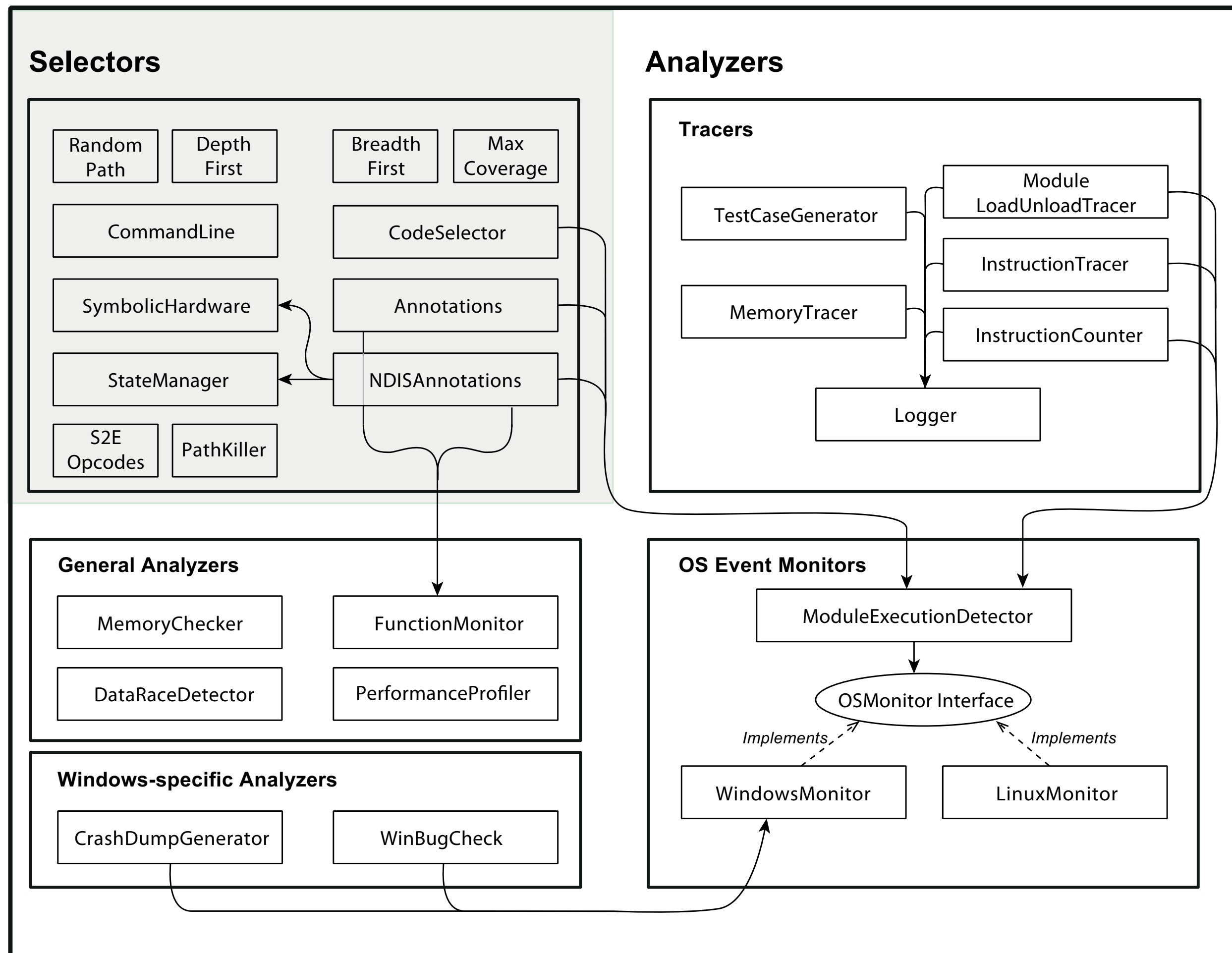

Figure 4.1 – S²E stock plugins. The arrows represent plugin dependencies (e.g., the *CodeSelector* plugin uses the functionality provided by *ModuleExecutionDetector*). To distinguish selectors from analyzers, we show the former on a shaded background.

Data-based selection provides a way to expand a regular execution into a multi-path one by introducing symbolic values into the system; then, any time S²E encounters a branch predicate that depends on symbolic values, execution will fork accordingly. Symbolic data can enter the system from various sources, and S²E provides a selector for each, ranging from command-line arguments using the *CommandLine* plugin to hardware input with the *SymbolicHardware* plugin.

Often it is useful to introduce a symbolic value at an interface that is internal to the system. For example, say a server program calls a library function `libFn`(x) almost always with $x = 10$, but may call it with $x < 10$ in strange corner cases that are hard to induce via external workloads. The developer might therefore be interested in exploring the behavior of `libFn` for all values $0 \leq x \leq 10$. For such analyses, S²E provides an *Annotations* plugin, which allows direct injection of custom-constrained symbolic values anywhere they are needed.

Code-based selection enables/disables multi-path execution depending on whether the program

counter is or not within a target code area; e.g., one might focus cache profiling on a web browser's SSL code, to see if it is vulnerable to side channel attacks. The *CodeSelector* plugin takes the name of the target program (or library, driver, etc.) and a list of program counter ranges. Code within these ranges should be explored in multi-path mode, while code that is outside should be run in single-path mode. *CodeSelector* is typically used together with data-based selectors to constrain the data-selected multi-path execution to within only code of interest.

Priority-based selection is used to define the order in which paths are explored within the family of paths defined with data-based and code-based selectors. S^2E includes basic policies, such as *Random*, *DepthFirst*, and *BreadthFirst*, as well as others. The *MaxCoverage* selector works in conjunction with coverage analyzers to heuristically select paths that maximize coverage. The *PathKiller* selector monitors the executed program and deletes paths that are determined to no longer be of interest to the analysis. For example, paths can be killed if a fixed sequence of program counters repeats more than n times; this avoids getting stuck in polling loops.

4.1.2 Path Analysis

Once the family of paths to be analyzed is defined via the choice of selector(s), the developer needs to choose the analyzer(s) to which S^2E will expose the chosen paths.

An analyzer is a piece of logic that checks for properties along execution paths. For example, a bug finder is an analyzer that may check for various types of crashes or assertion violations along the executed paths. A performance profiler is also a type of analyzer that checks for properties such as the number of cache misses along a path or the TLB hit count. S^2E has several multi-path analysis plugins, such as performance profilers, memory checkers, crash detectors, or tracers.

S^2E also lets developers take advantage of existing unmodified off-the-shelf single-path analysis tools, such as Valgrind, Oprofile [80], or Microsoft Driver Verifier [87], which can be wrapped into a plugin that adapts the output of these tools to the S^2E API (e.g., such a plugin could terminate every path reported as faulty by Microsoft's Driver Verifier).

Reusing Existing Single-Path Analysis Tools We illustrate how S^2E can reuse existing tools with the example of Valgrind and Oprofile. Valgrind instruments programs in order to analyze their cache behavior, memory safety, execution times, and call graphs. Oprofile is a sampling-based profiler that can analyze an entire software stack, including the OS kernel.

Both of these tools are single-path and rely on testers to guess the concrete inputs that would drive a program down a path of interest. In other words, testers have to design test cases that exhibit the behaviors to be studied, such as bugs or slowdowns. In contrast, S^2E automatically enumerates various execution paths and exposes these tools to them. Off-the-shelf tools are not

aware of the multi-path exploration: they just operate as usual, without any modification in their logic, but ultimately yield multi-path results.

However, S^2E does not remove the limitations of the existing tools. For example, Valgrind is still limited to profiling user-mode processes and cannot analyze the kernel, while Oprofile is still subject to imprecise results because it is based on sampling. Moreover, these tools remain single-path in nature: they cannot reason about multiple paths at a time. For instance, Valgrind cannot tell whether the path it has just run has the lowest instruction count. For this, a Valgrind-based plugin would need to be modified to look at all the paths that were explored so far.

Multi-Path Analyzers When off-the-shelf tools are not enough, users can employ S^2E analysis plugins. Plugins run outside of the guest OS and can observe the entire system state, without interfering with the software under analysis.

One class of analyzers are bug finders, such as the *WinBugCheck* and *MemoryChecker* plugins, which look for Windows kernel crashes and memory errors, and output the execution paths leading to the encountered bugs. Another class of analyzers are execution tracers, such as *InstructionTracer*, which selectively records the instructions executed along a path, or *MemoryTracer*, which logs memory accesses and hardware I/O. Tracing can be used for many purposes, like offline coverage measurement or profiling. Finally, the *PerformanceProfiler* analyzer counts cache misses, TLB misses, and page faults incurred along each path—this can be used to obtain the performance envelope of an application. We describe it in more detail in the evaluation (chapter 6).

While most plugins are OS-agnostic, S^2E also includes a set of analyzers that intercept Windows-specific events using undocumented interfaces or other hacks. For example, *WindowsMonitor* parses and monitors Windows kernel data structures and notifies other plugins when the kernel loads a driver, a library, or an application. Another example is the *CrashDumpGenerator* plugin, which generates a memory dump compatible with Microsoft WinDbg.

Offline Multi-Path Analyzers S^2E provides plugins for collecting execution traces and saving them to a file for offline analysis. This is useful for complex analyses that are hard to do online, such as reverse engineering of device drivers (§6.1). The core tracing plugin *Logger* provides an interface to other plugins that they can use to log arbitrary data. *Logger* wraps the written data into a trace item object containing a path identifier, timestamp, size, and the plugin that wrote the item. This allows offline analysis tools to focus only on trace items of interest.

The collected traces can be parsed by offline analysis tools to reconstruct the execution tree, walk through all trace items on a given path, and perform analyses on them. For example, a tool that measures code coverage would look for trace items written by the *InstructionTracer* plugin to determine which instructions were executed. That tool would also rely on the module information

provided in the trace by the *ModuleLoadUnloadTracer* plugin in order to associate each program counter with the corresponding module and display relevant debug information.

4.1.3 Configuration Interface

Developers can combine plugins using the S^{2}E configuration interface, which accepts scripts written in the Lua language [81]. Scripting offers an easier alternative to writing C++ plugins. For each selector and analyzer used, there is a section in the script that lets the user control the plugin's behavior. For instance, users can configure a data selector plugin to write symbolic values to some memory location after the system executes a particular function (e.g., users may want to fill a freshly `malloc`-ed buffer with symbolic values in order to track uses of uninitialized data).

4.2 Developing New Plugins

We now describe the interface that can be used to write new plugins or to extend the default plugins described above. Both selectors and analyzers use the same interface; the only distinction between selectors and analyzers is that selectors influence the execution of the program, whereas analyzers are passive observers of the selected execution paths.

Plugin Interface S^{2}E has a modular architecture, in which plugins communicate via events in a publish/subscribe fashion. S^{2}E events are generated either by the S^{2}E platform or by other plugins. To register for a class of events, a plugin invokes *regEventX(callbackPtr)*; the event callback is then invoked every time *EventX* occurs, and it is passed parameters specific to the event.

Table 4.1 shows the *core events* exported by S^{2}E that arise from regular code translation and execution. We chose these core events because they correspond to execution at the lowest possible level of abstraction: instruction translation, execution, memory accesses, and state forking.

Execution Path Abstraction For each path being explored, there exists a distinct *ExecutionState* object instance; when an execution path splits (or forks), each child execution receives its own private copy of the parent *ExecutionState*. The *ExecutionState* object captures the current state of the entire virtual machine along a specific individual path. It is the first parameter of every event callback. *ExecutionState* enables plugins to toggle multi-path execution on/off and gives them read/write access to the entire VM state, including the virtual CPU, VM physical memory, and virtual devices (see Table 4.2 for some of the *ExecutionState* object methods). A plugin can obtain the PID of the running process from the page directory base register, can read/write page tables and physical memory, can change the control flow by modifying the program counter, and so on.

Event	Description
onInstrTranslation	The DBT is about to translate a machine instruction. A plugin can use this event to mark instructions of interests (e.g., calls or returns).
onInstrExecution	The VM is about to execute a marked instruction. This event invokes the callbacks registered via *onInstrTranslation*.
onExecutionFork	S^2E is about to split (fork) the current execution path in two. Mainly used by tracing plugins to embed the execution tree in trace files.
onInstanceFork	S^2E is about to spawn a new process instance of itself in order to explore the forked path on a different CPU core (more details in §3.5).
onException	The VM interrupt pin has been asserted. Provides a convenient means of intercepting interrupts and exceptions (e.g., fatal double faults).
onMemoryAccess	The VM is about to execute a memory access. This event can be used to simulate a cache hierarchy, record a memory trace for offline analysis, etc.
onPortAccess	The VM is about to execute a port I/O operation.
onCustomInstruction	The VM is about to execute a custom opcode. Listening plugins parse the custom instruction's operands to decode the action to perform.
onPageFault	A page fault occurred in the guest code.
onTlbMiss	A TLB missed occurred in the memory management unit. Using this event is faster than checking for misses every time *onMemoryAccess* fires.
onTimer	Timer event to let plugins implement periodic tasks, such as flushing trace files, periodically terminating uninteresting paths, etc.

Table 4.1 – Core events exported by the S^2E platform.

Plugins partition their own state into per-path state (e.g., number of cache misses along a path) and global state (e.g., total number of basic blocks touched). The per-path state is stored in a *PluginState* object, which hangs off of the *ExecutionState* object. *PluginState* must implement a *clone* method, so that it can be cloned by S^2E together with *ExecutionState* whenever execution forks. Global plugin state can live in the plugin's own heap.

The dynamic binary translator (DBT) turns blocks of guest code into corresponding host code; for each block of code this is typically done only once. During the translation process, a plugin may be interested in instrumenting certain instructions (e.g., function calls) for subsequent notification. It registers for *onInstrTranslation* and, when notified, it inspects the *ExecutionState* object to see

`void setForking(bool enable)`	Turn on/off multi-path execution
`void read/writeMemory(uint64_t addr, Expr *buffer, size_t length)`	Read or write contents of memory (symbolic or concrete) at address *addr*
`Expr readReg(int reg)`	Read *val* (symbolic or concrete) from *reg*
`void writeReg(int reg, Expr val)`	Write *val* (symbolic or concrete) to *reg*
`TranslationBlock *getTb()`	Get currently executing code block from DBT
`PluginState *getPluginState(Plugin *plugin)`	Get per-path state object for the specified plugin instance

Table 4.2 – A subset of the *ExecutionState* object's interface.

which instruction is about to be translated; if it is an instruction of interest (say, for example, a CALL), the plugin instruments it. Whenever the VM executes an instrumented instruction, it raises the *onInstrExecution* event, which notifies the corresponding plugin. For example, the *CodeSelector* plugin is implemented as a subscriber to *onInstrTranslation* events; upon receiving an event, it instruments the instruction depending on whether it is or not an entry/exit point for a code range of interest. When such an instruction gets subsequently executed, having the *onInstrTranslation* and *onInstrExecution* events separate leverages the fact that each instruction gets translated once, but may get executed millions of times (e.g., as in the body of a loop). For most analyses, *onInstrExecution* ends up being raised so rarely that using it introduces no runtime overhead (e.g., catching the kernel panic handler requires marking only the first instruction of that handler).

Custom Instructions *S^{2}E opcodes* are custom guest machine instructions that are directly interpreted by S^{2}E. These form an extensible set of opcodes for creating symbolic values (S2SYM), enabling/disabling multi-path execution (S2ENA and S2DIS) and logging debug information (S2OUT). They give developers the finest grain control possible over multi-path execution and analysis; they can be injected into the target code manually or with the help of binary instrumentation tools like PIN [82]. In practice, opcodes are the easiest way to mark data symbolic and get started with S^{2}E, without involving any plugins.

The code fragment in Figure 4.2 shows a C function that writes unconstrained symbolic values to the *buf* buffer. Symbolic values can be given a name, e.g., to improve readability when printing an expression that involves symbolic values. Note that S^{2}E opcodes do not require specialized compiler or assembler support, since most compilers and assemblers can already emit arbitrary byte sequences within the generated code.

```
void s2e_make_symbolic(void* buf, int size, const char* name)
{
    __asm__ __volatile__(
        /* Binary encoding for S2SYM */
        ".byte 0x0f, 0x3f\n"
        ".byte 0x00, 0x03, 0x00, 0x00\n"
        ".byte 0x00, 0x00, 0x00, 0x00\n"
        : : "a" (buf), "b" (size), "c" (name) : "memory"
    );
}
```

Figure 4.2 – Embedding S^{2}E custom instructions in C programs.

The interface presented here was sufficient for all the multi-path analyses we attempted with S^{2}E. Selectors can enable or disable multi-path execution based on arbitrary criteria and can manipulate machine state. Analyzers can collect information about low-level hardware events all the way up to program-level events, they can probe memory to extract any information they need, and so on. We now provide two examples to illustrate the use of S^{2}E plugins.

4.2.1 Building the *Annotations* Plugin

An S^{2}E annotation is a piece of code written by an S^{2}E user in order to observe and manipulate the execution state. Therefore, annotations may be used for what might be called system-wide aspect-oriented programming, in which any instruction sequence can be preceded/followed/replaced by any other sequence of instructions.

Annotations can be used to implement different execution consistency models. Therefore, the *Annotations* plugin is a central piece in tools like DDT$^+$ (§6.3.1) and REV$^+$ (§6.1). For example, annotations can implement the RC-LC consistency by carefully replacing some function parameters and return values with symbolic data.

Appendix A details the steps that an S^{2}E plugin developer would take to develop the *Annotations* plugin as well as how an S^{2}E user would use such a plugin to perform the analysis of a Windows network device driver. In particular, we show how to implement *Annotations* in a platform-independent manner, making it suitable to analyze any kind of code on arbitrary guest operating systems. For this, we explain how an S^{2}E plugin developer could break down its functionality into smaller plugins that can be used independently or in combination by an S^{2}E user.

4.2.2 Combining S^{2}E Plugins and In-VM Tools

In this second example, we show an alternate way of implementing annotations with in-VM tools, using SystemTAP [100]. SystemTAP is a tracing framework for Linux that can intercept any function call or instruction in the kernel and invoke custom scripts. The scripts have full access to the system state. They can also leverage debug information to access variables by name.

S^{2}E users can leverage SystemTAP to obtain a flexible way of controlling path exploration. Users write SystemTAP scripts with embedded calls to S^{2}E opcodes. This allows injecting symbolic values at any place, terminating states based on complex conditions, interacting with S^{2}E plugins, and more generally developing arbitrary selection schemes directly inside the guest OS.

Suppose we want to analyze the behavior of the Linux network stack when a network packet is received (e.g., check whether there is a packet that could crash the kernel). One approach is to replace the content of the incoming packets with symbolic values, in order to explore all the paths that depend on the packet's content.

Injecting symbolic packets in the Linux kernel can be done in a few lines of code with SystemTAP [100], as shown in Figure 4.3. We define a SystemTAP*probe* that intercepts calls to `netif_receive_skb`. Network drivers call this function when they are ready to pass incoming packets to the kernel. Besides the probe, the SystemTAP script also contains a call to the `s2e_make_symbolic` function. This function is the same as the one in Figure 4.2, except that it uses the SystemTAP syntax.

Note that it is also possible to use the *Annotations* plugin to perform this analysis, because the concept of annotation is similar to the SystemTAP probes. S^{2}E gives users the freedom to choose any method that is the most convenient for them to carry out a given analysis. For example, users may choose to adapt their existing SystemTAP scripts instead of rewriting them using the *Annotations* plugin's configuration syntax. Likewise, users may employ the *Annotations* plugin if the guest OS does not have an equivalent of SystemTAP or if the use of such a tool interferes with some aspect of the analysis (e.g., performance profiling).

```
function s2e_make_symbolic(buf: long, size:long, name:string) %{
    #SystemTap allows arbitrary C code (including inline assembly)
    __asm__ __volatile__(
      ".byte 0x0f, 0x3f\n"
      ".byte 0x00, 0x03, 0x00, 0x00\n"
      ".byte 0x00, 0x00, 0x00, 0x00\n"
          : : "a" ((uint32_t)THIS->buf),
              "b" ((uint32_t)THIS->size), "c"(THIS->name)
          : "memory"
    );
%}

#Insert symbolic values on each invocation of netif_receive_skb
probe kernel.function("netif_receive_skb") {
    s2e_make_symbolic($skb->data, $skb->len, "symbolic packet");
}
```

Figure 4.3 – Example of a SystemTAP probe that injects symbolic data into network packets.

4.2.3 Summary

In this chapter, we showed how S^{2}E users can combine various S^{2}E plugins to carry out the desired analysis tasks and how S^{2}E developers can write custom plugins using the S^{2}E developer API. An S^{2}E user can combine path selection plugins to limit the multi-path exploration to the modules of interest with different analysis plugins, such as bug finders, performance profilers, and execution tracers. We explained how S^{2}E turns existing single-path analysis tools, such as Valgrind and Microsoft Driver Verifier, into multi-path analyzers without any modification. Finally, we showed how developers can write modular plugins by taking the example of the *Annotations* plugin, which is a central piece of tools like REV$^+$ and DDT$^+$.

Chapter 5

Decompiling Program Binaries to LLVM

S^{2}E is a platform that lets developers build multi-path dynamic analysis tools by combining path analyzers and path selectors. S^{2}E exposes each execution path to analysis plugins. The order in which S^{2}E enumerates execution paths is dictated by path selection plugins (§4.1.1). Path selection plugins inspect the system state in order to make decisions regarding which paths to explore next.

A purely dynamic approach makes certain types of analysis and path selection algorithms difficult to implement. The S^{2}E engine uncovers new code and makes it available to analysis plugins *dynamically*, i.e., as it translates and executes that code. Plugins can therefore only access and inspect code that has already been translated by the execution engine. In particular, they do not have a global view of the control flow graph (CFG). The CFG is often used as a navigation map by path selectors. Without a full CFG, a heuristic that aims, e.g., to prioritize paths that lead to a certain function might end up exploring parts of the CFG that never call that function.

We add a static analysis component to S^{2}E in order to allow writing more powerful analysis and path selection plugins. For example, static analysis can prove that certain program states are unreachable, thus helping symbolic execution avoid wasting time trying to reach those states. It can also guide symbolic execution and prioritize path exploration by providing, e.g., a list of potential bugs or a set of loops that path selectors should exit early.

The static analysis component, called RevGen, takes as input an x86 binary and outputs an equivalent binary in LLVM format. As we shall see later, using LLVM as an intermediate representation allows reusing a wealth of existing LLVM-based analysis tools. RevGen allows reusing these tools either as is on the translated binary or in combination with S^{2}E, enabling developers to write more effective path analysis and path selection plugins.

The rest of this chapter is organized as follows. First, we show the usefulness of LLVM as an intermediate representation for program binaries (§5.1.1). Second, we show examples of use cases that RevGen enables (§5.1.2). Then we expose the challenges RevGen faces (§5.2), present the design and implementation (§5.3), expose preliminary results (§5.8), discuss (§5.9), and conclude.

5.1 Motivation

5.1.1 LLVM as Intermediate Representation for Analysis of Binaries

There exist many powerful tools for various types of code analysis. For example, BitBlaze [117] combines dynamic and static analysis components to extract information from malware. CodeSurfer [4] can perform program slicing, to allow understanding code behavior. Calysto [3] is a static bug finder and `bddbddb` [76] provides a framework for querying programs for buggy code patterns.

Unfortunately, most of these tools require source code. Coverity [12], `bddbddb`, Saturn [46], and various methods based on abstract representation [13] require C code. Other tools like Java PathFinder [102] or CoreDet [10] rely on Java and LLVM compilers to transform the source code to their analysis format.

The reliance on source code leaves a significant portion of legacy and proprietary software unanalyzed. Even when the source is partially available, parsing it can be challenging [12] and the presence of binary libraries or even inline assembly can severely degrade the performance of both static and dynamic analysis tools. Bug finding and debugging tools like KLEE [27] and ESD [135] cannot work on such programs.

There also exist tools that directly analyze machine code, but they often use ad-hoc intermediate representations (IR), making it hard to extend them to other architectures and preventing easy reuse of analysis components. An IR abstracts the source language (e.g., C or assembly) to facilitate analysis. For example, CodeSurfer is based on the IR generated by the proprietary IDAPro [60] disassembler, while Jakstab [68] relies on the frontend of the Boomerang [14] decompiler, and Vine, the static analysis component of BitBlaze, uses yet another representation.

In recent years, LLVM gained a wide popularity, becoming a platform of choice for developing new source-based analysis tools, and arguably imposing its IR as a *de facto* standard for such tools. Currently, more than 160 LLVM-based projects exist [77], with numerous static analysis tools targeted at software verification [115, 38, 8, 134, 115], as well as instrumentation tools enforcing safety properties at run-time, like deterministic execution [10], dynamic bug finders [74, 3], or safe execution of error recovery code [54]. LLVM is now actively supported by Apple and forms the basis of several commercial applications, e.g., MacOS and Xcode.

Several powerful analysis frameworks have been built with LLVM. KLEE looks for bugs in programs using symbolic execution, a method for thorough path exploration. KLEE found deep bugs in Coreutils that were overlooked for a decade. Parfait [38] is an LLVM-based static analysis framework that scales to millions of lines of code using demand-driven analysis. Finally, LLBMC [115] is a tool that applies bounded model checking to LLVM programs.

5.1.2 Use Cases

In this section, we illustrate how RevGen can be used in practice with existing analysis tools that are based on LLVM and that implicitly rely on the availability of the source code. We show the use cases of deterministic program execution, bug finding in kernel-mode binaries using static analysis, reverse engineering of device drivers for safety and portability, inline assembly removal, and analysis of embedded software.

Debugging multi-threaded programs Multi-threaded programs are particularly prone to bugs. Threads share data and use synchronization mechanisms, which can potentially lead to data races and deadlocks. The difficulty of debugging these problems is compounded by the presence of synchronizations implemented in an ad-hoc way [128]. Tools like CoreDet [10] and SyncFinder [128] make debugging of concurrency bugs easier. However, they only run on LLVM code.

RevGen allows SyncFinder to annotate blocks of binary code that use ad-hoc synchronization. SyncFinder locates the loops in the LLVM code, analyzes exit conditions, determines which blocks of code can run concurrently, and whether the exit condition can be affected by concurrent writes. If it is the case, SyncFinder reports an ad-hoc synchronization.

Likewise, RevGen enables the use of CoreDet on binary programs. CoreDet is a compiler and runtime environment that instruments multi-threaded programs in order to make them behave deterministically. CoreDet ensures that all conflicting concurrent stores are performed in a specific sequence and that threads are created and scheduled in a fixed order, while introducing as little serialization as possible.

Analyzing kernel-mode code Proprietary binary drivers are a major source of system crashes and unreliability. On Linux, error rate in drivers is 3-7 times higher than in the rest of the kernel [35]. Windows drivers are no better, causing 85% of crashes [97]. Since drivers usually run in kernel mode at the highest privilege level, exploiting their bugs can lead to complete denial of service and full system compromise.

By converting binary drivers to LLVM, RevGen would enable the use of static analysis tools on such drivers. LLBMC [115] is a static analysis tool that checks properties like integer overflows, illegal memory accesses, buffer overflows, or invalid bit shifts. Its abilities make it one of the first choices to verify device drivers.

RevGen also enables static analysis of low-level OS code. Such code typically uses machine instructions that have no equivalent in programs written in high-level languages. The challenge is to accurately emulate these instructions using the LLVM IR in order to make them amenable to static analysis.

Reverse engineering safe drivers Static analysis tools are useful to check the quality of drivers but they cannot fix buggy drivers by themselves. Moreover, such tools are of little help to users who are often forced to load faulty drivers because there is no better choice. Even if run-time driver bug containment tools exist [120], they incur overhead and are limited to a few OSes. Ideally, there should be a tool that automatically fixes buggy drivers.

RevNIC [33] uses reverse engineering to synthesize safer drivers from buggy ones. RevNIC takes a binary driver and traces its execution to observe all the ways in which the driver interacts with the hardware. The traces contain LLVM instructions complemented with dynamic I/O, memory, and register data, that RevNIC uses to encode the hardware-interaction state machine.

RevGen can be used to improve the synthesized drivers. RevNIC has low code coverage on complex device drivers, resulting in incomplete LLVM code and reduced driver functionality, which forces developers to manually write the missing code. RevGen can help automatically transform the missing code to LLVM, minimizing manual intervention. We will discuss RevNIC in more details in §6.1.

Helping source-based tools LLVM supports native inline assembly, whose presence prevents most of the state-of-the-art analysis tools from running properly. To analyze such functions accurately, analysis tools must precisely model the semantics of each machine instruction (i.e., what the instruction does). Failing to do so may cause both false negatives and false positives. For example, KLEE [27] aborts execution paths that have inline assembly and static analysis tools either ignore it or make unsound assumptions about such code [12].

Inline assembly is common in large applications. For instance, network applications use byte-order conversion routines (e.g., `htons`) implemented with specific machine instructions, while multimedia libraries use inline assembly to efficiently implement various algorithms.

While such code can be tedious to transform to C by hand, RevGen can do it automatically. RevGen scans the LLVM code, extracts inline assembly, identifies input/output parameters, wraps the assembly into separate LLVM functions, and uses `llvm-gcc` to turn these functions into binary code. Finally, RevGen translates the obtained binary code back to pure LLVM, which it uses as a drop-in replacement of the inline assembly.

Analyzing embedded software While x86 is a common architecture on desktop PCs and servers, there are many more architectures in the embedded world. For instance, smartphones use MIPS and ARM processors. RevGen can automatically convert instruction sets of these platforms to LLVM. This immediately allows the reuse of LLVM-based tools on embedded proprietary software. We shall see in the next section how RevGen's design enables the support of different architectures.

5.2 Challenges

Enabling static analysis of machine code poses two main challenges for static translators like RevGen: extracting binary code's semantics and inferring type information.

First, translators must extract the semantics of the machine instructions. For this, they decompose each complex instruction in a sequence of simpler operations (the intermediate representation). However, virtually all tools ignore the system instructions that manipulate the control state (e.g., switching execution modes, loading segment registers on x86, etc.). Therefore, such tools cannot analyze OS kernel code accurately. Finding bugs such as privilege escalation through virtual 8086 mode (affecting all Windows versions from NT 3.1 to Windows 7 [90]) is out of reach for them. RevGen addresses this challenge.

Second, translators must infer type information to enable accurate analysis. The LLVM IR is designed to retain most of the type information present in the source code. However, binaries only manipulate integers and memory addresses. The absence of type information degrades the quality of some analyses, in particular alias analysis. Analyses that rely on precise alias information have their rate of false positives and negatives increased.

The challenge for RevGen is to rebuild the type information and other LLVM constructs as if the resulting LLVM code was obtained by compiling source code. This places RevGen in between disassemblers and decompilers. While disassemblers stop after generating the IR, decompilers turn the IR into human-readable high-level code, after reconstructing type information, variables, control flow, etc. RevGen does not need to reconstruct high-level control flow.

5.3 Solution Overview

RevGen takes as input an x86 binary and outputs an equivalent LLVM module in three steps. The general architecture is shown in Figure 5.1. First, RevGen looks for all executable blocks of code and converts them to LLVM translation blocks (§5.5). Second, when there are no more translation blocks (TB) to cover, RevGen transforms them into basic blocks and rebuilds the control flow graph (CFG) of the original binary in LLVM format (§5.6). Third, RevGen resolves external function calls to build the final LLVM module. For dynamic analysis, a last step links the LLVM module with a run-time library that allows the execution of the LLVM module (§5.7).

5.4 LLVM Background

LLVM is a compiler framework that uses a compact RISC-like, SSA-based instruction set with an unlimited number of registers. LLVM has about 30 opcodes, only two of which can access memory

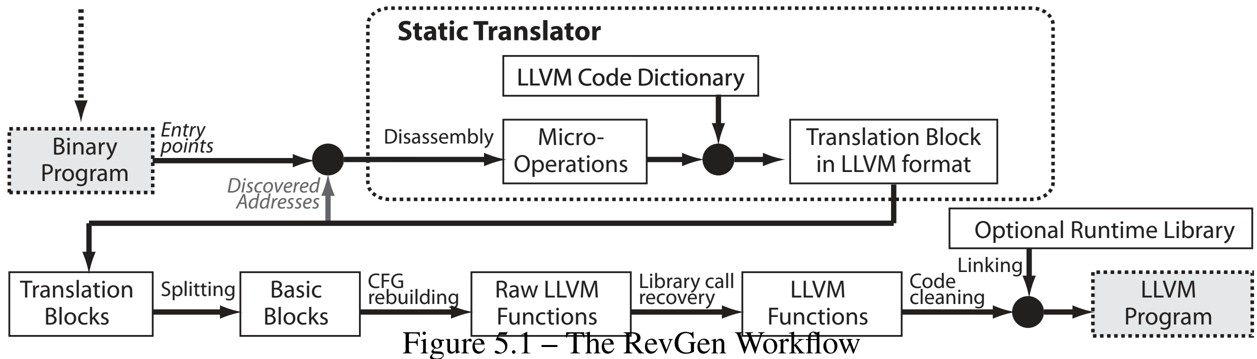

Figure 5.1 – The RevGen Workflow

(`load` and `store`), all other instructions operate on virtual registers. LLVM uses the static single assignment (SSA) code representation. In SSA, each register can be assigned only once. Hence, SSA also provides a `phi` instruction that assigns values to variables depending on the direction of the control flow. This instruction allows modifying the same variable in two different branches.

This makes LLVM programs amenable to complex analyses and transformations. LLVM code explicitly embeds the program's data flow and def-use graphs. This enables transformations like function inlining, constant propagation, or dead store removal, which are a key part of static and dynamic analysis tools.

A static translator must take into account LLVM specificities. This includes pointer arithmetic, different stack layouts, accesses to various code and data segments, indirect calls, and runtime support to be able to execute the generated LLVM programs. Finally, the translated code must be semantically-equivalent to the original binary.

5.5 Translating Blocks of Binary Code to LLVM

RevGen reuses S^{2}E's LLVM backend for translation, described in §3.1. Below, we briefly summarize the translation steps from the perspective of RevGen.

The static translator takes as input the binary file and a program counter and transforms all the machine instructions to LLVM until it encounters a terminator. A terminator is an instruction that modifies the control flow (e.g., branch, call, return). The translation has two steps: the input is first disassembled into micro-operations, which are then converted to LLVM instructions.

First, the translator converts machine instructions into an equivalent sequence of micro-operations. For example, the x86 instruction `inc [eax]` that increments the memory location pointed to by the `eax` register is split into a load to a temporary register, an increment of that register, and a memory store. The sequence of micro-operations forms a *translation block*.

Second, the translator maps each micro-operation to LLVM instructions, using a code dictionary. The dictionary associates each micro-operation with a sequence of LLVM instructions

that implement the operation. Most conversions are a one-to-one mapping between the micro-operations and the LLVM instructions (e.g., arithmetic, shift, load/store operations).

The translator also takes into account instructions that manipulate the system state. Current tools do not model such instructions to a sufficient precision level. For example, RevGen accurately translates to LLVM instructions like `fsave` or `mov cr0, eax`. The former saves the state of the floating point unit, while the latter sets the control register (e.g., to enable 32-bit protected mode, which changes the behavior of many instructions).

For this, the translator uses *emulation helpers*. An emulation helper is a piece of C code that emulates complex machine instructions that do not have equivalent micro-operations. RevGen compiles emulation helpers to LLVM and adds them to the code dictionary, transparently enabling the support of machine instructions that manipulate system state.

Third, the translator packages the sequence of LLVM instructions into an LLVM function that is *equivalent* to the original binary code. More precisely, given the same register and memory input, the translated code produces the same output as what the original binary does if executed on a real processor.

The translator stops when all translation blocks have been extracted. This happens when the translator cannot find new code to disassemble (e.g., by looking at not-yet explored jump and call target addresses).

5.6 Reconstructing the Control Flow Graph

RevGen primarily reuses existing tools (such as IDAPro [60]) in order to benefit of the many heuristics they developed over the years, which allows computing the CFG as accurately as possible. In addition to the binary, RevGen can take as input a set of basic block addresses, a set of control flow edges, and a set of function entry points generated by existing tools. RevGen uses this information in order to directly translate the basic blocks to LLVM, stitch them together, and group them into functions.

In case no advanced disassemblers are available, RevGen can use a few simple heuristics to reconstruct the CFG. These heuristics work on a variety of binaries produced by standard x86 compilers. However, they would fail on obfuscated or optimized binaries, e.g., if call instructions are replaced by push/jump instructions, in case of tail calls, etc.

RevGen implements the recursive traversal algorithm to disassemble the binary and rebuild the CFG [110]. This algorithm starts the disassembly at some known address (e.g., program entry point) and follows recursively all branches and function calls in order to discover new code. RevGen considers basic blocks that are targets of `call` instructions or that have no incoming edges to be function entry points.

After computing the CFG of each function, RevGen transforms it into an LLVM function. RevGen represents each basic block b of the original binary by an LLVM function f_b. RevGen first inserts an LLVM `call` instruction to the next basic block at the end of each f_b. Then, RevGen applies an LLVM function inlining pass to merge all the call targets into one large LLVM function.

5.7 Obtaining Analyzable LLVM Programs

The output of the CFG builder is a *raw* LLVM function that cannot be used by static or dynamic analyzers as is (e.g., it lacks explicit library calls and contains unresolved pointer arithmetic). We describe next how to transform the CFG builder output into analyzable code.

RevGen makes several assumptions about the original binary to synthesize analyzable LLVM code. RevGen requires the binary to provide a symbol table to identify library calls and a relocation table to identify all constants used as pointers. Relocation tables appear in shared libraries and are becoming common in program binaries because of ASLR [121] requirements. Moreover, the binary must not have self-modifying code. Finally, both the source and target architectures must have the same pointer size, in order to be able to run the translated code.

5.7.1 Enabling Static Analysis

First, RevGen identifies external function calls by scanning the import table of the program binary. An import table maps a list of function and library names to addresses. The OS loader patches the table with the actual function addresses so that indirect calls that reference the table can work properly.

Second, RevGen patches the raw LLVM functions with explicit external calls. Basic blocks originally encode external calls by an indirect jump to an address read from the import table. RevGen replaces such jumps by LLVM `call` instructions using the actual function names, allowing the LLVM linker to later resolve the call targets. This step is required because static analysis tools look for the use of specific functions. For example, memory checkers would track the calls to `malloc` and `free`.

Third, RevGen encodes the content of the program's code and data segments as LLVM arrays and embeds them in the LLVM program. This preserves the assumptions about the data layout in the original binary and accounts for programs that refer to segments with pointer arithmetic.

RevGen does not need to resolve indirect control flow (ICF). RevGen is not a static analysis tool, it only translates ICF from x86 to LLVM. Static resolution of ICF is left to the analysis tools. Such tools can resolve ICF to any precision they need and provide any soundness and completeness guarantees they wish.

5.7.2 Enabling Dynamic Analysis

Dynamic tools perform analysis at run-time, which requires executing the program. For this, in addition to applying the steps described in §5.7.1, RevGen links a specific runtime library that resolves indirect function calls, deals with memory layouts, and handles multi-threading. This runtime library does not affect run-time analysis tools, because they have full access to the library's LLVM bitcode. They see the runtime as another component of the analyzed program.

Resolving pointer arithmetic RevGen uses relocation tables to identify all pointers in the binary and adapt them to the LLVM memory model. Relocation tables list all code and data locations that the OS loader patches if it loads the binary at a different base address than what the compiler assumed. RevGen uses these tables to translate all hard-coded pointers to LLVM pointers, adapted to the memory layout seen by LLVM. In particular, RevGen remaps pointers that reference data segments to the corresponding LLVM arrays. Without relocation tables, RevGen performs this remapping at run-time, when all pointers are disambiguated.

Resolving indirect calls and branches The code generator embeds a table that maps basic blocks' native addresses to the corresponding LLVM basic blocks. It also stores which function the basic blocks belong to, as well as whether the basic block is the entry point of a function.

Whenever the translated code performs an indirect call or jump to a native address, the runtime looks for the corresponding LLVM basic block. If the block is not found or if there is a type mismatch (e.g., calling a block that is not a function entry point), the runtime aborts the program and notifies the user. This may occur when exploits attempt to violate control flow integrity [71].

Adapting stack pointers The translated code retains all the assumptions of the original binary about the stack layout. In particular, it assumes that local variables, return addresses, and parameters are located at precise memory addresses when doing stack pointer arithmetic.

The runtime library preserves the original stack layout by using a dual-stack architecture. There is one *native* stack used by the LLVM program and one *implicit* stack, whose pointer is passed as a parameter to each LLVM function, and which is manipulated by the LLVM functions. The runtime allocates the implicit stack and sets the implicit stack pointer before calling the main entry point of the program. It also copies the arguments to the native stack when calling library functions.

Supporting multi-threading Multi-threading consists in allocating an implicit stack for each thread. For this, the runtime library intercepts thread allocation routines and wraps the thread entry points into a special function that sets up the implicit stack. The stack is automatically freed when the thread routine finishes.

Self-modifying and obfuscated code RevGen does not support self-modifying code. This does not hurt RevGen because aside from malware, such code is practically restricted to JITed languages (e.g., C#, Java). In these cases, existing tools run on the bytecode of the respective languages, not on the final machine code itself. A side effect of this lack of support is that native code injected at run-time cannot be executed, increasing the safety of translated programs, similar to what other LLVM-based tools aim at achieving [42]. RevGen does not handle obfuscated code and would rely on existing deobfuscation techniques [67, 40] to obtain a CFG that can be turned into LLVM.

5.8 Results

In this section, we aim to give a preliminary answer to three questions: Does RevGen enable the reuse of existing LLVM-based analysis tools on x86 binaries? What completeness can RevGen achieve on typical binaries? How can one use RevGen in order to enable more effective path selectors in S^2E? We evaluate a prototype of RevGen that is based on S^2E's x86-to-LLVM translator (see §3.1). RevGen takes S^2E's dynamic translator and turns it into a static one.

To answer the first question, we convert an x86 micro-benchmark to LLVM using RevGen and run the result in CoreDet [10] . The micro-benchmark has several threads that access unprotected shared variables, whose value is printed at the end of each run. Without CoreDet, the printed output differs from run to run. With CoreDet, the output stays the same. This shows that RevGen enabled the reuse of CoreDet to make binary programs deterministic in the presence of race conditions.

Initial results suggest that RevGen's completeness is comparable to state-of-the-art disassemblers on standard non-obfuscated kernel-mode binaries. We disassembled the `pcntpci5.sys` Windows network device driver with RevGen and compared the results to IDAPro. IDAPro identified 78 functions, while RevGen found 77. RevGen failed to find 4 functions and misinterpreted 3 basic blocks as function starts, because of incomplete detection of jump tables. Of course, one can use IDAPro to produce a CFG and pass it to RevGen should better accuracy be needed (see §5.9).

Finally, we used RevGen to help path selectors in S^2E avoid polling loop in device drivers. A polling loop continuously reads a device register until it returns a certain value. Symbolic execution makes the register symbolic, causing a fork on each iteration, yielding path explosion. We are therefore interested in terminating all execution paths that follow the back-edge of the loop.

Avoiding polling loops can be done by combining static and dynamic analysis. LLVM has an analysis pass that identifies loops in CFGs. By running it on the LLVM bitcode produced by RevGen, it is possible to identify all the loops in the original driver binary. At runtime, the path selector checks whether forks occur in a loop header and whether the symbolic expression was created inside the loop body. If so, the selector marks the loop as a polling loop, terminating all states that execute its back-edge.

Combining static with dynamic analysis improves accuracy. Polling loops may follow many different patterns: they may or may not have a timeout counter, call delay functions, etc. Static analysis allows easily recognizing such patterns, whereas dynamic analysis would require exploring all the branches of the loop body first in order to have a complete view of the loop. Conversely, dynamic analysis allows easily checking whether the loop header forked because of a symbolic value that comes from hardware, which is hard to determine statically. In summary, by carefully restricting the heuristic to specific loop patterns, our polling loop detection achieved zero false positives on all 45 Windows network device drivers that we tested.

5.9 Discussion

In this section, we discuss three aspects that we believe will enable RevGen to become a major enabler for widespread static analysis of binary programs.

RevGen can effortlessly leverage existing disassemblers, should better accuracy be required. RevGen's translator only requires a list of program counters and an accurate list of function entry points in order to convert the binary to LLVM. Both can be directly obtained from disassemblers like IDAPro or state-of-the-art static analyzers such as Jakstab.

Extending RevGen to support other architectures than x86 is simple and requires limited efforts. The LLVM backend that translates micro-operations to LLVM need not be modified. In case no existing disassemblers are available, the only need is to modify the frontend (e.g., the ARM or MIPS frontend) with annotations specifying the types of basic blocks (e.g., branch, call, return, etc.) to allow the CFG builder to merge the basic blocks and reconstruct the functions.

We argue that RevGen enables analysis tools to check binary programs as well as their interaction with the processor. Analysis tools typically check programs that interact with libraries. In the context of RevGen, the program is the machine code translated to LLVM and the library is the collection of emulation helpers in LLVM format. For example, checking that the invocation of a software interrupt does not cause a general protection fault is reduced to verifying that the library does not invoke the corresponding program's entry point.

This can potentially open up all sorts of analyses on low level system code. We envision RevGen to enable analysis tools to answer questions like: Can the user-mode code issue a system call in such a way that would cause arbitrary code execution? Are there any bugs in the emulation helpers (and thus in QEMU) that would cause the application to malfunction? Since RevGen produces plain LLVM bitcode, we expect existing tools to answer such questions out of the box.

5.10 Summary

This chapter presented RevGen, a tool that automatically converts existing binary programs to the LLVM intermediate representation. RevGen can potentially enable a large number of static and dynamic analysis frameworks, as well as run-time instrumentation tools to work on legacy software. Preliminary results show that RevGen can successfully translate large Windows drivers, run existing dynamic analysis tools on binary programs, and let developers write more efficient path selectors for S^2E.

Chapter 6

Evaluation: Real Tools Built with S^2E

S^2E's main goal is to enable rapid prototyping of useful, deep system analysis tools. To evaluate this, we show in this chapter how we used S^2E to build a tool for reverse engineering of binary device drivers (§6.1) and performing multi-path performance analysis (§6.2). We also show the impact of S^2E in general by presenting several tools built by the research community (§6.3). Finally, we show that tools built with S^2E can achieve reasonable performance (§6.4) and explain the measured trade-offs involved in choosing different execution consistency models on both kernel-mode and user-mode binaries (§6.5).

Table 6.1 summarizes the productivity advantage we experienced by using S^2E to build our tools, compared to writing them from scratch. For these use cases, S^2E engendered two orders of magnitude improvement in both development time and resulting code volume. This justifies our efforts to create general abstractions for multi-path in-vivo analyses, and to centralize them into one platform.

Use Case	**Development Time** [person-hours]		**Tool Complexity** [lines of code]	
	from scratch	*with S^2E*	*from scratch*	*with S^2E*
Testing of proprietary device drivers	2,400	38	47,000	720
Reverse engineering of closed-source drivers	3,000	40	57,000	580
Multi-path in-vivo performance profiling	n/a	20	n/a	767

Table 6.1 – Comparative productivity when building analysis tools from scratch (i.e., without S^2E) vs. using S^2E. Reported LOC include only new code written or modified; any code that was reused from QEMU, KLEE, or other sources is not included. For reverse engineering, 10 KLOC of offline analysis code is reused in the new version. For performance profiling, we do not know of any equivalent non-S^2E tool, hence the lack of comparison.

6.1 Reverse Engineering of Closed-Source Drivers

The ability to use a hardware device with an operating system requires that a corresponding device driver be available, i.e., a program that knows how to mediate the communication between the OS kernel and that specific device. The driver may be supplied by either the hardware vendor or the developer of the operating system.

Hardware vendors typically provide drivers for the one or two most popular OSes. It appears that supporting many other platforms is not profitable, because the high cost of development and technical support can be amortized only over comparatively fewer customers. As a result, drivers are rarely available for every OS/device combination. This issue is common to various device classes, including network drivers. Alas, for an operating system to be viable and widely adopted, it must support a wide range of hardware.

Even when drivers are available, they are often closed-source and proprietary. Despite this making them less trustworthy, proprietary drivers are still permitted to run at the highest level of privilege in an operating system. Not surprisingly, buggy drivers are a leading cause of crashes [119]. They can also be a security threat, as was the case of a driver shipped with all versions of Windows XP that was found to contain a zero-day privilege escalation vulnerability [85].

Writing a driver for an OS that is not supported by the device vendor is challenging, because the device specification is often not available. While the interface exposed by an OS to the driver is well known, the specification of the interface between the hardware and the driver is often not public. The classic approach is to manually reverse engineer the original driver, but that involves a lot of work. When the device is too complex to be reverse engineered, developers resort to emulating the source OS using wrappers (e.g., NDISwrapper [95] allows running Windows NIC drivers on Linux). However, this adds performance overhead, can only use drivers from one source OS, and requires changing the wrapper each time the source OS driver interface changes.

Even when the hardware specification is available, writing the driver still requires substantial effort. That is why, for example, it took many years for Linux to support widely used wireless and wired NIC devices. Vendor-provided specifications can miss hardware quirks, rendering drivers based on these specifications incomplete (e.g., the RTL8139 NIC driver on Linux is replete with workarounds for such quirks).

Our proposed approach overcomes unavailability of specifications and costly development with a combination of automated reverse engineering and driver code generation. We observe that a device specification is not truly necessary as long as there exists one driver for one platform: that one driver is a (somewhat obscure) encoding of the corresponding device protocol. Even if they are not a perfect representation of the protocol, proprietary drivers incorporate handling of hardware quirks that may not be documented in official specification sheets. We also observe that

writing drivers involves a large amount of boilerplate code that can be easily provided by the OS developer. In fact, an entire category of drivers can use the same boilerplate; driver writers plug into this boilerplate the code specific to their hardware device.

We implemented our approach in RevNIC, a tool for automating the reverse engineering of network drivers. This tool can be used by hardware vendors to cheaply support their devices on multiple platforms, by OS developers to offer broader device support in their OS, or by users who are skeptical of the quality and security of vendor-provided closed-source proprietary drivers.

When it originally appeared in [33], RevNIC made three main contributions. First, it introduced a technique for tracing the driver/hardware interaction and turning it into a driver state machine. Second, it demonstrated the use of binary symbolic execution to achieve high-coverage reverse engineering of drivers. Third, it showed how symbolic hardware can be used to reverse engineer drivers without access to the actual physical device.

RevNIC was the precursor of S^2E. RevNIC was originally a standalone, monolithic tool. Then, we factored out its path exploration engine, which turned out to be useful for many use cases. We describe these later in this chapter (§6.3). We then moved RevNIC-specific functionality into standalone plugins. Finally, RevNIC's offline trace generation tool gave birth to RevGen.

In this section, we show how to rebuild RevNIC using S^2E. We call REV^+ the re-implementation of RevNIC using S^2E. After providing an overview of RevNIC (§6.1.1), we describe how RevNIC "wiretaps" drivers (§6.1.2) and synthesizes new driver code (§6.1.3), we evaluate RevNIC (§6.1.4), discuss limitations (§6.1.5) and how RevNIC influenced S^2E (§6.1.6), survey related work (§6.1.7), and summarize (§6.1.8).

6.1.1 System Overview

To reverse engineer a driver, RevNIC observes the driver-hardware interaction, i.e., the manifestation of the device-specific protocol, encoded in the driver's binary. RevNIC synthesizes an executable representation of this protocol, which the developer can then use to produce a driver for the same or a different OS.

RevNIC employs a mix of concrete and symbolic execution to exercise the driver and to wiretap hardware I/O operations, executed instructions, and memory accesses. When rebuilding RevNIC, we leveraged S^2E's symbolic execution engine and tracing plugins (wiretap), as shown in Figure 6.1. The output of the wiretap is fed into a code synthesizer, which analyzes the trace information and generates snippets of C code that, taken together, implement the functionality of the device driver. The developer then pastes the code snippets into a driver template to assemble a new driver that behaves like the original one in terms of hardware I/O.

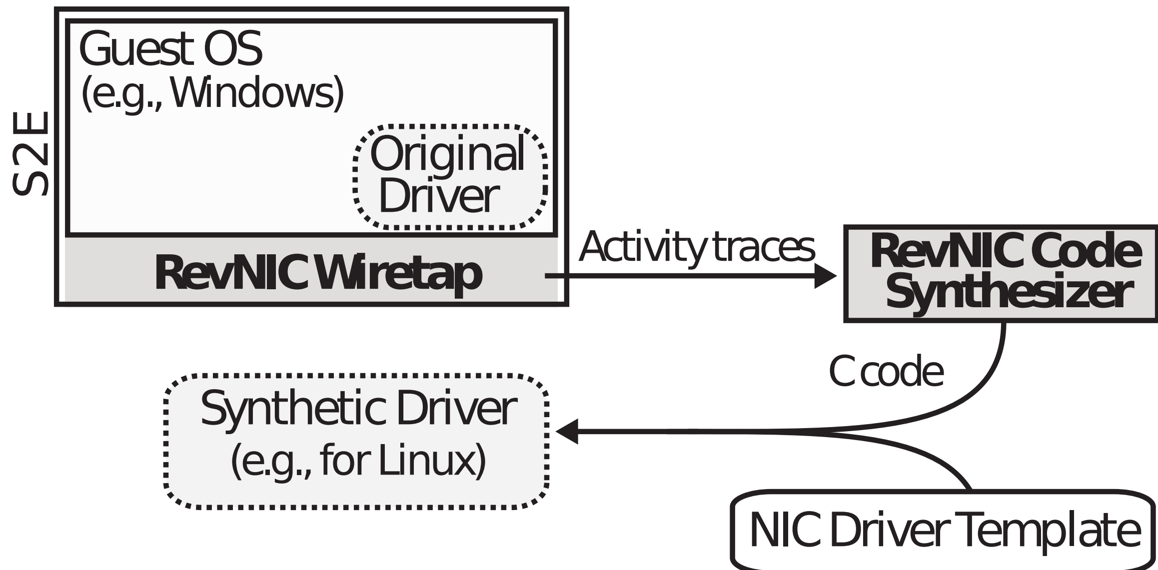

Figure 6.1 – High-level architecture of RevNIC.

Exercising the driver. It is difficult to exercise every relevant code path in the driver using just regular workloads. Many paths correspond to boundary conditions and error states that are hard to induce. For example, a NIC driver could take different paths depending on the packet type transmitted by the network stack (ARP, VLAN, etc.) or depending on how the hardware responds.

In order to induce the device driver to perform its operations, RevNIC guides the execution with a mix of concrete and symbolic workload. The concrete workload initiates driver execution by triggering the invocation of the driver's entry points. RevNIC selectively converts the parameters of kernel-to-driver calls into symbolic values (i.e., values that are not constrained yet to be any specific, concrete value) and also treats the responses from the hardware side as symbolic. This drives the execution down many feasible paths through the driver, as well as exercises all code that depends on hardware input/returns.

By using symbolic values, RevNIC is completely independent of the physical hardware device. Unconstrained symbolic values provide the driver a representation of all the responses from the hardware that the driver thinks could ever be received.

Recording driver activity. REV$^+$ uses S^{2}E's tracing plugins to record the driver's hardware I/O, along with the driver's memory accesses and an intermediate representation of the instructions it executes (§6.1.2). Such detailed tracing imposes performance overheads, but the overheads are irrelevant to reverse engineering.

Synthesizing driver code. RevNIC infers from the collected activity traces the state machine of the binary driver and produces C code that implements this state machine. RevNIC automatically merges multiple traces to reconstruct the control flow graph (CFG) of the original driver.

Since the generated CFG is equivalent to that of the original driver, covering most of the basic blocks is sufficient to reverse engineer the driver—complete path coverage is not necessary.

RevNIC uses virtualization and symbolic execution instead of mere decompilation for four main reasons. First, static decompilers face undecidable problems (e.g., disambiguating code from data) and can produce inaccurate results [110]. Second, while some decompilers can record dynamic execution traces [60] to improve accuracy, RevNIC explores multiple paths in parallel and covers unexplored code faster using symbolic execution. Third, since the VM catches all hardware accesses, RevNIC can distinguish accesses to a memory-mapped device from regular memory accesses, which is notoriously difficult to do statically on architectures like x86. Identifying such instructions is crucial to preserving memory access ordering in the generated code (e.g., write barriers). Finally, RevNIC must recognize DMA-allocated regions assigned by the OS to the driver (by recording the address values returned by the OS API); doing this in a decompiler requires complex inter-procedural data flow analysis.

Writing a driver template. The template contains all OS-specific boilerplate for interfacing with the kernel (e.g., NDIS API on Windows, network API on Linux). Templates can be arranged in a class hierarchy with an abstract template implementing the basic boilerplate, and derived templates implementing additional functionalities. For example, a base template may target a generic PCI-based, wired NIC, while a derived template further adds DMA capabilities. This modularity allows accommodating all common types of network devices. Depending on the OS, templates can be derived from examples in driver development kits (e.g., the Windows Driver Kit [86]) or automatically generated, with tools like WinDriver [65].

Besides mandatory boilerplate, a template also contains placeholders for the actual hardware interaction. Since devices in a given class (e.g., NICs) tend to operate in the same manner (e.g., initializing, sending, receiving, computing checksums using well-defined standards), the functional differences between drivers are at the level of code implementing this hardware I/O.

Producing the synthetic driver. The developer pastes synthesized code snippets into the template's placeholders in order to specialize the template for the device of interest. The result is then compiled into a driver. In order to paste code snippets correctly, the developer has to know how to write drivers both for the source and for the target OS.

We use RevNIC to port drivers from/to various OSes: Linux, Windows, the µC/OS-II real-time embedded OS for FPGA systems, and our own experimental KitOS. The following sections detail how RevNIC traces driver activity and processes activity traces to produce the synthetic driver.

6.1.2 Tracing and Exercising the Original Driver

We now describe what information RevNIC records along each execution path in order to generate a working driver for the target OS, how it uses symbolic execution and symbolic hardware to exercise the driver, and how it ensures that symbolic execution achieves high coverage.

6.1.2.1 Wiretapping the Driver

Along each execution path, RevNIC records several pieces of information that the offline processing tool will use to synthesize the new driver.

First, the wiretap saves the instructions executed by the driver in an intermediate representation. This serves as a basis for C code generation during the synthesis phase (§6.1.3). Second, the wiretap records whether the instructions access device-mapped memory or regular memory, along with the value of the corresponding pointer and the transferred data. This simplifies the data flow analysis during reverse engineering (§6.1.3.1), by disambiguating aliased pointers. Third, the wiretap records the type of executed basic blocks (conditional vs. direct/indirect jumps vs. function calls) and the contents of the processor's registers at the entry and exit of each basic block. This helps reconstruct the control flow during synthesis.

REV^+ leverages two additional plugins that come with S^2E in order to re-implement the wiretapping capabilities of RevNIC:

- The *InstructionTracer* plugin allows recording the instructions executed by the driver as well as the content of the CPU registers.
- The *MemoryTracer* plugin records all memory and MMIO accesses performed by the driver.

6.1.2.2 Mechanic of Exercising the Driver

In order to expose the driver wiretap to as many behaviors as possible, RevNIC exercises the driver using symbolic execution. Whereas RevNIC used an ad-hoc implementation of symbolic execution, REV^+ uses S^2E and combines several plugins in order to re-implement RevNIC's high-coverage path exploration:

- The *CodeSelector* plugin restricts multi-path exploration to the target driver. This prevents path explosion outside of the driver being reverse engineered.
- The *SymbolicHardware* plugin enables exploring paths that depend on hardware input. Consider, for example, the NIC interrupt handler: since a read of the status register returns a symbolic value, all conditional branches that depend on that value are automatically explored, without requiring a cleverly crafted workload that would induce a real NIC into producing all possible return values (which may be impossible to do solely using a workload).

As I/O to and from the hardware is symbolic thanks to the *SymbolicHardware* plugin, the actual device is never needed. This allows developers using RevNIC to reverse engineer drivers for rare or expensive devices they do not (yet) have access to.

6.1.2.3 Achieving High Coverage

RevNIC attempts to maximize the basic block coverage of the driver, in order to maximally capture its behavior. RevNIC first determines the entry points of the driver and makes the OS invoke them, in order to initiate their symbolic execution. Then, RevNIC guides symbolic execution using heuristics whose goal is to maximize coverage while reducing the time spent exercising the driver.

RevNIC's requirements. To exercise the driver, RevNIC must know the semantics of the OS interface. This requires that the OS driver interface and all API functions used by the driver be documented. The documentation must include the name of the API functions, the parameter descriptions, along with information about data structures (type and layout) used by these functions. RevNIC internally encodes this information in order to correctly determine and exercise the entry points provided by the driver.

Discovering driver entry points. RevNIC monitors OS-specific entry point registration calls. In the case of Windows, drivers usually export one function, which is called by the kernel to load the driver. This function registers the driver's entry points with the OS by passing a data structure to a specific OS function (`NdisMRegisterMiniport`). At run-time, the driver can register other entry points, like timers (via `NdisInitializeTimer`). RevNIC monitors calls to such OS APIs to record the contents of the data structures and function pointers. Since these structures contain actual function pointers and have documented member variables, RevNIC knows which entry points need to be exercised and the developer is aware of the functionality each entry point is responsible for. RevNIC includes a default set of NDIS function descriptions and allows users to specify what additional functions to monitor.

RevNIC invokes each entry point of the driver via a user-mode script or program that runs in the guest OS. The script first loads the driver so as to exercise its initialization routine, then invokes various standard IOCTLs, performs a send, exercises the reception, and ends with a driver unload. Interrupt handlers are triggered by the VM, as we shall see shortly. Once an entry point is called, its code is executed symbolically until no more new code blocks are discovered within some predefined amount of time.

Initiating symbolic execution. RevNIC intercepts entry point invocations, then fills with symbolic data the user buffers and the integer parameters passed in, while keeping the other parameters, like pointers, concrete. For example, to exercise sending, RevNIC runs a program that sends packets of various sizes. RevNIC catches the invocation of the send entry point, then replaces the concrete data within the packet and the packet length with symbolic values. This exercises the

paths corresponding to the various packet types and sizes.

Injecting symbolic values from the OS side, however, may cause execution to reach an impossible error state (e.g., wrongly crash the driver) if the symbolic values subsume concrete values the kernel would never pass to the driver. When any error state is reached, RevNIC terminates the execution path and resumes a different one. Reaching these infeasible error states does not perturb the reverse engineering process, since RevNIC merely aims to touch as many basic blocks as possible and cause them to manifest in the traces. RevNIC's goal is not to expose the driver to a realistically functioning device or OS, but rather to reverse engineer the state machine implemented by the driver.

In S^2E's terminology, RevNIC uses overapproximate consistency. The goal of the tracer is to see each basic block execute, in order to extract its logic—full path consistency is not necessary. The offline trace analyzer only needs fragments of paths in order to reconstruct the original CFG. By using overapproximate consistency, RevNIC sacrifices strict consistency in exchange for obtaining coverage fast.

Guiding driver exploration with heuristics. Symbolic execution generates a large number of paths, with execution having progressed to various depths down each path. RevNIC executes one path at a time, but frequently switches between them. The choice of which path to execute next is driven by a strategy that, in RevNIC, relies on several heuristics. RevNIC's heuristics aim to choose the paths most likely to lead to previously unexplored code. Discarding early on paths that are unlikely to discover any new code helps cope with the large number of paths. Note that RevNIC allows these heuristics to be modularly replaced, when and if better ones are discovered.

The first heuristic explicitly selects paths most likely to discover new code. Every time RevNIC completes executing one basic block of the driver, it decides whether to continue that same execution path (by executing the next basic block) or to switch to a basic block in another execution path. We refer to a <path,block> tuple as "state," as it directly determines program state. RevNIC associates with each basic block a counter that is incremented after that block is executed. The next state to execute is the one corresponding to the basic block with the lowest count. A good side effect of this strategy is that it does not get "stuck" in loops, since it decreases the priority of states that merely re-execute a previously explored loop. We found this heuristic to speed up exploration, compared to depth-first search (which can get stuck in polling loops) or breadth-first search (which can take a long time to complete a complex entry point).

In contrast to RevNIC, REV^+ uses a simple random path heuristic. RevNIC's heuristic was designed to discover new basic blocks as quickly as possible while forking the least number states, because of various limitations (§6.1.6). REV^+ can explore a much larger number of states much quicker and has multi-core support. In this case, we found that the random path strategy was giving better results (§6.1.4.5).

A separate heuristic selects paths to discard in polling loops and large entry points. Symbolic execution forks virtually identical states at a high rate in case of polling loops. To avoid memory exhaustion, RevNIC keeps the paths that step out of the polling loops and kills those that go on to the next iteration. A large number of states can also get RevNIC stuck in one entry point, preventing subsequent entry points from ever being reached. To cope with this, whenever an entry point completes with a successful error code a given number of times, RevNIC discards all paths except one successful one chosen at random. The execution then proceeds to the next entry point, controlled by the script.

REV$^+$ combines the *EdgeDetector* and *EdgeKiller* plugins in order to implement this heuristic. *EdgeDetector* triggers an event when it detects that execution goes through a specified edge in the CFG. *EdgeKiller* listens for events generated by *EdgeDetector* and kills the state whenever the event corresponds to the back-edge of a polling loop.

A third heuristic guides RevNIC in injecting interrupts at specific points in the execution, to exercise interrupt handlers. For NIC drivers, triggering interrupts after returning from a driver entry point (e.g., send) works well, since that is the moment when the device either triggers a completion interrupt, a receive interrupt, or some other type of interrupt (error, buffer overflow, etc.). This strategy results in virtually complete coverage of the interrupt handlers. In general, however, the code paths depend on previous execution histories. For example, the initialization phase may expect an interrupt after writing to some register. If the interrupt does not occur, the initialization fails, preventing the rest of the driver from running. The heuristic must therefore be able to detect such cases, e.g., by analyzing writes to shared variables. We are exploring ways to use data dependency information to optimally and automatically choose the moments to inject such asynchronous events.

A final heuristic helps RevNIC skip over unnecessary function calls. First, device drivers often call functions irrelevant to the hardware protocol, such as writing debug/log information (via calls like `NdisWriteErrorLogEntry`). Such (OS-specific) functions can be indicated in RevNIC's configuration file, and RevNIC will skip them. Second, some hardware-related functions can be replaced with annotations, to speed up execution. For example, a common pattern is to construct an integer by reading it bit by bit from a register. This is commonly done in a loop that has 16, 32, or 64 iterations depending on the size of the integer. Each iteration may branch on the value of the bit, causing an exponential path explosion. RevNIC can report on the first run the functions that fork the most, in order to let the developer specify which ones should be replaced with annotations on subsequent runs. Such annotations have a few lines of code and are easy to write: they just need to set the program counter appropriately to skip the call, and return a symbolic value. This may yield orders-of-magnitude reduction of the state space, e.g., in case the annotated function is a complex register read, or computes a checksum.

REV$^+$ leverages the *Annotations* plugin in order to implement many of the heuristics. *Annotations* can be combined with *FunctionMonitor* in order to detect functions called by the driver and perform function-specific actions, such as manipulating the virtual machine state in order to skip them. It can also be used to inject interrupts, by calling from the annotation the corresponding API exposed by the *SymbolicHardware* plugin. Discovering driver's entry points and initiating symbolic execution can also be done via the *Annotations* plugin.

6.1.3 Synthesizing A New Equivalent Driver

RevNIC exercises the driver and outputs a trace consisting of translated LLVM blocks, along with their sequencing and all memory and I/O information. Now we describe how this information is processed and used to synthesize C code for a new driver that behaves like the original.

6.1.3.1 From Trace to a C-encoded State Machine

Generating C code from the traces consists of rebuilding the control flow graph of the driver's functions and converting the corresponding basic blocks from LLVM to C.

Rebuilding the CFG. The driver wiretap produces raw execution traces that contain explicit paths through the driver's execution tree (§6.1.2.2). Each such path, from the root to a leaf, corresponds to an execution of the driver, exercising a different subset of the code. The trace does not contain OS code, because RevNIC stops recording when execution leaves the driver. A traced path ends when it is terminated by RevNIC (e.g., due to being stuck in a polling loop), when driver initialization fails, or when the unload routine of the driver completes (and thus there is nothing more left to execute). The traces also contain interspersed snippets of asynchronous execution, like interrupt and timer handlers.

RevNIC merges the execution paths from traces in order to rebuild the state machine (i.e., control flow graph) of the original driver. A CFG contains all the paths that a driver could traverse during its execution. To build a CFG equivalent to that of the original driver, it is sufficient to execute at least once each basic block of the driver. Building is done in two steps: First, RevNIC identifies function boundaries by looking for call-return instruction pairs. Second, the translation blocks between call-return pairs are chained together to reproduce the original CFG of the function. RevNIC splits translation blocks into basic blocks in the process.

Execution paths can contain manifestations of asynchronous events that disrupt the normal execution flow. RevNIC detects these events by checking for register value changes between two consecutively executed translation blocks. The register values are saved in the trace before and after the execution of each block. RevNIC builds the CFG of each such event just like for normal functions.

From CFG to C code. The output of the CFG builder is a set of functions in LLVM format. The last phase turns these functions into C code, reconstructs the driver's state, and determines the function parameters, the return values, and the local variables. Listing 6.1 shows a sample of generated code. The control flow is encoded using direct jumps (`goto`) and all function calls are preserved.

```
void function_12606(uint32_t GlobalState)
{
  uint32_t Vars4[4]; //Local variables
  Vars4[3] = 0x0;
  //Driver's state is accessed using pointer arithmetic
  Vars4[2] = *(uint32_t*)(GlobalState + 0x10);
  write_port32(Vars4[2] + 0x84, Vars4[3]);
  //Remainder omitted...
}
```

Listing 6.1 – Generated code sample (annotated).

RevNIC preserves the local and global state layout of the original driver (Listing 6.1). Drivers usually keep global state on the heap, a pointer to which is passed to the driver upon each entry point invocation. To access their global state, they use offsets from that pointer. Binary drivers access local variables similarly, by adding an offset to the stack frame pointer. The synthesized code preserves this mechanism by keeping the pointer arithmetic of the original driver.

RevNIC determines the number of function parameters and return values using standard def-use analysis [37] on the collected memory traces. Since the traces contain the actual memory access locations and data, it is possible to trace back the definition of the parameters and the use of the possible return values. To determine whether a function f has a return value, RevNIC checks whether there exists an execution path where the register storing the return value[1] is used without being redefined after f returns. The number of parameters is determined by looking for memory accesses whose addresses are computed by adding an offset to the stack frame pointer, resulting in an access to the stack frame of the parent function.

The generated code may be incomplete if the driver is not fully covered (i.e., the code has an incomplete CFG). Incompleteness manifests in the generated source by branches to unexercised code. RevNIC flags such branches to warn the developer. Missing basic blocks happen for driver functions containing API calls whose error recovery code is not usually exercised. It does not affect the synthesized driver, since error recovery code is part of the template (§6.1.3.2). However, in the case when code for hardware I/O is missing, the developer can request QEMU's DBT to generate the missing translation blocks by forcing the program counter to take the address of the unexplored block. Since RevNIC does not execute such blocks, they do not appear in the execution trace: the developer must insert the code for these blocks manually, which slows down the reverse engineering process.

[1]This is specific to the Windows `stdcall` calling convention; other conventions can be implemented as well.

6.1.3.2 From State Machine to Complete Drivers

Producing a new driver consists of pasting the synthesized C code into a driver template. A template is written once and can be reused as long as the OS/driver interface does not change. The template contains all the boilerplate to communicate with the OS (e.g., memory allocation, timer management, and error recovery). Depending on the OS, more or less boilerplate code may be required (e.g., a driver for an embedded OS typically has less boilerplate than an NDIS driver). The boilerplate can also vary depending on the OS version. For example, a template could use the newer NAPI network model on Linux, or the older API. This only affects the organization of the template. Besides the boilerplate, the template also contains placeholders where the actual hardware I/O code is to be pasted. Listing 6.2 shows a fragment of the `init` function from the Linux NIC template, which we use as a running example. RevNIC includes a NIC template for each supported target OS.

Drivers typically have four types of functions. The first type corresponds to functions that only call hardware I/O routines (e.g., `write_port32` in Listing 6.1) or other hardware functions: register read-modify-write, disable interrupts on a NIC, read the MAC address, etc. The second type consists of OS-dependent functions that assemble hardware-dependent routines to perform a high-level functionality. For example, a `send` would call a function that sets the ring buffer index, then call the OS to get the packet descriptor, after which it would invoke another hardware-specific routine, passing a pointer and a size to transmit the packet. The third type is similar to the second, except that it mixes hardware accesses with calls to the OS. This happens, e.g., when the driver inlines hardware functions. Finally, the fourth type includes functions that implement OS-independent algorithms, such as checksum computation.

Given a NIC driver template, a developer inserts the calls to OS-independent functions generated by RevNIC into the template. The amount of effort required to build a working driver is usually minimal: look at the context in which the functions were called in the original driver (in an interrupt, a send packet routine, a timer entry point, etc.) and paste them in the corresponding places in the template. The developer also has to adapt various OS-specific data structures to the target OS, e.g., to convert the Windows `NDIS_PACKET` structure to the equivalent `sk _buff` Linux structure. This is the most time-consuming part of reverse engineering, but could be simplified by annotating the generated code with type information (e.g., based on the source OS's header files). The developer also needs to match OS-specific API calls to those of the target OS. In Windows, such APIs and structures are public and documented.

Filling in a template is straightforward when only functions of types 1, 2, and 4 occur in the traces. However, when hardware I/O is mixed with OS-specific code (type 3), more effort is required. Without RevNIC, the developer would have to look at the disassembly or the decompiled code to understand what the driver does. This requires distinguishing regular from device-mapped

memory accesses, understanding complex control flow, and grasping the interaction with the OS. Instead, RevNIC provides the developer with execution traces annotated with hardware I/O, which can be used to retrace the execution instruction by instruction. This makes it easier to understand the interaction between the driver's components and eases integration into the template.

NIC driver templates follow common patterns, which we found to be similar across different OSes. For example, the `init` entry point implemented by the Linux, Windows and µC/OS-II templates first allocates device resources, calls a hardware-specific function that checks for the presence of the hardware, registers the device, and brings it to its initial state. Likewise, an interrupt handler in these three OSes first calls a hardware routine to check that the device has indeed triggered the interrupt, before handling it.

Common template patterns facilitate driver synthesis for multiple platforms. For example, each template contains one lock to serialize the entry points (this ensures correct operation but may affect performance). The developer strips all OS-specific locks that might be present in the original driver, because they are not needed anymore. Then, the developer pastes that code in the same places across all templates, without worrying about target OS-specific synchronization.

Ideally, the merging of the synthesized driver code with the template would be fully automated. However, the process of translating from one OS to another requires refactoring the original driver's OS-specific functions and translating API calls to fit them in the generic template for the target OS. While human developers can guess quite easily how to translate these, an automated translator would need to correctly reconstruct the driver binary's missing type information to understand how the driver manipulates the data structures (e.g., lists) in order to adapt them to the target OS.

```
int revnic_pci_init_one (struct pci_dev *pdev, const struct pci_device_id *ent)
{
  /* Variable declarations omitted */

  // The template first allocates PCI device resources
  if (pci_enable_device (pdev)) {
    // OS-specific error handling provided by template
  }
  ioaddr = pci_resource_start (pdev, 0);
  irq = pdev->irq;
  if (request_region(ioaddr, ADDR_RANGE, DRV_NAME)) {
    // OS-specific error handling provided by template
  }

  // Then the template allocates persistent state. A pointer to this state
  // is passed to each reverse engineered entry point.
  dev = alloc_netdev(/*..*/, /*..*/, ethdev_setup);
  if (dev) {
    // OS-specific error handling provided by template
  }
  memset(netdev_priv(dev), 0, /*..*/);
  SET_NETDEV_DEV(dev, &pdev->dev);
  revnic = REVNICDEVICE(dev);

  // The synthesized functions may expect specific state (e.g., the I/O address)
  // to be initialized. Here, the I/O address is stored at offset IOADDR_OFFSET.
  revnic->Private.u4[IOADDR_OFFSET] = ioaddr;

  //*****************************************************************************
  // Developers paste calls to RevNIC-synthesized hardware-related functions here.

  // A driver may want to check the hardware first...
  if (ne2k_check_device_presence(&revnic->Private) < 0) {
     // Error recovery provided by the template (e.g., unload the driver)
  }

  // ...before initializing it
  if (ne2k_init(&revnic->Private) < 0) {
    //Device-specific recovery synthesized by RevNIC...
    ne2k_shutdown(&revnic->Private);
    // ...followed by template-provided recovery code (e.g., unload the driver)
  }
  //*****************************************************************************
  // More OS-specific initialization goes here.
  // Initialize IRQ, I/O addresses, entry points, etc.
  // ...

  // Template adapts the driver's data structures to the target OS.
  // Here, it copies the MAC address from driver's memory to the Linux
  // data structure. Adaptation is done by the driver developer.
  for (i=0; i<MAC_ADDR_LEN; i++) {
    dev->dev_addr[i] = revnic->Private.u1[0x14b + i];
  }
  register_netdev(dev);
  return 0;
}
```

Listing 6.2 – Example `init()` routine of the Linux NIC template (edited for brevity).

6.1.4 Evaluation

In this section, we address several important questions: Do RevNIC-generated drivers have functionality (§6.1.4.1) and performance (§6.1.4.2) equivalent to the original drivers? How much effort does the reverse engineering process entail (§6.1.4.3)? How far can RevNIC scale (§6.1.4.4)? To answer these questions, we use RevNIC to port four closed-source proprietary Windows NIC drivers to three other operating systems as well as back to Windows, producing a total of 11 driver binaries. At no time in this process did we have access to the drivers' source code.

Experimental Setup

We first present the evaluated drivers, describe the three target operating systems, and give details on the hardware used for measurements.

Evaluated Drivers. We used RevNIC to reverse engineer the Windows drivers of four widely used NICs (Table 6.2). Three of the four ship as part of Windows, attesting to their popularity. The drivers range in size from 18KB to 35KB, which is typical for NIC drivers in general (e.g., 80% of network drivers in Linux 2.6.26 are smaller than 35KB). The number of functions the drivers implement ranges from 48 to 78 and the number of used OS API functions ranges from 37 to 51. The Windows driver files for AMD PCNet, RTL8139, SMSC 91C111, and RTL8029 are `pcntpci5.sys`, `rtl8139.sys`, `lan9000.sys`, and `rtl8029.sys`, respectively. Linux has equivalent drivers for the same chipsets: `pcnet32.c` (2300 LOC), `8139too.c` (1900 LOC), `smc91x.c` (1300 LOC), and `ne2k-pci.c` / `8390.c` (1200 LOC).

Reverse Engineered Windows Driver	RevNIC Ported from Windows to ...	Driver Size	Code Segment Size	Imported Windows Functions	Functions Implemented by the Original Driver
AMD PCNet	Windows, Linux, KitOS	35KB	28 KB	51	78
Realtek RTL8139	Windows, Linux, KitOS	20KB	18 KB	43	91
SMSC 91C111	µC/OS-II, KitOS	9KB	10 KB	28	40
Realtek RTL8029 (NE2000)	Windows, Linux, KitOS	8KB	14 KB	37	48

Table 6.2 – Characteristics of the proprietary, closed-source Windows network drivers used to evaluate RevNIC.

Target Platforms. We use RevNIC to port the PCNet, RTL8139, and RTL8029 drivers to Linux 2.6.26, and 91C111 to µC/OS-II. This shows RevNIC's ability to port drivers between systems with different APIs of varying complexity. It also enables a comparison of the performance of synthesized drivers to that of the native drivers on the target OS.

We used RevNIC to port all drivers to our custom operating system, called KitOS, running on "bare" hardware. This OS initializes the CPU into protected mode and lets the driver use the hardware directly, without any OS-related overhead (no multitasking, no memory management, etc.). This experiment evaluates the performance of the synthesized drivers in the absence of OS interference. Bare hardware is the mode in which RevNIC would be used during the initial development of new drivers (KitOS boots instantly and starts executing immediately the driver, thus shortening the compile/reboot cycle, allowing developers to fix driver bugs quicker). Once the driver works properly, the developers can "transplant" the driver to the target OS.

We also ported the PCNet, RTL8139, and RTL8029 drivers back to Windows XP SP3. Porting to the same OS enables quantifying the overhead of the generated code with respect to the original Windows driver. In practice, porting to the same OS is useful when the binary driver exists for one version but not the other (e.g., 32-bit vs. 64-bit Windows), or when the original driver causes the OS to crash or freeze.

Test Hardware. We evaluate the performance of the synthesized drivers by running them on an x86 PC, an FPGA-based platform, and two virtual machines. This allows us to measure performance of generated drivers in a wide range of conditions. The PC and VMs run fresh installations of Windows XP SP3, Debian Linux 2.6.26, and KitOS. The FPGA system runs the µC/OS-II priority-based preemptive real time multitasking OS kernel for embedded systems.

We measure the performance of the RTL8139 driver on a PC based on an Intel Core 2 Duo 2.4 GHz CPU with 4 GB of RAM. The physical NIC is based on a Realtek RTL8139C chip, widely used in commodity desktop systems during the Windows XP era.

We evaluate the 91C111 driver on the FPGA4U [104] development board. It is based on an Altera Cyclone II FPGA with a Nios II processor, 32 MB of SDRAM, and an SMSC 91C111 network chip. The FPGA and the SDRAM run at 75 MHz, while the 91C111 chip runs at its native frequency of 25 MHz. This allows quantifying the overhead on a severely resource-constrained system.

Finally, we evaluate the RTL8029 driver on QEMU and the PCNet driver on VMWare. Virtualization is seeing increasing use in networked computing infrastructures, so performance in such an environment is important. The virtual machines are QEMU 0.9.1 and VMWare Server 1.0.10. The host OS is Windows XP x64 edition SP2 in both cases, running on a dual quad-core Intel Xeon CPU at 2 GHz, with 20 GB of RAM. QEMU uses a TAP interface for networking, while VMWare runs a NAT interface. VMs allow us to better zoom in on driver bottlenecks, which can be harder to observe on a real machine. For example, VMs disregard the rated speed of the NIC, so one can send data at even 1 Gbps using a driver for a 100 Mbps NIC (since there is no physical cable, the virtual NIC can confirm transmission immediately after the driver has given it all the data).

6.1.4.1 Effectiveness

RevNIC can extract all essential functionality from network device drivers. Table 6.3 shows the capabilities of the original NIC drivers compared to those of the reverse engineered drivers. A check mark indicates functionality available both in the original and the synthesized driver.

We identify the functionality implemented in the original driver by looking at the `QueryInformation` status codes supported by Windows, checking the configuration parameters in the registry that reveal additional functionality, and looking at the datasheets. For RTL8029 and PC-Net, given that the virtual hardware does not have LEDs and does not support Wake-on-LAN, we could not directly test these functions. However, the corresponding code was exercised and reverse engineered. The RTL8029 and the 91C111 chips support neither DMA nor Wake-on-LAN.

Functionality	AMD PCNet	RTL8139	SMSC 91C111	RTL8029
Init/Shutdown	✓	✓	✓	✓
Send/Receive	✓	✓	✓	✓
Multicast	✓	✓	✓	✓
Get/Set MAC	✓	✓	✓	✓
Promiscuous Mode	✓	✓	✓	✓
Full Duplex	✓	✓	✓	✓
DMA	✓	✓	N/A	N/A
Wake-on-LAN	N/T	✓	N/A	N/A
LED Status Display	N/T	✓	✓	N/T

Table 6.3 – Functionality coverage of reverse engineered drivers (N/A=Not available, N/T=Cannot be tested).

We manually checked the correctness of the reverse engineered functionality by comparing hardware I/O operations. For this, we ran the original driver on real hardware and recorded its I/O interaction with the device. Then, we ran the reverse engineered driver and compared the resulting I/O traces with that of the original driver. We exercised each function using a workload specific to the functionality in question. For example, to check send and receive, we transmitted several files via FTP. Checking the packet filter (i.e., promiscuous mode) involved issuing standard IOCTLs.

Finally, we manually checked that the original driver is a correct encoding of the hardware protocol specification. For this, we compared I/O interaction traces with the I/O sequence prescribed by the hardware specification. We focused on the send/receive functionality, since it is crucial for a network driver. We did not find meaningful discrepancies between the collected sample traces and the specifications. The original drivers did not have proprietary IOCTLs.

6.1.4.2 Performance

We evaluate the performance of the reverse engineered drivers by measuring throughput and CPU utilization. We first compare the original Windows driver to the synthesized Windows driver, in order to quantify the overhead of the code generated by RevNIC. Then we show the performance of drivers ported to a different operating system. We wrote a benchmark that sends UDP packets of increasing size, up to the maximum length of an Ethernet frame. In the case of KitOS, the benchmark transmits hand-crafted raw UDP packets, since KitOS has no TCP/IP stack. The reverse engineered drivers turn out to have negligible overhead on all platforms.

Figure 6.2 shows throughput and Figure 6.3 shows CPU utilization for the RTL8139 drivers. Synthesized drivers incur practically no overhead. The driver for KitOS is the fastest, since there is no TCP/IP stack overhead. For unknown reasons, the original Windows driver's performance drops for UDP packets over 1 KB; the reverse engineered driver does not have this problem. We also observe that the synthesized Windows driver has a slightly higher CPU utilization than the original, while both the native Linux and the ported Linux driver have a similar one for most packet sizes.

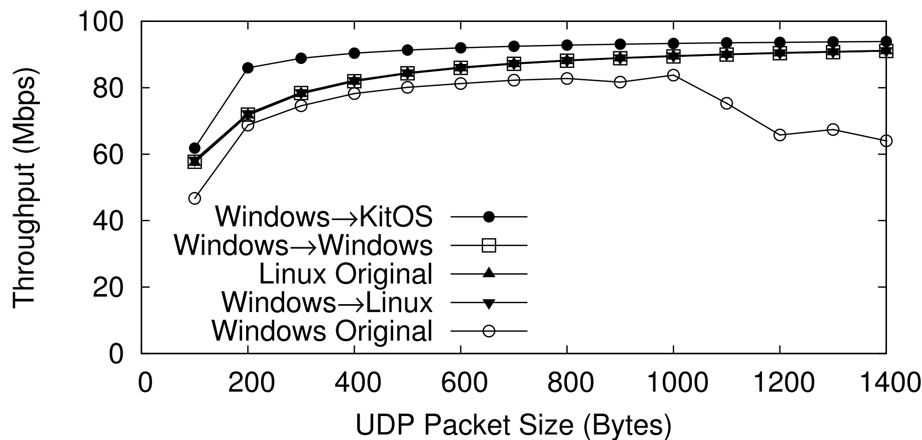

Figure 6.2 – RTL8139 driver throughput on x86.

Turning our attention to embedded systems, we note that synthesizing a driver for severely resource-constrained environments is one of the toughest performance challenges for RevNIC. Original drivers are typically hand-optimized, whereas RevNIC's drivers are not. In Figure 6.4, we show the performance of the 91C111 driver ported to the FPGA platform. Throughput is within 10% of the original driver, and we suspect this difference is mainly due to its cache footprint: the RevNIC-generated binary has 87KB, compared to 59KB for the native driver. With further optimizations on the generated code, we expect this 10% gap to be narrowed. CPU time spent in the synthesized 91C111 driver is comparable to that of the original (Figure 6.5), ranging roughly from 20% to 30% for both drivers. The overall CPU usage is 100%, since DMA is not available. The maximum achievable throughput is limited by the FPGA's system bus, shared between the NIC, the SDRAM, and other components.

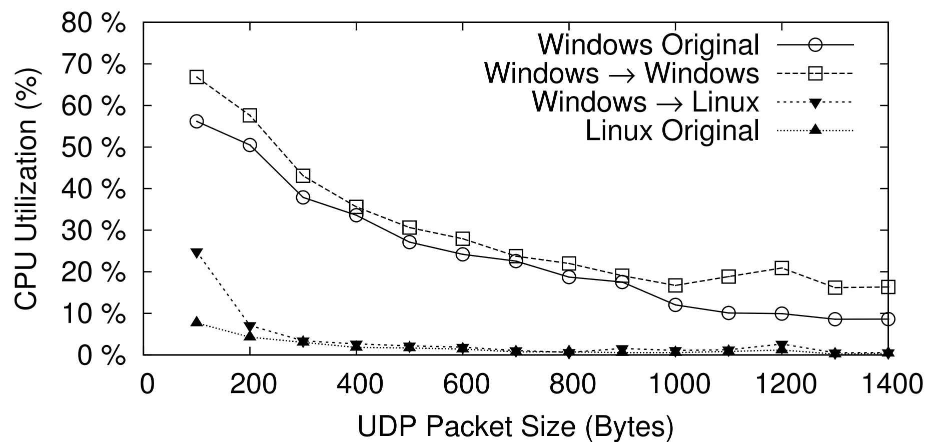

Figure 6.3 – CPU utilization for RTL8139 drivers on x86.

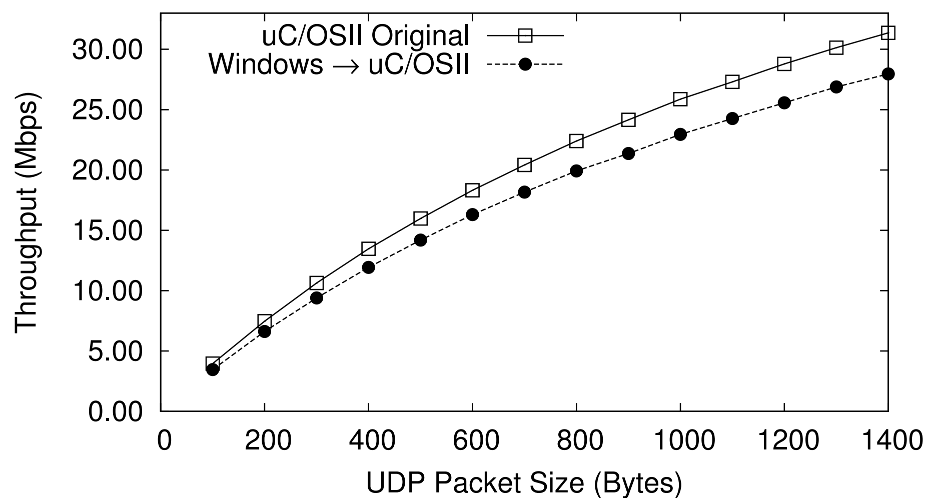

Figure 6.4 – 91C111 driver ported from Windows to an FPGA.

Finally, Figure 6.6 and Figure 6.7 show performance in virtualized environments. For QEMU, we show the RTL8029 driver, since QEMU provides an RTL8029-based virtual NIC. CPU utilization is close to 100% in all cases, since RTL8029 does not support DMA. The driver ported from Windows to Linux is on par with the native Linux driver. The lean KitOS driver again has the highest throughput. The difference between Linux and Windows is due to different behavior of TCP/IP stack implementations in the VM.

For VMware, which provides an AMD PCNet virtual NIC, we get similar results. Even though DMA is used, CPU utilization is still 100% in all cases, because the virtual hardware sends packets at maximum speed, generating a higher interrupt rate than that of real hardware. Performance on KitOS is lower, but same as that of the original Windows driver, most likely due to interactions with VM quirks.

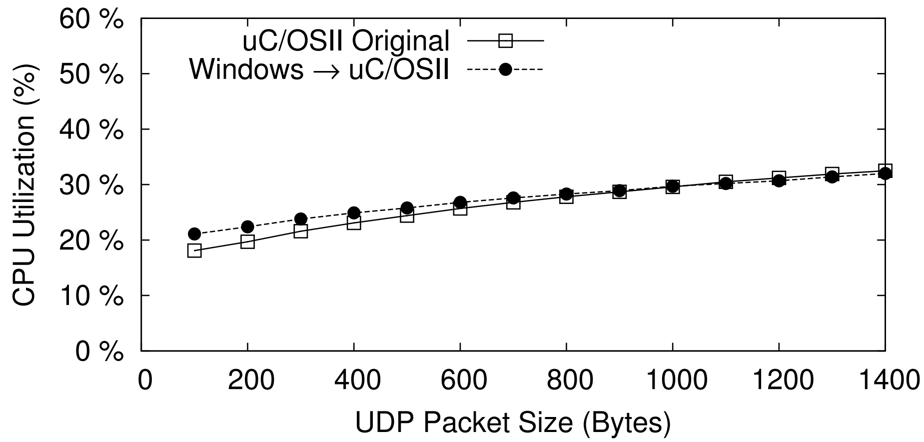

Figure 6.5 – CPU fraction spent inside the 91C111 driver.

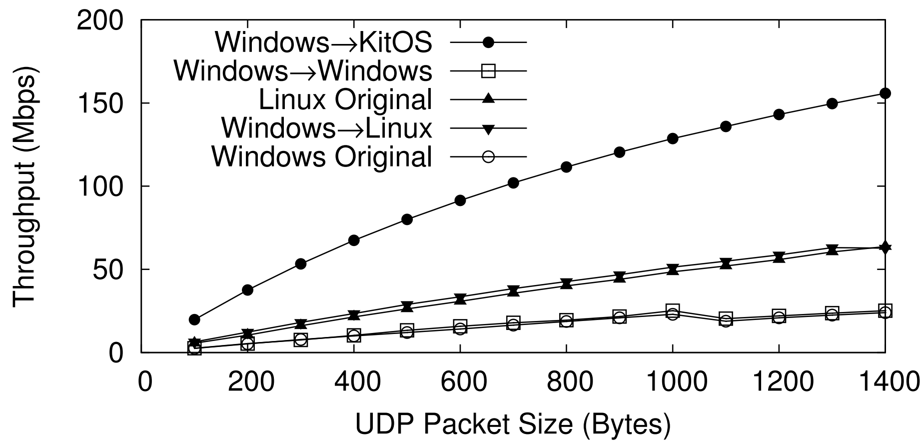

Figure 6.6 – RTL8029 throughput (QEMU).

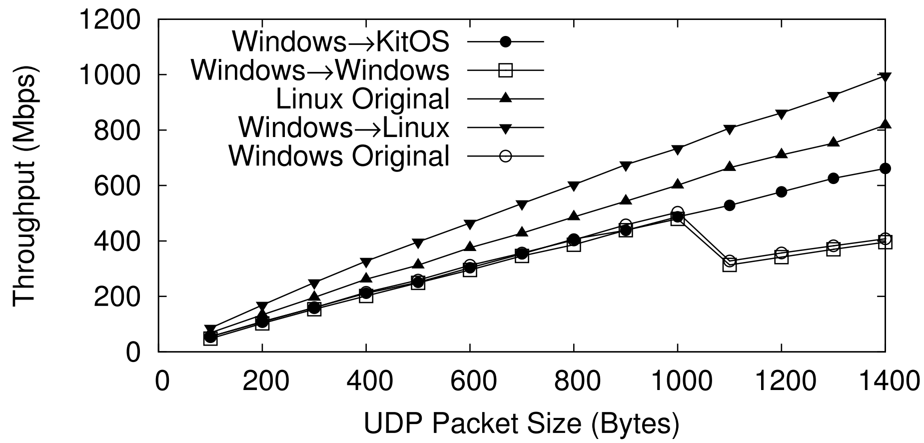

Figure 6.7 – AMD PCNet throughput (VMWare).

6.1.4.3 Automation

RevNIC exercises drivers and generates code in less than an hour. Template instantiation, though manual, takes orders of magnitudes less time than writing new drivers from scratch and does not require obtaining, reading, and understanding hardware specifications.

Obtaining portable C code. Obtaining the code for OS-independent and hardware-specific functionality is fully automated and fast. In Figure 6.8, we show how driver coverage varies with RevNIC running time—most tested drivers reach over 80% basic block coverage in less than twenty minutes, due to our use of symbolic execution. RevNIC stops either when all hardware-related functions get close to 100% coverage, or when a specified timeout expires (§6.1.2.3). The tests were run using the original RevNIC prototype [33] on a dual quad-core Intel Xeon CPU at 2 GHz, with 20 GB of RAM.

The running time and memory usage of the RevNIC code synthesizer is directly proportional to the total length of the traces it processes. RevNIC can process a little over 100 MB/minute. For the drivers we tested, code synthesis took from a few seconds to a few minutes.

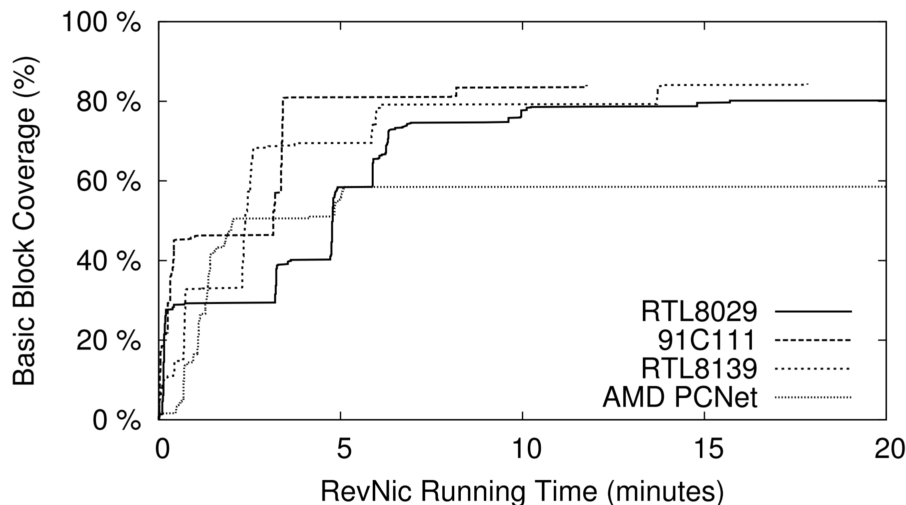

Figure 6.8 – Basic block coverage.

Writing a driver template. Producing NIC driver templates for the four OSes took a few days (Table 6.4). Writing a template is done manually, but it is a one-time effort, considerably simplified by using existing driver samples shipped with SDKs.

We first wrote one generic template for all NIC devices, and then extended it to provide DMA functionality for RTL8029 and 91C111. KitOS's network interface is limited to an initialization and send entry point, and a receive callback. The driver can directly talk to the hardware, without

Target OS	Person-Days
Windows	5
Linux	3
μC/OS-II	1
KitOS	0

Table 6.4 – Time to write a template for NIC drivers.

interacting with the OS. The template for μC/OS-II is slightly more complex than the one for KitOS, because it requires implementing more entry points and the use of APIs (e.g., memory management). It took one day to write it.

Integrating Hardware Interaction Code in the Template. A large portion of drivers' code is hardware-specific. In Figure 6.9, we show what fraction of the driver is fully reverse-engineered by RevNIC. hardware-related function to be completely reverse-engineered when execution engine did not exercise them). Overall, about 70% of the functions are fully synthesized. The other functions contain mostly OS-related code and correspond to high-level functions of the device drivers, like send and receive. They also include functions that mix OS and hardware calls (~10%–15% per driver). These functions are only partly exercised, and the corresponding traces serve as hints to the developer for the template integration.

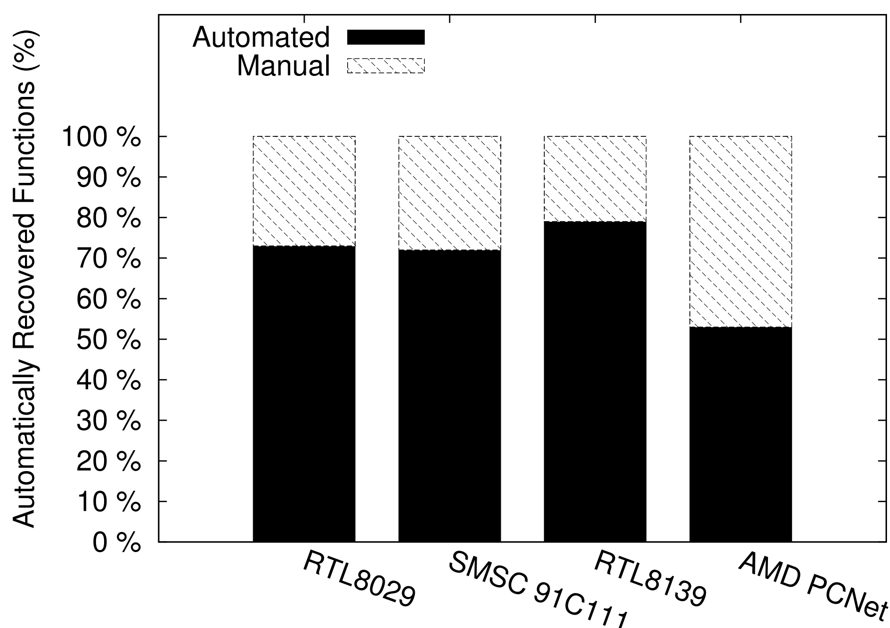

Figure 6.9 – Breakdown of OS-specific vs. hardware-specific functions (% of recovered functions).

Writing drivers, even from specifications, is hard. Table 6.5 attempts to give a rough estimation of how much effort it took to write and/or debug Linux open source drivers. We looked at how many developers were acknowledged in the headers of the source files and at the reported time span for development. The numbers also include adaptations to newer versions of Linux. Even when assuming that developers do not work full-time on a driver, but on a best-effort basis (like in the open source community), it still takes a considerable effort. The change logs of the RTL8139 driver suggest that most time went into coding workarounds for undocumented hardware quirks.

In contrast, RevNIC uses the original proprietary driver, which has all the hardware details for all supported devices readily available. When using RevNIC, most developer time goes into instantiating the driver template. This is roughly proportional to the size of the driver and the number of hardware functions it implements. Table 6.5 also includes the time to debug RevNIC, since porting the drivers and debugging our prototype were done together. Debugging required manually checking the synthesized C code against the original driver's binary; this took 1-3 days, depending on driver size.

Device	Manual (Linux)		RevNIC	
	Persons	Span	Persons	Span
RTL8139	18	4 years	1	1 week
SMSC 91C111	8	4 years	1	4 days
RTL8029	5	2 years	1	5 days
AMD PCNet	3	4 years	1	1 week

Table 6.5 – Amount of developer effort. RevNIC numbers include time to debug the RevNIC prototype itself.

6.1.4.4 Scalability

We have shown that our approach works for drivers from 18KB to 35KB. While most network device drivers fall into this size range, modern drivers for Ethernet and Wifi are often two orders of magnitude larger, exceeding 100,000 basic blocks.

Scalability is limited by the performance of symbolic execution. Symbolic execution is subject to exponential state growth and memory consumption [28, 27, 102, 112], both of which affect RevNIC. We improved scalability in REV$^+$ by using several new features offered by S^{2}E, such as parallel symbolic execution, that we evaluate next. Further improvements in the field of symbolic execution will automatically benefit to RevNIC. We discuss these limitations and how we plan to address them in Chapter 8.

6.1.4.5 RevNIC vs. REV$^+$

In this section, we compare the performance of RevNIC with REV$^+$, an S^{2}E-based implementation of RevNIC. In order to evaluate the impact of the number of cores on exploration speed, we run the tests on a 48-core, 2.0 GHz AMD Opteron machine with 512 GB of RAM.

We ran REV$^+$ on the same drivers as for RevNIC, and REV$^+$ reverse engineers them with better coverage than RevNIC in the same amount of time (see Table 6.6). Manual inspection of the reverse engineered code blocks reveals that the resulting drivers are equivalent to those generated by RevNIC, and thus to the originals too [33]. Figure 6.10 shows how coverage evolves over time during reverse engineering both for single-core S^{2}E [34] and multi-core S^{2}E. The single-core prototype completes the exploration of each driver in 30-90 minutes, whereas multi-core S^{2}E completes in a few minutes.

	RevNIC	**REV$^+$ (S^{2}E)**		**REV$^+$ (Multi-Core S^{2}E)**	
	Coverage	Coverage	Increase	Coverage	Increase
PCnet	59%	66%	+7%	74%	+15%
91C111	84%	87%	+3%	89%	+5%
RTL8139	84%	86%	+2%	89%	+5%

Table 6.6 – Basic-block coverage obtained by RevNIC, REV$^+$, and REV$^+$ using 48-core S^{2}E. We also show the coverage increase over RevNIC.

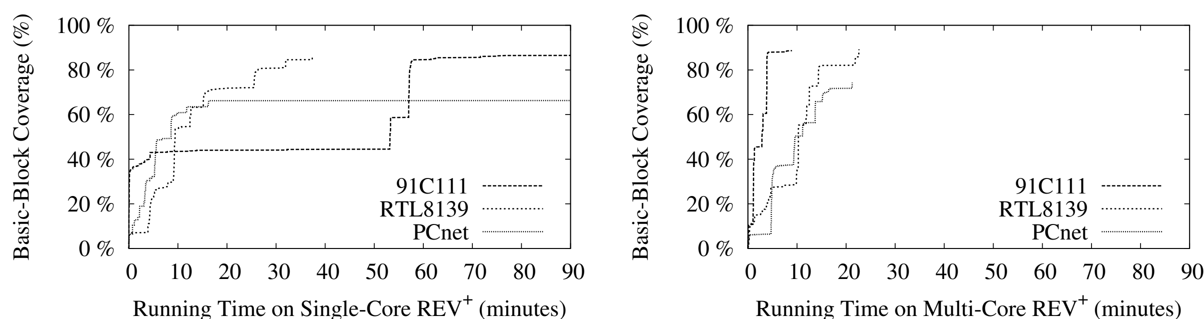

Figure 6.10 – Basic-block coverage over time for REV$^+$ and REV$^+$ on multi-core S^{2}E. The graph for REV$^+$ on multi-core S^{2}E shows data for 91C111, RTL8139, and PCnet on 48 cores.

Table 6.7 shows that more cores allow REV$^+$ to explore many more paths and achieve higher basic block coverage. The exploration time increases with the number of cores because REV$^+$ invokes each driver entry point sequentially, and when it does not discover any new basic block within some time interval, it kills all paths except one and invokes the next entry point. Since multiple cores can explore more paths in parallel and have thus a higher likelihood of discovering

	PCnet			RTL8139			91C111		
# of cores used by S^{2}E	Time (min)	Cov (%)	# Paths	Time (min)	Cov (%)	# Paths	Time (min)	Cov (%)	# Paths
1	9	64	4,551	5	76	4,274	6	85	2,185
2	10	66	8,466	7	86	13,841	7	84	2,287
4	14	67	18,631	17	83	27,266	6	85	5,853
8	17	69	38,919	32	84	61,743	7	87	15,322
16	19	71	126,590	17	88	164,426	10	89	31,625
32	21	74	257,813	27	87	398,895	12	89	100,977
48	26	70	539,037	22	89	554,458	21	89	145,498

Table 6.7 – Impact of additional cores on exploration time, basic-block coverage, and number of explored paths for REV$^+$ on multi-core S^{2}E using the overapproximate consistency model.

more basic blocks, REV$^+$ resets the timeout more frequently, thus increasing the average exploration time. We used a timeout of 5 to 10 seconds for multi-core S^{2}E and up to 1200 seconds for the single-core S^{2}E prototype combined with a random-path search heuristic [27]. Given these settings, 32 cores are enough to get 74% to 89% basic-block coverage in less than 25 minutes.

Although more cores allow exploring more paths, this does not necessarily yield higher basic-block coverage. First, S^{2}E does not ensure that different cores do not perform redundant work. It may happen that all the cores explore one particular part of a driver (e.g., one function) instead of covering different parts. In future work, we plan to focus on achieving disjunction of explored paths, as in Cloud9 [21], in order to minimize the amount of redundant work. Second, we believe that more cores influence the path selection heuristics that S^{2}E uses; we plan to investigate this phenomenon in future work.

In general, it is hard to test device drivers with pure symbolic hardware. Symbolic hardware returns unconstrained register values on every read, which forks execution on every branch that depends on such a value. The path selection heuristic does not know which branch outcome would have been chosen had the driver been running on real hardware. We are currently exploring ways of combining *real* and symbolic hardware in order to guide symbolic execution. Real hardware provides the concrete input that drives symbolic execution in a depth-first manner, preventing path exploration from getting stuck in various parts of the driver. On each branch that depends on symbolic hardware input, the path selection heuristic evaluates the branch condition by plugging the concrete values from the real hardware. The result indicates which path the heuristic should follow first. This path would have been taken had the driver been run on a real machine with concrete inputs.

6.1.5 Discussion and Limitations

RevNIC cannot produce a driver that is "more correct" than the original binary with regards to hardware interaction. It is hard to fix buggy hardware interaction, when there are no specifications. However, certain classes of bugs, like unchecked use of array indexes coming from hardware or buffer overflows are eliminated by reverse engineering, resulting in a safer driver.

Reverse engineering of proprietary IOCTLs is similar to that of standard entry points. IOCTLs encode functions that were not foreseen by the OS driver interface designers. Instead of triggering proprietary behavior using standard OS APIs, RevNIC must use the vendor-supplied configuration tools for doing so. For the standard interface, the semantics of the behavior are provided by the OS interface; for the proprietary one, the semantics are derived from the tool's documentation.

RevNIC-generated code is not as readable as the original source, because it does not reconstruct high-level C statements, like loops. The generated code relies on `goto` for control flow. We believe that existing transformation techniques [37] can make generated code more readable. However, the produced code is substantially more accessible than disassembly. The generated code is easier to understand and adapt, because it uses familiar C operators, instead of x86 instructions. Moreover, C code can be easily compiled to any OS or processor architecture, unlike assembly.

Of course, RevNIC can be rerun easily every time there is an update to the original binary driver. The resulting source code can be compared to the initially reverse engineered code and the differences merged into the reverse engineered driver, like in a version control system. One could also use binary diffing methods [18] to update the synthesized driver every time there is a new patch for the proprietary driver that fixes hardware-related bugs. Thus, we expect RevNIC-generated code to require minimal maintenance.

Although porting between two OSes by instantiating a driver template requires substantial code refactoring, one could automate it to a certain extent. For example, Coccinelle [98] automatically translates device drivers between two versions of the same OS. RevNIC could treat two different OSes as an evolution from one to the other. Another possibility is to synthesize a specification from the binary and use existing tools, like Termite [107], to automatically generate a driver for any target OS, solving once and for all the safety and portability problems of device drivers.

RevNIC currently supports NIC drivers, but it is in theory possible to extend it to any class of device drivers. Exercising the driver and generating the code is device-agnostic: all RevNIC needs is OS and hardware input. The developer has to write a device driver template for the new class of devices, and this requires a general understanding of what the device class is supposed to do and how it interfaces with the OS (e.g., that a sound card is supposed to play sound by copying a buffer to some memory, very much like a NIC sends a packet after it is copied to some buffer).

Finally, RevNIC is not meant for reverse engineering the internals of a device, only its interaction with the driver. For instance, devices like graphics cards can compile and run code internally

(such as vertex shaders). Reverse engineering firmware or the particular programming language of a chip is beyond the scope of RevNIC. What a tool like RevNIC could do for a graphics driver is to extract the initialization steps and set up a frame buffer, possibly extracting 2D acceleration, if it only involves I/O. It could also make a synthesized driver replay hardware interactions (e.g., upload firmware to the card) the same way the original driver would.

6.1.6 How RevNIC's Limitations Influenced S^{2}E's Design

The original tracer combined QEMU and KLEE, but in a way that had several limitations that we later overcame in S^{2}E. There were three main limitations: RevNIC was a 32-bit tool limited to 4GB of RAM, did not support symbolic memory addresses, and did not have a consistent state sharing between concrete and symbolic domains.

Symbolically executing drivers can generate tens of thousands of states. Even though RevNIC was compiled to use the full 4GB of virtual address space, RevNIC required more memory. We implemented swapping in order to move unused KLEE memory objects to disk and free the address space in order to explore more states. However, RevNIC was still running out of address space because of the overhead of the additional bookkeeping. When later building S^{2}E, we made it 64-bit right from the start, rendering unnecessary the explicit swapping mechanism. The host OS performs the swapping by itself, if necessary.

RevNIC avoided the complexity of dealing with symbolic pointers by concretizing them. Reasoning about all possible addresses for each memory access that uses a symbolic location is expensive. Symbolic pointers occur when symbolic input is used to reference memory, e.g., when a symbolic IOCTL number is used as an index in a table. Since there are typically only a few concrete values, RevNIC generated all of them and forked the execution for each such value. S^{2}E can behave in the same way, or instead keep the address symbolic and pass it as is to the solver. The latter ensures no false negatives, but at the expense of increased solving complexity.

Finally, RevNIC did not have a consistent state sharing between symbolic and concrete domains. In the way RevNIC was implemented, symbolic data could not flow outside of the driver. RevNIC ran the whole driver in KLEE and executed the rest of the system concretely. The symbolic state was not shared with the concrete domain (memory or CPU registers). If the system accessed a memory location or a CPU register that contained a symbolic value, RevNIC would return the concrete value that was previously there. Similarly, a concrete write outside of the driver would not discard the symbolic value stored there. This happened to work for the drivers that we considered for RevNIC, because they did not leak symbolic data into the system, but became a major problem later that we solved in S^{2}E, in order to enable testing multiple components that interact with each other.

6.1.7 Related Tools

Building portable drivers has been a goal of several previous projects, such as UDI [101], Devil [84], and HAIL [118]. Recent work, like Termite [107], proposes a formal development process in which a tool generates drivers from state machine specifications. These approaches require vendors to provide formal specifications of the hardware protocols. RevNIC complements these efforts by extracting the encoding of the protocol from existing device drivers, making the task of reverse engineering existing drivers more productive. In some sense, RevNIC can help tools like Termite become practical. VM-based approaches [79] can reuse existing binary drivers, but are generally heavyweight. Approaches like NDISwrapper [95] can directly reuse existing drivers by emulating the source OS. However, wrappers may be prone to bugs, add overhead, and work only on the OS for which they were developed. In contrast, RevNIC makes the reverse engineered driver independent from the source OS and even the source computer architecture.

Most of the existing techniques for improving device driver safety rely on source code [120, 124, 127]. Since RevNIC can obtain a source code representation of a driver binary, it can enable the use of these tools on closed-source, proprietary device drivers, to improve their reliability. For example, some binary drivers do not have proper timeouts in polling loops; this would be straightforward to fix using [66]. OS-related safety properties could be checked prior to compilation [5], or the driver could be split to enhance reliability [51]. Furthermore, if the driver generation follows a formal development approach (as in [49] or [107]), it is possible to guarantee that a reverse engineered driver will not crash the system, hang, and introduce security vulnerabilities.

In using VMs to observe system activity, we build upon a rich set of prior work, including tools such as Aftersight [36] and Antfarm [64]. Reverse debugging [70] used VMs to debug systems, including device drivers. Symbolic execution has also been used for program testing [28, 27, 55, 112] and malware analysis [92, 41, 131]. We extended these approaches to provide kernel-mode instrumentation. In RevNIC, we combine VM-based wiretapping with symbolic execution to exercise control on the analyzed system. Reverse engineering often uses static decompilation [14]; this, however, faces a number of challenges (e.g., disambiguating code from data), so we minimized RevNIC's reliance on static decompilation.

Recent work has been aimed at automatically reverse engineering message formats in network protocols [43] as well as files [44], based on traces containing these messages. Our reliance on driver activity traces is similar but, due to the specifics of device drivers, RevNIC manages to also reverse engineer the relationship between hardware registers, not just the format. RevNIC extracts the semantics of driver code dynamically, using traces of memory accesses.

6.1.8 Summary

We presented a new approach and tool built on top of S^2E for reverse engineering binary device drivers. One can use this approach to either port drivers to other OSes or to produce safer drivers for the same OS.

The tool, called RevNIC, does not require access to any device documentation or driver source code—it relies on collecting hardware interaction traces and synthesizing, based on these, a new, portable device driver. We showed how RevNIC reverse engineered several closed-source Windows NIC drivers and ported them to different OSes and architectures. RevNIC produces drivers that run natively with performance that is on par with that of the target OS's native drivers.

6.2 Deriving Performance Envelopes with Multi-Path Profiling

To further illustrate S^2E's generality, we used it to develop PROF_S, a multi-path in-vivo performance profiler and debugger. To our knowledge, such a tool did not exist previously, and we believe this use case is the first in the literature to employ symbolic execution for performance analysis. In this section, we show through several examples how PROF_S can be used to predict performance for certain classes of inputs. To obtain realistic profiles, performance analysis can be done under local consistency or any stricter consistency model.

PROF_S allows users to measure instruction count, cache misses, TLB misses, and page faults for arbitrary memory hierarchies, with flexibility to combine any number of cache levels, size, associativity, line sizes, etc. This is a superset of the cache profiling functionality found in Valgrind [123], which can only simulate L1 and L2 caches, and can only measure cache misses.

For PROF_S, we developed the *PerformanceProfiler* plugin. It counts the number of instructions along each path and, for memory reads/writes, it simulates the behavior of a desired cache hierarchy and counts hits and misses. For our measurements, we configured PROF_S with 64-KB I1 and D1 caches with 64-byte cache lines and associativity 2, plus a 1-MB L2 cache that has 64-byte cache lines and associativity 4. The path exploration in PROF_S is tunable, allowing the user to choose any execution consistency model.

The first PROF_S experiment analyzes the distribution of instruction counts and cache misses for Apache's URL parser. In particular, we were interested to see whether there are opportunities for a denial-of-service attack on the Apache web server via a carefully constructed URL. The analysis ran on PROF_S using 48 cores under local consistency for 1 hour and explored 51,530 different execution paths. The analysis spent 44% of the running time in the constraint solver.

We found each path involved in parsing a URL to take on the order of 4.3×10^6 instructions, with one interesting feature: for every additional "/" character present in the URL, there are 10

extra instructions being executed. We found no upper bound on the execution of URL parsing: a URL containing $n+k$ "/" characters will take $10 \times k$ more instructions to parse than a URL with n "/" characters. The total number of cache misses on each path was predictable at $13,315 \pm 65$. These are examples of behavioral insights one can obtain with a multi-path performance profiler. Such insights can help developers fine-tune their code or make it more secure (e.g., by ensuring that password processing time does not depend on its content, to avoid side channel attacks).

We also set out to measure the page fault rate experienced by the Microsoft IIS web server inside its SSL modules while serving a static page workload over HTTPS. Our goal was to check the distribution of page faults in the cryptographic algorithms, to see if there are opportunities for side channel attacks. We found no page faults in the SSL code along any of the paths, and only a constant number of them in `gzip.dll`. This suggests that counting page faults should not be the attack of first choice if trying to break IIS's SSL encryption.

Next, we aimed to establish a performance envelope in terms of instructions executed, cache misses, and page faults for the ubiquitous `ping` program (1.3 KLOC). The performance analysis ran under local consistency, focusing exploration on the IP packet options parser. S^2E explored 907 different paths in 1 hour using 48 cores. Around 30% of the time was spent in the constraint solver. Note that, in [34], we focused the analysis on the entire packet-parsing code but with additional constraints on the packet content to prevent path explosion; we now analyze the part of the parser that focuses on packet options but include all possible paths through it. This makes the results easier to interpret because, now, all the obtained paths go through the packet options parser.

The analysis does not find a bound on execution time, and it points to a path that could enter an infinite loop. This happens when the reply packet to `ping`'s initial packet has the record route (RR) flag set and the option length is 3 bytes, leaving no room to store the IP address list. While parsing the header, `ping` finds that the list of addresses is empty and, instead of `break`-ing out of the loop, it does `continue` without updating the loop counter. This is an example where performance analysis can identify a dual performance and security bug: malicious hosts could hang `ping` clients. Once `ping` is patched, the performance envelope becomes 2,581 to 2,728 executed instructions. With the bug, the maximum during analysis had reached 1.1×10^6 instructions and kept growing.

PROF_S can find "best case performance" inputs without having to enumerate the input space. For this, we modify slightly the *PerformanceProfiler* plugin to track, for all paths being explored, the common lower bound on instructions, page faults, etc. Any time a path exceeds this minimum, the plugin automatically abandons exploration of that path, using the *PathKiller* selector described in chapter 4. This type of functionality can be used to efficiently and automatically determine workloads that make a system perform at its best. This use case is another example of performance profiling that can only be done using multi-path analysis.

To conclude, we used S^{2}E to build a thorough multi-path in-vivo performance profiler that improves upon classic profilers. Valgrind [123] is thorough, but only single-path and not in-vivo. Unlike Valgrind-type tools, PROF$_S$ performs its analyses along multiple paths at a time, not just one, and can measure the effects of the OS kernel on the program's cache behavior and vice versa, not just the program in isolation. Although tools like Oprofile [80] perform in-vivo measurements, but not multi-path, they are based on sampling, so they lack the accuracy of PROF$_S$—it is not feasible, for instance, to count the exact number of cache misses in an execution. Figure 6.11 summarizes the capabilities of Valgrind, Oprofile, and PROF$_S$. Such improvements over state-of-the-art tools come relatively easily when using S^{2}E to build new tools.

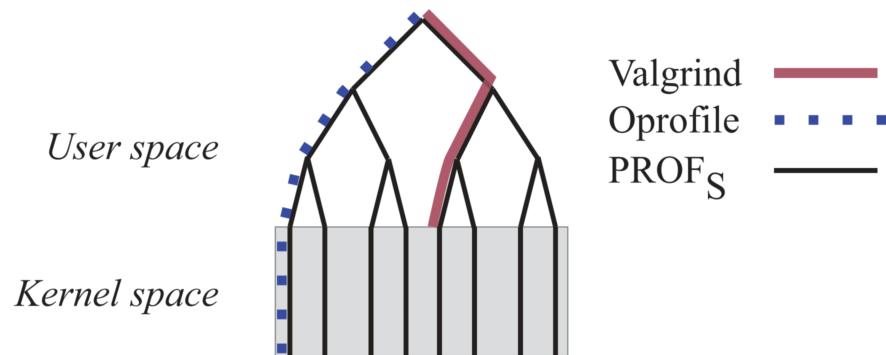

Figure 6.11 – The path coverage of classic profilers vs. the coverage level enabled by S^{2}E: PROF$_S$ is a thorough multi-path in-vivo analyzer; Oprofile is in-vivo, but only single-path and sampling-based; Valgrind is thorough, but only single-path and not in-vivo.

6.3 Other Tools

We believe S^{2}E can be used for pretty much any type of system-wide analysis. We give here several additional examples, including use cases from other researchers.

6.3.1 Automated Testing of Proprietary Device Drivers

We used S^{2}E to build DDT$^+$, a tool for testing closed-source Windows device drivers. This is a reimplementation of DDT [74], an ad-hoc combination of changes to QEMU and KLEE, along with hand-written interface annotations: 35 KLOC added to QEMU and 7 KLOC modified, 3 KLOC added to KLEE and 2 KLOC modified. By contrast, DDT$^+$ has 720 LOC of C++ code, which combine several exploration and analysis plugins, and provides the necessary kernel API annotations to implement RC-LC.

DDT$^+$ combines several plugins: the *CodeSelector* plugin restricts multi-path exploration to the target driver, while the *MemoryChecker*, *DataRaceDetector*, and *WinBugCheck* analyzers look

for bugs. To collect additional information about the quality of testing (e.g., coverage), we use the *InstructionTracer* analyzer plugin. Additional checkers can be easily added. DDT+ implements local consistency (RC-LC) via interface annotations that specify where to inject symbolic values while respecting local consistency—examples of annotations appear in [74]. None of the bugs reported by DDT+ are false positives, indicating the appropriateness of local consistency for bug finding. In the absence of annotations, DDT+ reverts to strict consistency (SC-SE), where the only symbolic input comes from hardware.

We ran DDT+ on two Windows network drivers, RTL8029 and AMD PCnet; DDT+ finds the same 7 bugs reported in [74], including memory leaks, segmentation faults, race conditions, and memory corruption. Of these bugs, 2 can be found when operating under SC-SE consistency; relaxation to local consistency (via annotations) helps find 5 additional bugs. DDT+ achieves 42 % basic-block coverage of the PCnet driver in 30 minutes, exploring more than 164,000 paths. For the RTL8029 driver, DDT covers 76% and 380,000 paths in less than 15 minutes.

For each bug found, DDT+ outputs a crash dump, an instruction trace, a memory trace, a set of concrete inputs (e.g., registry values and input from hardware devices) and values that were injected according to the RC-LC model that trigger the buggy execution path.

While it is always possible to produce concrete inputs that would lead the system to the desired local state of the unit (i.e., the state in which the bug is reproduced) along a globally feasible path, the exploration engine does not actually do that while operating under RC-LC. Consequently, replaying execution traces provided by DDT+ usually requires replaying the symbolic values injected into the system during testing. Such replaying can be done in S²E itself. Despite being only locally consistent, the replay is still effective for debugging: the execution of the driver during replay is valid and appears consistent, and injected values correspond to the values that the kernel could have passed to the driver under real, feasible (but not exercised) conditions.

S²E generates crash dumps readable by Microsoft WinDbg [88]. Developers can thus inspect the crashes using their existing tools, scripts, and extensions for WinDbg. They can also compare crash dumps from different execution paths to better understand the bugs.

6.3.2 Finding Bugs in Linux Device Drivers with SymDrive

SymDrive [105] is a tool that brings *static analysis* to S²E in order to test Linux and FreeBSD device drivers more effectively. SymDrive combines static analysis with source code instrumentation and dynamic analysis in order to make path exploration more efficient. Its static analysis component analyzes drivers for loops and forwards loop information to the path selection heuristic, which favors paths that exit loops early (loop elision). It also favors paths that exercise the "good" behavior of the driver rather than the error recovery paths.

SymDrive also simplifies driver testing by letting developers write annotations and checkers directly in kernel code. Like in DDT, these annotations inject symbolic values into the driver in order to enable multi-path exploration. Checkers verify various properties, e.g., that the driver calls the kernel API correctly, and if not, flag an error.

SymDrive found 39 distinct bugs in 26 device drivers. SymDrive can detect API misuse, races, allocation/deallocation mismatches, memory leaks, bad pointers, and various hardware-dependent bugs. SymDrive achieves better scalability than DDT thanks to its static analysis component. It is a challenge for DDT to find execution paths that successfully exercise most of the driver's entry points because its search heuristics get stuck early on inside the driver's error recovery code. SymDrive can however deprioritize such paths thanks to its global view of the driver's code.

6.3.3 Scalable Testing of File System Checkers with SWIFT

SWIFT [30] is a tool that systematically explores the recovery code of file system checkers. It tests that in case the checker reports a successful recovery, the file system is not in a corrupted state. Moreover, if the checker reports an error, SWIFT verifies that the disk state is consistent with the reported error. SWIFT also tests that the checker recovers as much information as possible from the corrupted disk.

SWIFT uses S^2E to inject faults into a healthy file system and then symbolically execute the file system checker. SWIFT invokes the checker twice on each execution path in order to test the checker's consistency. If the first invocation fixed the errors, the second invocation should not find more errors. To check for completeness of the recovery, SWIFT runs multiple checkers in parallel for possibly different file systems. Semantically identical faults (e.g., similarly corrupted inode) should normally produce similar behavior and recovery regardless of the file system.

SWIFT adds *selective concretization* to S^2E in order to scale to larger constraints. Analyzing file system checkers frequently produces large constraints that are difficult to solve. A straightforward solution would be to concretize all the variables involved in the difficult constraints. However, this severely limits the state space, introducing false negative. To mitigate this, SWIFT allows users to define a concretization policy. Such a policy can, for example, concretize only a subset of variables. The subset could include variables that are most frequently used or those that appear in complex arithmetic expressions.

SWIFT found 5 bugs in `e2fsck`, 2 bugs in `reiserfsck`, and 3 bugs in `fsck.minix`. Bug types include segmentation faults in the checkers, infinite loops, data loss, and various incorrect recoveries. It took SWIFT from 9.5 to 37 hours to explore up to 163,000 paths in these file system checkers.

6.3.4 Prototyping Symbolic Execution Engines for Interpreted Languages with CHEF

CHEF is a recipe and tool that transforms vanilla interpreters (such as Python or Lua) into sound and complete symbolic execution engines. CHEF runs the interpreter inside S^{2}E and uses a path exploration heuristic in order to focus testing the high-level code (e.g., Python program or library) rather than the low-level implementation of the interpreter.

The insight behind CHEF is that one high-level path through an interpreted program can actually correspond to many low-level paths in the interpreter. For example, a simple assignment in Python translates to many checks in the interpreter, which in case of a symbolic operand would fork several paths. CHEF uses a heuristic that groups together paths that forked at the same high-level instruction, then selects one group, and finally picks one path in the group. This prevents imbalance issues in case one high-level instruction forks many more paths than the other instructions.

Conceptually, CHEF relies on S^{2}E's selectivity and path selection mechanisms in order to match S^{2}E's low level of abstraction to that of higher-level languages. S^{2}E operates natively at the machine code level, which is well suited for analyzing low level code, such as C/C++ programs. While S^{2}E can run programs written in interpreted languages, the gap between the levels of abstraction makes it hard for analyzers to reason about such programs. CHEF bridges this gap.

CHEF allowed building a symbolic execution engine for Lua in 3 person-days and one for Python in 5 person-days. The path exploration heuristics implemented by CHEF yielded up to 1,000 $\times$ speedup compared to running an interpreter on vanilla S^{2}E. CHEF also found bugs in existing symbolic execution engines.

6.3.5 Finding Trojan Message Vulnerabilities in Distributed Systems with Achilles

Achilles is a tool that that searches for trojan messages in distributed systems [7]. Trojan messages are messages that look correct but cannot actually be generated by any correct sender. Such messages can be a major source of vulnerability because they are often not tested by developers. Achilles analyzes the senders and receivers in order to extract the message grammar that the former generate and the latter accept. Achilles then computes the trojan messages as the difference between the two grammars.

Achilles uses S^{2}E to symbolically explore the distributed system and find trojan messages. On the receiver side, Achilles makes the input packets symbolic and explores the paths that handle accepted messages. The disjunction of path constraints over such paths form the receiver's grammar. On the sender side, Achilles makes all local inputs symbolic (e.g., syscall return values) in order

to explore different paths through the sender, generating different types of symbolic packets and forming the sender's grammar. The set of trojan messages are the messages that satisfy the path constraints of the receiver, but not those of the sender.

Achilles found several bugs in FSP, a UDP-based file transfer protocol, and PBFT, a Byzantine-fault-tolerant replication system. Achilles discovered that FSP clients do not allow filenames that contain wildcard characters. However, FSP servers accept them. It is thus possible for a malicious client to create files on the server that cannot be directly manipulated by valid clients. In the case of PBFT, Achilles found that the replica servers accept client requests without checking their authentication code. Replicas that later receive an incorrect code force the system to enter an expensive recovery mechanism.

6.3.6 Scalable Testing of Distributed Systems with SymNet

Testing distributed systems poses several scalability challenges. One approach to test such systems would be to run each node of a system as a separate process in a guest OS running in S2E and let these nodes communicate via a virtual network. First, this does not scale to many nodes because a fork in one node would fork the state of the entire system. Second, symbolic execution is orders of magnitudes slower than native execution, causing time drifts and synchronization issues when nodes run at a difference pace. Third, this approach makes it difficult to test different implementations of the same protocol running on different types of hardware.

SymNet [108] solves these challenges by running each node of the distributed system in a separate S^2E instance and having a coordinator to keep the nodes synchronized. When a node sends a symbolic packet, it serializes the associated constraints and concretizes the packet. The receiving S^2E instance turns the packet back symbolic according to the deserialized constraints. The coordinator ensures that the packets are received in the context of the right execution state on the target node and makes sure that each node has a consistent view of the virtual time by enforcing round-based communication.

SymNet expands the limits of the environment that S^2E can handle. S^2E cannot control the environment outside of its virtual machine. First, S^2E must concretize all the symbolic data that leaves it. Second, S^2E cannot track constraints in the outside world. Therefore, any incoming data can clobber any execution state that happens to be running, making the system inconsistent. This is similar to the state clobbering problem encountered by KLEE [27] whenever it invokes a system call. SymNet alleviates these problems, making S^2E aware of the world outside of its boundaries.

SymNet found several assertion failures in the Linux implementation of the Host Identity Protocol (HIP). Marking the version and the reserved field of the header symbolic uncovered paths in which these header fields cause sanity check to fail incorrectly.

6.3.7 Security Analysis of Embedded Systems' Firmware with AVATAR

Traditional analysis tools need to emulate the entire embedded system or provide models of various hardware components in order to be able to run and test the firmware. The problem is that there are many types of devices, each having its own unique set of hardware peripherals. Combined with a lack of documentation, this makes modeling or emulation impractical.

AVATAR is a platform for analysis of embedded systems' firmware that combines multi-path analysis with real hardware [133]. It alleviates the need of emulating the hardware. AVATAR executes ARM firmware in S^{2}E while forwarding hardware I/O to the real device. The device provides a concrete skeleton execution path through the firmware and S^{2}E fuzzes around that main path in order to find corner-case vulnerabilities deep in the code, without getting stuck at the start of the program. AVATAR can also selectively execute timing-sensitive parts of the firmware natively on the device itself (at full speed), or symbolically in S^{2}E. This solves the challenge of the slow communication speed between the device and the emulator (a few kilobits per second over JTAG or serial ports), which prevents heavy data transfers necessary for state synchronization.

AVATAR leverages and extends the selectivity provided by S^{2}E and its ability to communicate with hardware. S^{2}E allows selecting which portion of code to run symbolically and which to run concretely, by defining symbolic and concrete domains. AVATAR adds a *native* domain, which lets users run code concretely inside the physical device, bypassing emulation overhead while preserving execution consistency.

AVATAR was used to detect backdoors in a harddisk's firmware, analyze a ZigBee sensor for vulnerabilities, and reverse engineer a GSM feature phone. The hard disk has a text-based configuration interface. AVATAR verified that the interface did not have any hidden commands but found that some of them were more permissive than the specification. In the wireless sensor, AVATAR found a manually-injected vulnerability. Finally, AVATAR found portions of the phone's SMS parsing code that manipulates pointers derived from user input. The pointers turned out to be overly constrained in order to be exploitable.

6.3.8 Verifying Dataplanes in Software Switches and Routers

Although S^{2}E is mostly used to *test* software, it can also *verify* it. In testing, one tries to exercise as many paths as possible while checking for properties along each of them, hoping to find one or more paths with a violation. Enumerating paths can show the presence of bugs, but does not prove their absence. Proving that certain properties hold for a program implies enumerating all possible execution paths. Even though this is often impossible for large programs, it can be feasible for small units. Once a property is verified for all individual units, it may be possible to show that the composition of the units still satisfies that property.

Dobrescu et al. have built a tool that uses S^{2}E in order to verify properties of software router pipelines [47]. In their work, they break down the pipeline into separate smaller elements that can be verified individually by exhaustively exploring all their execution paths. For example, if a pipeline has an IP option parsing module followed by a module that handles NAT, showing that the pipeline cannot crash reduces to showing that each element taken in isolation (and explored with arbitrary inputs) cannot crash.

This tool defines a programming model that dictates how the elements of the pipeline can manipulate the state in order to allow proof composability. The model defines three types of states that the packet processing elements can manipulate: packet state, private state, and static state. The packet state is owned by the element currently processing the packet. When processing completes, ownership is transferred to the next pipeline element. Private state of an element cannot be accessed by any other element and static state can only be read by all elements (it is immutable).

The tool adds further restrictions on loops and data structures to improve scalability. Loop iterations are viewed as "mini-pipelines" that can be symbolically executed separately. As such, loops can only access the packet state. State that is shared across iterations must therefore be part of the packet state. Finally, data structures need to follow a key/value-store interface and their implementation is verified separately.

The tool successfully verified crash freedom and bounded execution time of an edge router, a core router, and a network gateway within minutes. It also found bugs in several Click [73] elements, such as incorrect processing of IP options when fragmenting packets or an assertion in the NAT element that can be triggered by incoming packets.

6.4 Performance of the S^{2}E Prototype

S^{2}E introduces approximately 3× runtime overhead over vanilla QEMU when running in concrete mode, and 78× in symbolic mode. Concrete-mode overhead is mainly due to checks for accesses to symbolic memory, while symbolic-mode overhead is due to LLVM interpretation and constraint solving. S^{2}E incurs these overheads along each execution path both in single and multi-core mode.

The overhead of symbolic execution is mitigated in practice by the fact that the symbolic domain is much smaller than the concrete domain. For instance, in the `ping` experiments (§6.2), S^{2}E executed 30,000× more x86 instructions concretely than it did symbolically. All the OS code (e.g., page fault handler, timer interrupt, system calls) that is called frequently, as well as all the software that is running on top (e.g., services and daemons) run in concrete mode. Moreover, S^{2}E distinguishes inside the symbolic domain instructions that can execute concretely (e.g., that do not touch symbolic data) and runs them natively. `ping`'s four orders of magnitude difference between the number of concretely vs. symbolically running instructions is a *lower* bound on the amount

of savings that selective symbolic execution brings over classic symbolic execution: by executing concretely those paths that would otherwise run symbolically, S^{2}E *also* saves the overhead of further forking paths that are ultimately not of interest (e.g., on branches in the concrete domain).

Another source of overhead are symbolic pointers. We compared the performance of symbolically executing the `unlink` UNIX utility's x86 binary in S^{2}E on a single core vs. symbolically executing its LLVM version in KLEE. Since KLEE recognizes all memory allocations done by the program, it can pass to the constraint solver memory arrays of exactly the right size; in contrast, S^{2}E must pass entire memory pages. In 1 hour, using one core, with a 256-byte page size, S^{2}E explored 7,082 paths, compared to 7,886 paths in KLEE. Average solving time was 0.06 sec for both. With 4 KB pages, though, S^{2}E explored only 2,000 paths, averaging 0.15 sec per constraint.

We plan to reduce the overhead in concrete and symbolic modes in several ways, which are further described in chapter 8.

6.5 Trade-Offs in Using Execution Consistency Models

Having seen the ability of S^{2}E to serve as a platform for building powerful analysis tools, we now experimentally evaluate the trade-offs involved in the use of different execution consistency models. In particular, we measure how total running time, memory usage, and path coverage efficiency are influenced by the choice of models. We illustrate the trade-offs using both kernel-mode binaries (the SMSC 91C111 and AMD PCnet network drivers) and a user-mode binary (the interpreter for the Lua embedded scripting language [81]). The 91C111 closed-source driver binary has 19 KB, PCnet has 35 KB; the symbolic domain consists of the driver, and the concrete domain is everything else. Lua has 12.7 KLOC; the concrete domain consists of the lexer and parser (2 KLOC) and the environment, while the symbolic domain is the remaining code of the interpreter. Parsers are the bane of symbolic execution engines, because they have many possible execution paths, of which only a small fraction are paths that pass the parsing/lexing stage [59]. The ease of separating the Lua interpreter from its parser and lexer in S^{2}E without touching the Lua source code illustrates the benefit of selective symbolic execution.

We use a script in the guest OS to call the entry points of the drivers. Execution proceeds until all paths have reached the driver's `unload` method. We configure a selector plugin to exercise the entry points one by one. If S^{2}E has not discovered any new basic block for some time, this plugin kills all paths but one. The plugin chooses the remaining path such that execution can proceed to the driver's next entry point. We use the same settings as in §6.1 and vary the consistency model.

Without path killing, drivers could get stuck in the early initialization phase, because of path explosion (e.g., the tree rooted at the initialization entry point may have several thousand paths when its exploration completes). The selector plugin also kills redundant subtrees when entry

points return, because calling the next entry point in the context of *each* of these execution states (subtree leaves) would mostly exercise the same paths over again.

For Lua, we provide a symbolic string as the program input, under SC-SE consistency. In SC-UE mode, the input is symbolic like in SC-SE, but all symbolic data is concretized when accessed by code outside of the symbolic domain (i.e., outside of the Lua execution engine). Under local consistency, the input is concrete, and we insert suitably constrained symbolic Lua opcodes *after* the parser stage. Finally, in RC-OC mode, we make the Lua opcodes completely unconstrained.

In this section, we run S^2E in single-core mode and average results over 10 runs for each consistency model. Running S^2E in multi-core mode would introduce additional randomness to the results, making it difficult to compare different data points—since the state selection is local to each S^2E process, some of the processes may end up executing states that would never be selected if the state selection was global. Whether this happens or not depends on the initial distribution of the states between S^2E processes, which is difficult to predict.

Generally speaking, weaker (more relaxed) consistency models help achieve higher basic-block coverage in a given amount of time—Figure 6.12 shows results for the running times from Table 6.8. For PCnet, coverage varies between 14% and 65%, while 91C111 ranges from 10% to 84%. The stricter the model, the fewer sources of symbolic values, hence the fewer explorable paths and discoverable basic blocks in a given amount of time. For the Windows drivers, system-level strict consistency (SC-SE) keeps all registry inputs concrete, which prevents several configuration-dependent blocks from being explored. In SC-UE, concretizing symbolic inputs to arbitrary values prevents the driver from loading and prevents Lua from executing a meaningful command, thus yielding poor coverage and short running time.

Consistency	**91C111 Driver**	**PCnet Driver**	**Lua**
RC-OC	6 min	9 min	32 min
RC-LC	10 min	13 min	31 min
SC-SE	5 min	9 min	33 min
SC-UE	<1 min	<1 min	<1 min

Table 6.8 – Time to complete exploration of two device drivers and the Lua interpreter under different consistency models.

In the case of Lua, the local consistency model allows bypassing the lexer component, which is especially difficult to symbolically execute due to its loops and complex string manipulations. RC-OC exceptionally yielded less coverage because execution got stuck in complex crash paths reached due to incorrect Lua opcodes and their operands (such opcodes could never reach the parser during normal execution, hence the parser does not check for them but instead continues erroneous execution for some time, leading to multiple forks before it finally crashes).

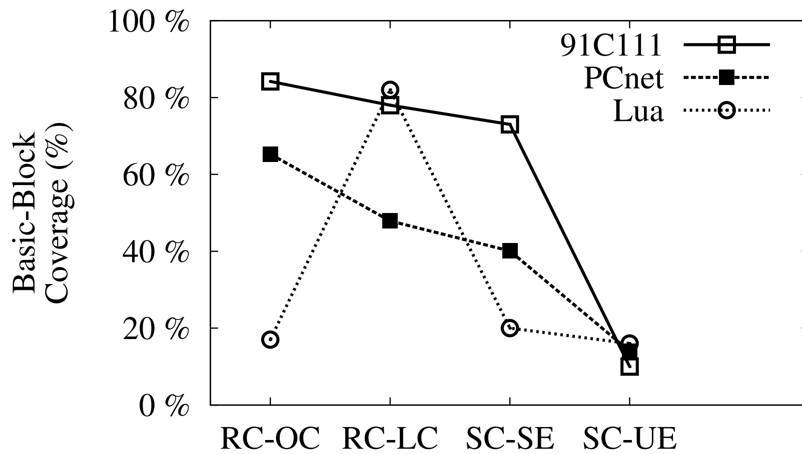

Figure 6.12 – Effects of memory consistency models on coverage

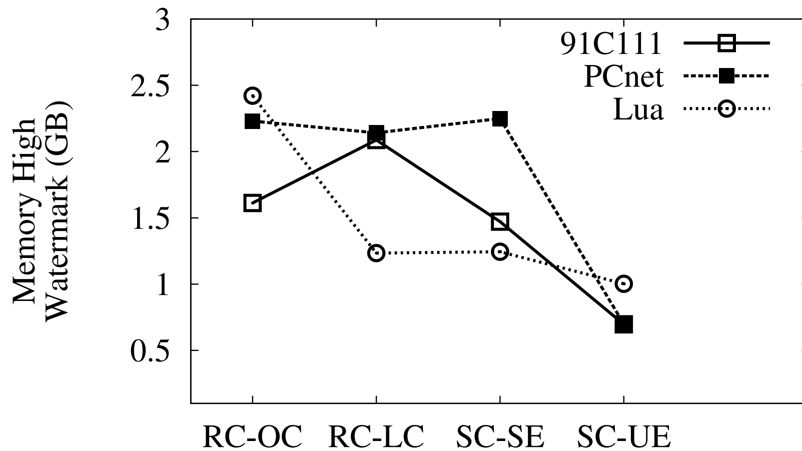

Figure 6.13 – Effects of memory consistency models on memory usage

Path selection together with adequate consistency models reduce memory usage (Figure 6.13). Stricter models generate fewer execution paths to explore, which in principle should reduce memory consumption. However, in practice memory usage is more strongly correlated with running time. For example, in RC-OC and SC, it takes roughly 6 minutes to complete the execution of the 91C111 driver, taking 1.5 GB of memory for 1,950 and 620 paths respectively. RC-LC takes longer, using 2 GB of memory for 955 paths. The reason is that, the longer the paths run, the bigger the corresponding program states grow, due to copy-on-write effects: various OS components have more time to write into more memory pages, yielding higher per-state memory consumption.

Finally, consistency models affect constraint solving time (Figures 6.14 and 6.15). The relationship between consistency model and constraint solving time often depends on the structure of the system being analyzed—at a first level of approximation, the deeper a path, the more complex

the corresponding path constraints. The fraction of execution time spent in the constraint solver decreases with stricter execution consistency models, because stricter models restrict the amount of symbolic data, generating fewer queries. Note that analyzing Lua under RC-OC exceptionally yielded a low fraction of time spent in the constraint solver for the same reason it got poor coverage: execution could not reach the more complex parts of the interpreter.

We observe that, except for SC-UE, the average time spent to solve a query remains roughly constant across consistency models. This is in contrast to our earlier results [34], where S^2E used an older version of STP (revision #943 vs. #1432 in this thesis). That older version took more time to solve queries generated by weaker consistency models. We plan to investigate this behavior in future work. SC-UE concretizes symbolic values early, which strongly reduces the number and complexity of solver queries and makes them quicker to solve on average.

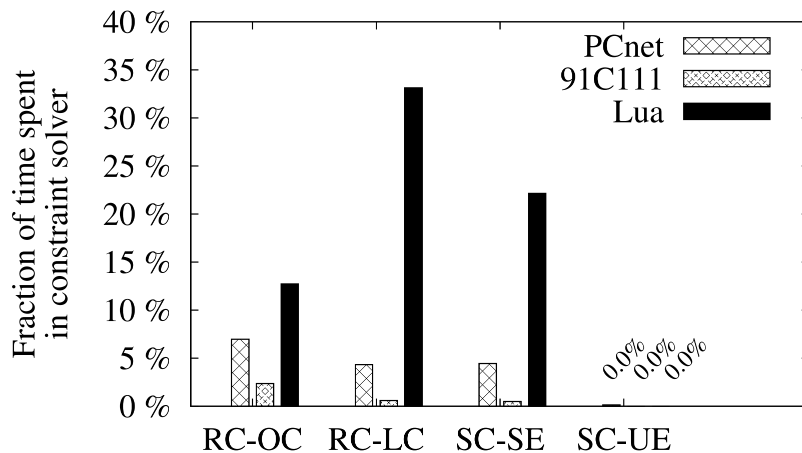

Figure 6.14 – Impact of consistency models on the time spent in the constraint solver.

We attempted to run Lua in KLEE in order to compare the results for different execution consistency models with those obtained in S^2E. We expected that the Lua interpreter, being completely in user-mode and not having any complex interactions with the environment, could be handled by KLEE. However, KLEE does not model some of its operations. For example, the Lua interpreter makes use of `setjmp` and `longjmp` instructions, which turn into `libc` calls that manipulate the program counter and other registers in a way that confuses KLEE. Unlike S^2E, other analysis engines do not have a unified representation of the hardware, so all these details must be explicitly coded for (in KLEE's case, detect that `setjmp` / `longjmp` is used and ensure the execution state is appropriately adjusted). In S^2E, this comes "for free" because the CPU registers, memory, and I/O devices are shared between the concrete and symbolic domain.

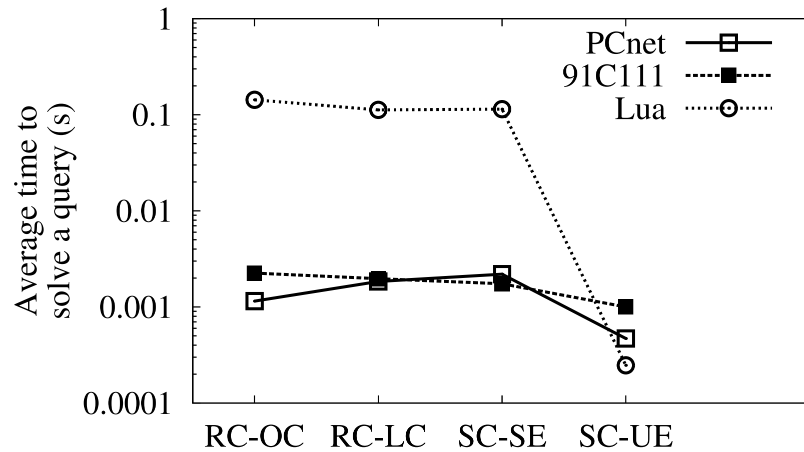

Figure 6.15 – Impact of consistency models on the average time to solve a query.

6.6 Summary

Our evaluation shows that S^2E is a general platform that can be used to write diverse and interesting system analyses—we illustrated this by building, with little effort, tools for bug finding, reverse engineering, and comprehensive performance profiling. Consistency models offer flexible trade-offs between the performance, completeness, and soundness of analysis. By employing selective symbolic execution and relaxed execution consistency models, S^2E is able to scale these analyses to large systems, such as an entire Windows stack. Analyzing real-world programs like Apache httpd, Microsoft IIS, and `ping` takes a few minutes up to a few hours, in which S^2E explores hundreds of thousands of paths through the binaries.

Chapter 7

Related Work

In this chapter, we categorize the existing analysis techniques and tools in terms of the trade-offs they make. Analysis tools can be classified in terms of their accuracy and performance, whether they operate on source code or binaries, and at which level of software stack they operate. We explore the benefits and drawbacks of each design choice, and the challenges involved. Finally, we show how S^2E differs from existing methods by allowing flexible choices of performance, accuracy, and software stack level for building new analysis tools.

7.1 Accuracy vs. Performance in Analysis Tools

In general, analysis tools must make trade-offs between accuracy and performance. A tool is 100% accurate when it does not have false positives or false negatives, i.e., every defect that it reports actually exists and it does not miss any defect. Performance denotes the scalability of the tool, measured by the number of execution paths the tool can analyze and the size of the system it can handle (e.g., an isolated program or an entire software stack).

Certain types of simpler analysis allow for accurate and scalable tools. For example, a compiler can be seen as a tool that checks that the input program is syntactically valid. If the compiler is correct, it is both accurate and scalable. An inaccurate compiler would emit an error for syntactically correct files (false positives) or miss syntax errors (false negatives), and would take a very long time on large programs if it did not scale.

Verification tools are accurate but typically do not scale to large software. Such tools must *prove* that the programs they analyze have given properties, possibly subject to assumptions. Isabelle [62] is an interactive theorem prover that provided the proof of functional correctness of the seL4 [72] kernel, which consists of ~10 KLOC of C code. The proof consisted of 200 KLOC of Isabelle script and it took 20 person-years to develop the machinery to generate and verify it. It was restricted to a subset of C99 and a simple machine model, omitted small portions written in

assembly code, and assumed a correct C compiler. Other techniques based on state exploration, such as symbolic execution [69] or model checking [11], can also be used to prove properties, but suffer from path explosion and/or require significant manual help.

Static analysis tools can trade accuracy for scalability. Coverity analyzes millions of lines of codes in minutes [12], at the expense of false negatives and false positives. A high rate of false positives can overwhelm users with many false bugs. Saturn [46] and `bddbddb` [76] prove the presence or absence of bugs using a path-sensitive analysis engine to decrease the number of false positives. Saturn uses function summaries to scale to larger programs.

Dynamic analysis tools can analyze large systems without false positives but suffer from false negatives in practice because of the path explosion problem. For example, Valgrind [123], AddressSanitizer [113], SAGE [59], DART [55], or BitBlaze [117] have no false positives and can analyze large software, up to an entire VM for BitBlaze. However, they cannot report all the bugs they are designed to find, because they can only explore a tiny fraction of the program state space in a finite amount of time. Tools based on symbolic execution can explore a larger state space in the same amount of time than those based on random test generation, but are still limited by the exponential nature of the state space. State merging techniques allow to exponentially reduce the state space by combining together similar states. This considerably reduces false negatives because a much larger state space can be explored in the same amount of time. State merging allows tools like Cloud9 [75] to achieve a speed up of up to 11 orders of magnitude. A related approach to cope with path explosion is compositional symbolic execution [57]. The results of a frequently-called function can be summarized in a formula that can be used next time the program calls the function, avoiding the cost of re-executing that function.

Another way to tackle the path explosion problem and improve accuracy is to use environment models. Programs interact with their environment, which consists of all the elements that the program needs in order to run (e.g., file system, network stack, other processes, etc.). The environment is often considerably larger than the program itself (e.g., the file system in case of programs that access files). A model of the environment abstracts away the complex implementation details of the actual environment, while preserving sufficient functionality to allow analyzing the program. File system models have allowed KLEE to test UNIX utilities without involving the real filesystem [27]. However, based on our own experience, writing models is a labor-intensive and error-prone undertaking. Other researchers report that writing a model for the kernel/driver interface of a modern OS took several person-years [5]. Of course, the advantage of using models is generally faster analyses.

Another approach is to relax execution consistency. A consistent execution is one in which there exists a feasible execution path from the system's start state to the system's current state (see §2.3). For example, fault injection tools [83] relax the consistency by replacing the original

function calls (e.g., `malloc`) with a realistic error value (e.g., `NULL`). On the one hand, execution becomes inconsistent because there may not exist a path in the real system where the functions would be replaced in this way. On the other hand, this speeds up the analysis by avoiding exploring all the paths through the system hoping to find one that really fails (e.g., a path where `malloc` returns `NULL` due to memory shortage), without necessarily decreasing the accuracy of the analysis (e.g., programs most likely do not check why `malloc` actually failed).

Other approaches can include the environment directly in symbolic analysis by executing the environment concretely, with various levels of consistency that were appropriate for the specific analysis in question, most commonly bug finding. For instance, CUTE [112] can run environment code consistently without modeling, but it is limited to strict consistency and code-based selection. SJPF [102] can switch from concrete to symbolic execution, but does not track constraints when switching back, so it cannot preserve consistency in the general case. Non VM-based approaches, in general, cannot control the environment outside the analyzed program. For instance, both KLEE and EXE allow a symbolically executing program to call into the concrete domain (e.g., perform a system call), but they cannot fork the global system state. As a result, different paths clobber each other's concrete domain, with unpredictable consequences. Concolic execution [55, 111] runs everything concretely and scales to full systems (and is not affected by state clobbering), but may result in lost paths when execution crosses program boundaries. CUTE, KLEE, and other similar tools cannot track the branch conditions following calls into concrete code (unlike S^2E), and thus cannot determine how to redo calls in order to enable overconstrained but feasible paths.

7.2 Source Code vs. Binary Analysis

Concrete implementations of analysis techniques operate either on source code or program binaries. In this section, we explore the trade-offs involved in choosing between source code and binary form as well as implementation challenges that the analysis techniques face when they need to operate on either of these two representations.

7.2.1 Source Code Analysis

Symbolic execution can be directly applied to source code. EXE [28], DART [55], and CUTE [112] use CIL [96], a source-to-source C compiler. The source-to-source compiler takes the source code and annotates all assignments in order to track the propagation of the symbolic values through the program. CIL also annotates branches, in order to determine which outcome to follow in case of symbolic predicates. The output is a rewritten C program that can be compiled and run natively. KLEE [27] is another symbolic execution tool that operates on the LLVM representation of the

source code. Unlike CIL-based approaches, KLEE implements symbolic execution by interpreting LLVM instructions. Java PathFinder [63] uses a similar approach on Java byte code.

Model checkers transform the source code into a large boolean formula in order to verify different properties. This formula encodes the program and the desired properties. A theorem prover [52] can be used to ensure the formula is valid for all inputs, thus proving the property holds. In case of recursive functions or loops, the formulas can be large or even unbounded. Bounded model checkers such as CALYSTO [3] or LLBMC [115] transform the source to LLVM and use custom LLVM analysis passes that unroll loops and recursive functions up to a certain depth in order to produce bounded formulas that can be handled by the theorem prover.

In addition to intermediate representations such as LLVM, static analysis tools can also operate on the source code's abstract syntax trees (AST). The AST captures all information present in the source code such as variable names and types, line numbers, etc. This information can be lost in lower-level representations. Compilers like `clang` [39] or `gcc` analyze the AST to warn developers about potential errors, unsafe coding styles, etc.

Source-based approaches face several challenges. They must be able to read the source code, it is difficult for them to handle inline assembly, they cannot analyze third-party components for which there is no source code, and implementing source-based tools is often an ad-hoc, labor-intensive effort. These challenges affect the accuracy and scalability of source-based tools.

Reading the source code requires a parser that can handle many flavors of the language. For example, Coverity [12] has to deal with dozens of variants of C. Inline assembly is usually skipped, potentially missing errors or reporting false alarms [12, 27]. Third-party components must be abstracted away or modeled [27, 21, 2, 13, 3, 46]. Generic analysis algorithms like symbolic execution are usually rewritten from scratch for every language [27, 29]. Overall, source-based tools require a large error-prone engineering effort in order to be applied at scale and to produce acceptable results.

7.2.2 Binary Analysis

Unlike source-based tools, binary analysis tools only require the machine code in order to run, which has several advantages. First, it allows analyzing the actual code that runs on the hardware, which may uncover problems missed by source-based tools. A source code analysis tool may declare a program safe, but compile-time optimizations may introduce subtle bugs [125]. Second, binary analysis helps when the program is built from different modules written in different languages (e.g, a program in C++ calling optimized library routines written in assembly). Third, no reliance on source code allows analyzing third-party proprietary software, which can be useful for security audits.

Binary analysis tools use a wide range of instrumentation techniques in order to observe and possibly modify the behavior of the program under analysis. Common techniques include system and library call interception (e.g, DTrace [48]), shim libraries (e.g., `LD_PRELOAD`), disassembly, and binary translation.

Tools that need to reason about binary code first disassemble it and optionally transform it to an intermediate representation (IR), depending on the needs of the analysis. Simple disassembly may be enough to extract an approximate control flow graph [60], while more advanced binary translation techniques may be used for more sophisticated static and dynamic analysis.

Dynamic binary translation (DBT) is suitable for run-time analysis. A dynamic binary translator continuously fetches blocks of binary code, disassembles and translates them to IR, instruments the IR, and compiles them back to native code. Many tools use this instrumentation capability to find memory errors, resource leaks, do performance profiling, taint analysis, symbolic execution, etc. DBT offers flexible instrumentation capabilities: tools can limit instrumentation to fragments of code of interest to reduce overhead, passively observe the execution to gather information (e.g., count instructions or cache misses) or actively modify program or system state. Pin [82] and PinOS [24] are two examples of such analysis platforms that expose DBT to tools built on top of them. Other tools, such as Mayem [31] or BitBlaze [117] use DBT to implement symbolic execution at the program and system level.

Static translators are suitable for static analyses that need to operate on the entire binary at once. They take as input a binary in machine code and output a different binary in an IR form, with varying accuracy. Accuracy is dictated by the disassembly technique they use. Linear sweep (e.g., used by BAP [17]) disassembles the binary sequentially and is oblivious to code mixed with data. Recursive disassembly analyzes call and jump targets to discover new code. It is more accurate because it is better at avoiding disassembling data. To improve accuracy further, different heuristics can be used [60], as well as more sophisticated and expensive analysis, such as abstract interpretation [68]. As we shall see next, accuracy has various impacts on the analysis tools.

Both static and dynamic translation have limitations. Static translators (and disassemblers in general) need to distinguish code from data, handle variable instruction sizes, indirect control flow, position-independent code, obfuscated binaries, the absence of high-level data types and variable information, etc. Dynamic disassemblers are not affected by most of these problems, because they execute the translated code, resolving all the ambiguities encountered by static translators. However, this comes at the cost of a limited view of the RC-CC exposed to the analysis tools.

These limitations can decrease the accuracy of the tools built on top of binary translators. Incorrect lifting of machine code to IR can cause otherwise sound and complete analysis techniques to miss bugs (false negatives), or report erroneous faults (false positives). The partial view of the code provided by DBT usually leads to false negatives.

7.3 Choosing the Software Stack Level

Analysis tools can operate at various levels of the software stack. They can handle user space software and libraries [123, 113, 114], interpreted [29] or compiled programs [27, 82, 123, 24], work on kernel-mode drivers [5], interact with hardware [132], or even handle the entire software stack [117]. The level at which tools operate in the software stack and the assumptions they make about the analyzed program affect their accuracy and scalability.

Choosing one particular level of the stack forces the analysis tools to make assumptions about the environment of the program. Verification tools usually assume a bug-free runtime or compiler, which may potentially render correctness proofs irrelevant should this assumption be violated. It is hard for symbolic execution tools for user-space programs to track the side effects of system calls, compromising accuracy [28, 27, 102]. They cope with this be modeling the environment, hoping that the model is accurate enough for the analyzed program, or by using VM-based approaches. VM-based approaches [117] have a complete view of the system, but cannot control the outside environment (e.g., interaction with external network nodes or hardware devices).

A particular level in the stack may also force particular analysis and instrumentation techniques on the tool. For example, an analysis tool that requires kernel instrumentation might be constrained to use a kernel-mode DBT engine [50], whose implementation is very different from that of a user-mode DBT [82]. Tools for taint analysis of malware typically need to operate at the VM level, because tainted data can flow across the entire system [117].

7.4 S^2E in the Analysis Tools Design Space

S^2E is a platform that enables multi-path analysis of binary software stacks with flexible accuracy/performance trade-offs. S^2E operates on binaries using VM-based dynamic binary translation. This allows S^2E users to build tools that can operate at any level of the software stack, eliminates most of the problems faced by source-based analysis tools, and does not restrict the type of analyses that can be performed. S^2E enables flexible accuracy/performance trade-offs by introducing several execution consistency models. In this section, we compare S^2E to existing platforms.

BitBlaze [117] is the closest dynamic analysis framework to S^2E. It combines virtualization and symbolic execution for malware analysis and offers a form of local consistency to introduce symbolic values into API calls. In contrast, S^2E has several additional consistency models and various generic path selectors that trade accuracy for exponentially improved performance in more flexible ways. To our knowledge, S^2E is the first to handle all aspects of hardware communication, which consists of I/O, MMIO, DMA, and interrupts. This enables symbolic execution across the entire software stack, down to hardware, resulting in richer analyses.

S^2E can add multi-path analysis abilities to any single-path dynamic tool, while not limiting the types of analysis. PTLsim [132] is a VM-based cycle-accurate x86 simulator that selectively limits profiling to user-specified code ranges to improve scalability. Valgrind [123] is a framework best known for cache profilers, memory leak detectors, and call graph generators. PinOS [25] can instrument OSes and unify user/kernel-mode tracers. PinOS relies on Xen and a paravirtualized guest OS, unlike S^2E. SimOS [106] is a machine simulator that provides interchangeable simulation models, allowing flexible simulation speed and simulation details trade-offs. PTLsim, PinOS, and Valgrind implement cache simulators that model multi-level data and code cache hierarchies. S^2E allowed us to implement an equivalent *multi-path* simulator with little effort (see §6.2).

S^2E complements classic single-path, non VM-based profiling and tracing tools. DTrace [48] is a framework for troubleshooting kernels and applications on production systems in real time. DTrace and other techniques for efficient profiling, such as continuous profiling [1], sampling-based profiling [26], and data type profiling [99], trade accuracy for low overhead. They are useful in settings where the overhead of precise instrumentation is prohibitive. Other projects have also leveraged virtualization to achieve goals that were previously prohibitively expensive. These tools could be improved with S^2E by allowing the analyses to be exposed to multi-path executions.

S^2E uses mixed-mode execution as an optimization to increase efficiency. Mixed mode interprets only instructions that touch symbolic data, while running the rest natively. This idea first appeared in DART [55], CUTE [112], and EXE [28], and later in Bitscope [16]. However, automatic bidirectional data conversions across the symbolic-concrete boundary did not exist previously, and they are key to S^2E's scalability (see §2.2).

In-situ model checkers [56, 63, 130, 94, 129] can directly check programs written in a common programming language, usually with some simplifications, like data-range reduction, without requiring a model. Since S^2E directly executes the target binary, one could say it is an in-situ tool.

S^2E goes further and provides a consistent separation between the environment (whose symbolic execution is not necessary) and the target code to be tested (which is typically orders of magnitude smaller than the rest). This is what we call “in vivo” in S^2E: analyzing the target code in-situ, while facilitating its consistent interaction with that code's unmodified, real environment. Note that other researchers have used the term “in vivo” in similar contexts as well, but with a different meaning from S^2E's—e.g., [93] propose a technique for testing where “in vivo” stands for executing tests in production environments.

To summarize, S^2E embodies numerous ideas that were fully or partially explored in earlier work. What is unique in S^2E is its generality for writing various analyses, the availability of multiple user-selectable (as well as definable) consistency models, automatic bidirectional conversion of data between the symbolic and concrete domains, and its ability to operate without any modeling or modification of the (concretely running) environment.

Chapter 8

Limitations and Future Work

In this chapter, we summarize the limitations that emerged while using S^2E in practice and present several future directions in which researchers can improve S^2E in order to make it more practical and easier to use. This is important in order to enable a wider adoption.

8.1 Limitations

Despite its strengths, S^2E has three main limitations: path explosion, complexity of execution consistency models, and ease of use of the platform.

S^2E still suffers from path explosion occurring inside the modules of interest. For example, testing a moderately-sized network service with full-sized symbolic packets can still lead to a prohibitive number of paths. This makes it hard for existing heuristics to pick an execution path that would lead execution to deep parts of the program. S^2E also adds a high constant execution overhead for each path, both for concrete execution (10-20$\times$ compared to native execution) and symbolic execution (about two orders of magnitude), limiting the paths that can be explored in a given time budget. Finally, path explosion creates deep paths with complex constraints, which bottlenecks the constraint solver embedded in any symbolic execution engine.

While S^2E allows relaxing execution consistency models in order to alleviate path explosion, doing so may be complex in practice. In section 2.3.2, we explained how a platform developer would implement support for execution consistency models. However, we do not show a systematic way in which a user could use them. In practice, users often need to reason about the unit-environment interface in order to derive annotations that relax the model in the desired way. For example, in order to perform fault injection to test the error recovery code in device drivers [74], users need to understand what error values a kernel API function may return in order to relax the model without introducing false positives. However, it is not clear how to relax the models *automatically* without any knowledge of the interface. We leave this investigation for future work.

This complexity is compounded by the semantic gap [32] between the low level nature of the APIs offered by the S^2E engine and how users reason about code. Users typically reason in terms of programs, libraries, modules, functions, and source code. However, S^2E exposes program counters, translation blocks, CPU registers, and memory accesses. Although S^2E has an extensive plugin library that provides high level abstractions in order to attempt to bridge the semantic gap, it requires complex introspection of the guest OS [53]. In practice, a developer who wants to write an annotation (e.g., in order to relax a consistency model), needs to use a complex API in order to access guest data structures. Because of this, users often resort to writing instrumentation directly in the guest rather than via S^2E plugins [105, 20]. More research is needed in order to find approaches that offer the right trade-offs between ease of use and flexibility.

8.2 Future Work

In this section, we present several ideas to improve the issue of path explosion that we raised earlier. Our goal is to combine all these ideas into one platform in order to increase by at least an order of magnitude the execution speed and the number of paths that can be explored in a given amount of time and resource constraints (memory and CPU). This performance increase must be achieved without sacrificing accuracy. We believe that raw speed will let S^2E users perform more complex analysis while decreasing the need for aggressively relaxed execution consistency models, thereby improving the ease of use.

Path explosion can be mitigated with several new elements: *dynamic state merging*, *parallel symbolic execution*, *incremental constraint solving service*, combining S^2E with *static analysis*, *fast symbolic interpreter*, and *hardware virtualization*.

Dynamic state merging has been shown to yield up to 11 orders of magnitude improvement over traditional symbolic execution [75]. It exponentially reduces the state space by merging similar states at run-time. It also applies a static analysis on the program's source code in order to determine if the merge will excessively impact the constraint solver. Adapting dynamic state merging to S2E would require performing the same static analysis on *binaries* and finding efficient ways of merging paths that contain the state of an *entire* system.

Parallel symbolic execution helps cope with path explosion by parallelizing symbolic execution in a way that scales well on large clusters of cheap commodity hardware. Existing techniques [21] scale linearly with the number of nodes in the system, thus enabling us to "throw hardware at the problem". We plan to apply these same principles to a parallel version of S^2E that can efficiently combine the resources of large clusters.

Incremental constraint solving will reduce the fraction of the time S^2E spends in the constraint solver. S^2E currently uses the STP solver [52]. On each invocation, STP considers all the

constraints accumulated so far to produce a result, instead of leveraging previous results and exploiting the incremental nature of additional queries to speed up solving. We plan to extend S^2E to use in parallel several incremental solvers, such as Z3 [45] and Boolector [19]. Different solvers can be faster or slower for a given constraint, and the portfolio approach can take the results that arrive the soonest, thus operating at the speed of the fastest solver (at the cost of extra hardware). This portfolio of solvers could be further decoupled from S^2E and exposed as a service, in order for S^2E to benefit more easily from advances in the constraint solving field. We present in our recent work [22] an early prototype that efficiently implements incremental constraint solving in the context of multi-path exploration.

Bringing *static analysis* to S^2E would enable more effective heuristics. It could, for example, determine the location of all loops and help heuristics to pick paths that exit them early [105]. Building RevGen (Chapter 5) was the first step towards achieving this goal.

A *fast symbolic interpreter* would reduce time spent interpreting machine instructions in symbolic mode. S^2E currently translates machine code to LLVM in order to run it in the KLEE interpreter. The LLVM translation phase is not only ~40× slower compared to translating code to QEMU's native targets, but the resulting LLVM code is also unnecessarily verbose and slow to interpret. We envision replacing the LLVM translator with QEMU's built-in TCI back-end, a light-weight backend that is optimized for quick interpretation of machine code. Of course LLVM would still be available for tools that require deeper analysis of the guest code.

Finally, *hardware virtualization [122]* would bring hardware speeds to symbolic execution. S^2E relies on a DBT to instrument guest code and transform it into a format suitable for symbolic interpretation. However, most of the system not only runs in concrete mode, but is also never instrumented. It is therefore possible to let it run natively, cutting the dynamic binary translation phase. A major challenge will be to efficiently switch back and forth between hardware virtualization, DBT mode, and symbolic interpretation.

Chapter 9

Conclusion

This thesis presented S^{2}E, a new platform for *in-vivo multi-path analysis* of systems, which scales to large, proprietary, real-world software stacks, like Microsoft Windows. It is the first time virtualization, dynamic binary translation, and symbolic execution are combined for the purpose of generic behavior analysis. S^{2}E simultaneously analyzes entire *families of paths*, operates directly on *binaries*, and operates *in vivo*, i.e., includes in its analyses the entire software stack: user programs, libraries, kernel, drivers, and hardware. S^{2}E uses automatic bidirectional symbolic–concrete data conversions and relaxed execution consistency models to achieve scalability.

S^{2}E's scales to large systems using two new ideas: *selective symbolic execution* and *execution consistency models*. Selectivity limits multi-path exploration to the module of interest (e.g., a library) to minimize the amount of symbolically-executed code, which avoids path explosion outside of that module. Execution consistency models allow making principled performance/accuracy trade-offs during analysis.

This thesis showed that S^{2}E enables rapid prototyping of a variety of system behavior analysis tools with little effort. We built an in-vivo multi-path performance profiler (PROF$_S$) and a system for semi-automatically reverse engineering binary device drivers (RevNIC). PROF$_S$ allows predicting the performance for certain classes of inputs, using metrics such as instruction count or cache misses. RevNIC analyzes closed-source device drivers to synthesize new, safer, and portable drivers for different OSes and architectures.

The S^{2}E platform is open sourced and available at `http://s2e.epfl.ch`, with a ready-to-use demo, documentation, and tutorials. Three years after release, S^{2}E acquired a rapidly growing user community of more than 150 members and is actively used by researchers and companies around the world in order to test distributed networks, analyze file systems, detect private data leaks in smartphone apps, perform security analysis, and more.

Appendices

Appendix A

Tutorial: Using the S^2E API to Build the *Annotations* Plugin

In section 4.2.1, we gave an overview of how to build the *Annotations* plugin. An S^2E annotation is a piece of code written by an S^2E user in order to observe and manipulate execution states. The *Annotations* plugin can be used to implement different execution consistency models and is a central piece in tools like DDT$^+$ (§6.3.1) and REV$^+$ (§6.1).

Given its wide use, the *Annotations* plugin must be generic and work on any piece of code, no matter what guest OS is running. *Annotations* implements only the mechanisms that let users specify the desired annotations and relies on other plugins for unrelated tasks, such as OS monitoring.

We show here in detail how to build a plugin that monitors the guest OS and notifies other plugins when programs, drivers, libraries, or any kind of modules are loaded (§A.1), how to use the information about loaded modules to detect the execution of a specific module (§A.2), how to track function calls in those modules (§A.3), and finally, how to let users annotate the desired code and make sure the annotations are executed at the right moment (§A.4). Figure A.1 summarizes the relationship between these plugins and Figure A.2 shows the corresponding S^2E configuration file that we use throughout the remainder of this section as a running example. This example shows how users can configure the *Annotations* plugin in order to insert symbolic data in network packets during the analysis of the `rtl8029.sys` network device driver, that is part of Windows XP.

A.1 Monitoring Module Loads

S^2E requires a specific monitoring plugin for each OS in order to track OS-level events, such as module loads and unloads. Tracking these events in a system may be difficult and platform-specific. For example, getting the process identifier of the currently executing process requires

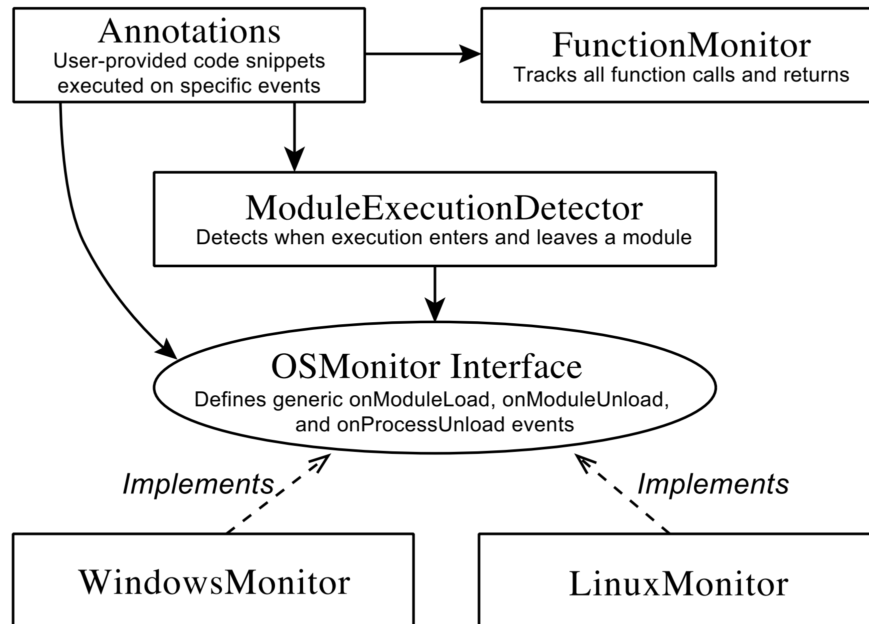

Figure A.1 – Dependencies of the *Annotations* plugin. *Annotations* requires *ModuleExecutionDetector*, *FunctionMonitor*, and a plugin that implements the *OSMonitor* interface (such as *WindowsMonitor* or *LinuxMonitor*). This is a more detailed view of Figure 4.1.

parsing OS-specific data structures in the guest's kernel heap. Moreover, the exact layout of these structures varies for different versions of the same operating system. Such implementation details must be hidden behind a generic interface.

S²E plugins that monitor OS-level events must implement a generic interface in order to be used interchangeably by client plugins. For example, a plugin such as *Annotations* (§A.4) does not care about whether the guest is running on Microsoft Windows or Linux. Such a plugin only needs to know when the OS loads a specified module in order to activate annotations. For this, *Annotations* relies on the generic interface exposed by the underlying OS monitoring plugin (e.g., *WindowsMonitor* or *LinuxMonitor*).

S²E provides the *OSMonitor* interface, which OS monitoring plugins implement. *OSMonitor* defines the *onModuleLoad*, *onModuleUnload*, and *onProcessUnload* events. An OS monitoring plugin triggers *onModuleLoad* (respectively *onModuleUnload*) when a module is loaded (respectively unloaded) and passes the name, size, load address, and address space identifier to the callback functions. The plugin triggers *onProcessUnload* when the OS frees the address space of a process. There is no corresponding *onProcessLoad* event, because the first *onModuleLoad* implicitly defines the new address space.

Consider *WindowsMonitor*, a plugin that implements the *OSMonitor* interface for Microsoft Windows. Detecting driver loads on Windows XP SP3 involves catching the execution of the instruction located at address `0x805A399A` in kernel space. When execution reaches this address,

```
pluginsConfig.WindowsMonitor = {
    version="XPSP3",
    userMode=false,
    kernelMode=true
}

pluginsConfig.ModuleExecutionDetector = {
    rtl8029_sys_1 = {
        moduleName = "rtl8029.sys",
        kernelMode = true
    }
}
```

```
pluginsConfig.FunctionMonitor = { }

pluginsConfig.Annotation = {
  my_annotation = {
    module="rtl8029_sys_1",
    address=0x1233a,
    paramcount=4,
    callAnnotation="rtl8029_copyup_packet"
  }
}
```

```
function rtl8029_copyup_packet(state, pluginState)
   buffer = state:readParameter(1);
   length = state:readParameter(3);

   for i = 0, length - 1, 1 do
        state:writeMemorySymb("copyup_buffer", buffer + i, 1);
   end

   state:writeRegister("eax", 1);
   pluginState:setSkip(true);
end
```

Figure A.2 – Combining S^{2}E plugins to inject symbolic network packets in the `rtl8029.sys` driver. This S^{2}E configuration file is written in the Lua scripting language.

WindowsMonitor parses the driver descriptor located on the stack, extracts the name, load address, and size of the driver, then triggers the *onModuleLoad* event. Subscribers are notified of the driver load and can perform actions accordingly, e.g., detect when execution enters a particular module (see §A.2).

Say the S^{2}E user wants to analyze the `rtl8029.sys` driver running on Windows XP SP3. Since device drivers run in kernel mode, *WindowsMonitor* must be configured to instrument kernel module loads and unloads. This requires five lines of configuration, as shown in Figure A.3.

```
pluginsConfig.WindowsMonitor = {
    version="XPSP3",
    userMode=false,
    kernelMode=true
}
```

Figure A.3 – Configuring *WindowsMonitor* to track driver load/unload events on Microsoft Windows.

A.2 Tracking Module Execution with *ModuleExecutionDetector*

The *ModuleExecutionDetector* plugin publishes two main events: *onModuleInstrTranslation* and *onModuleTransition*. *onModuleInstrTranslation* forwards all the *onInstrTranslation* core events that are triggered inside the modules of interest. *onModuleTransition* notifies its clients whenever execution enters or leaves modules of interest.

Subscribers can use these two events in several ways. For example, *CodeSelector* subscribes to *onModuleTransition* to be notified of when execution enters or leaves a module of interest in order to toggle symbolic execution. *onModuleInstrTranslation* is used by *InstructionCounter*, which relies on this event to register a callback that S^{2}E will call for each instruction executed by the module. The callback increments an instruction counter and periodically writes its value to a log file.

ModuleExecutionDetector relies on a plugin that implements the *OSMonitor* interface. When S^{2}E starts, *ModuleExecutionDetector* automatically looks for a plugin that implements the *OSMonitor* interface. It subscribes to the *onModuleLoad*, *onModuleUnload*, and *onProcessUnload* to maintain the current memory map of the system. This map allows efficiently finding the module that owns a particular address and trigger the *onModuleInstrTranslation* as well as *onModuleTransition* when appropriate.

To enable *ModuleExecutionDetector*, the S^{2}E user adds the appropriate section in the S^{2}E configuration script as shown in Figure A.4. This section can go right below the one for *WindowsMonitor* that we have seen previously. *ModuleExecutionDetector*'s configuration section accepts one section per module to be tracked. Each module to track is identified by its name and whether it is a kernel module or not. sections can be named (e.g., `rtl8029_sys_1`) to allow other plugins to refer to them, as we will illustrate later.

```
pluginsConfig.ModuleExecutionDetector = {
    rtl8029_sys_1 = {
        moduleName = "rtl8029.sys",
        kernelMode = true
    }
}
```

Figure A.4 – Configuring *ModuleExecutionDetector* to track the execution of the `rtl8029.sys` driver.

A.3 Monitoring Function Calls with *FunctionMonitor*

The *FunctionMonitor* plugin notifies its subscribers of function calls and returns. When subscribing, a client plugin passes to *FunctionMonitor* the address of the function to monitor, the identifier of the address space to which the function belongs, and an event callback. *FunctionMonitor* invokes the registered callback whenever a function call or return occurs. The address space identifier allows distinguishing functions at the same virtual address but in different processes.

FunctionMonitor tracks pairs of call and return machine instructions. When a call occurs, besides invoking the registered callback, *FunctionMonitor* also stores in a map the association between the current stack pointer, the address space, and the event callback that corresponds to the called function. When a return instruction is about to be executed, *FunctionMonitor* looks up the current stack pointer and address space identifier in the map and invokes the associated callback. Such tracking is required because return instructions do not carry any information about the function to which they belong.

FunctionMonitor subscribes to the *onInstrTranslation* core event in order to mark and intercept all call and return machine instructions. Whenever these marked instructions are executed, S^{2}E triggers the *onInstrExecution* event which invokes the callbacks previously registered by *FunctionMonitor* when processing the *onInstrTranslation* events. These callbacks check whether there are clients of *FunctionMonitor* that registered for the specific function call or return that is being executed and, if yes, invoke the corresponding client's event callback.

FunctionMonitor assumes that the processor's instruction set has explicit call and return instructions, which is the case e.g., of x86 or MIPS. MIPS uses the `jal` (jump and link) instruction for function calls. This instruction jumps to the specified address while saving in the `$ra` register the program counter of the instruction that follows the jump. Since `$ra` holds the current return address by convention, it can be used to detect jumps that use this register to return to the caller.

FunctionMonitor does not have any user-configurable option. Thus, it is enough to write an empty configuration section as shown in Figure A.2.

A.4 Annotating Code with the *Annotations* plugin

The *Annotations* plugin combines *FunctionMonitor* and *ModuleExecutionDetector* to let users annotate not only function calls but also arbitrary machine instructions. The user writes the annotation directly inside the S^{2}E configuration file, using the Lua language.

The *Annotations* plugin has four configurable parameters: the module name (`module`), the address of the function to intercept (`address`), the number of its parameters (`paramcount`), as well as the name of the Lua annotation to invoke (`callAnnotation`). It is also possible to use

`instructionAnnotation` to annotate arbitrary instructions.

In our example, we configure the *Annotations* plugin to annotate the function that copies a data packet from the network card to a buffer allocated by the driver. This function has four parameters and is located at address `0x1233a` relative to the start of the `rtl8029.sys` driver (Figure A.5).

```
pluginsConfig.Annotation = {
  my_annotation = {
    module="rtl8029_sys_1",
    address=0x1233a,
    paramcount=4,
    callAnnotation="rtl8029_copyup_packet"
  }
}
```

Figure A.5 – Configuring the *Annotations* plugin to inject symbolic network packets in the `rtl8029.sys` driver.

The annotation is contained in the `rtl8029 _copyup_packet` Lua function (Figure A.6). All annotations have two parameters: the current execution state and the current plugin state. The execution state object can be manipulated using the *ExecutionState* object's methods. Similarly, the plugin state parameter exposes the API of the *Annotations* plugin, which allows annotations to change the plugin's configuration at run-time.

```
function rtl8029_copyup_packet(state, pluginState)
   buffer = state:readParameter(1);
   length = state:readParameter(3);

   for i = 0, length - 1, 1 do
        state:writeMemorySymb("copyup_buffer", buffer + i, 1);
   end

   state:writeRegister("eax", 1);
   pluginState:setSkip(true);
end
```

Figure A.6 – Example of an annotation written in the Lua language.

Bibliography

[1] Jennifer Anderson, Lance Berc, Jeff Dean, Sanjay Ghemawat, Monika Henzinger, Shun-Tak Leung, Dick Sites, Mark Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Symp. on Operating Systems Principles*, 1997.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, 2011.

[3] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *30thIntl. Conf. on Software Engineering*, 2008.

[4] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. Technical report, 2005.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM EuroSys European Conf. on Computer Systems*, 2006.

[6] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *Intl. Conf. on Computer Aided Verification*, 2010.

[7] Radu Banabic, George Candea, and Rachid Guerraoui. Finding a distributed system's achilles' heel via grammar differencing. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[8] Bjorn Bartels and Sabine Glesner. Formal modeling and verification of low-level software programs. In *Intl. Conf. on Quality Software*, 2010.

[9] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Symp. on Operating Systems Principles*, pages 267–284, Boulder, CO, Oct 1995.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[13] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Intl. Conf. on Programming Language Design and Implem.*, 2003.

[14] Boomerang decompiler. http://boomerang.sourceforge.net/.

[15] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[16] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. BitScope: Automatically dissecting malicious binaries. Technical Report CMU-CS-07-133, Carnegie Mellon University, 2007.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Intl. Conf. on Computer Aided Verification*, July 2011.

[18] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. 2008.

[19] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[20] Stefan Bucur, George Candea, and Johannes Kinder. Prototyping symbolic execution engines for interpreted languages. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[22] Edouard Bugnion, Vitaly Chipounov, and George Candea. Lightweight snapshots and system-level backtracking. In *Workshop on Hot Topics in Operating Systems*, 2013.

[23] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems*, 30(4):12, 2012.

[24] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Intl. Conf. on Virtual Execution Environments*, 2007.

[25] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Intl. Conf. on Virtual Execution Environments*, 2007.

[26] Michael Burrows, Ulfar Erlingson, Shun-Tak Leung, Mark T. Vandevoorde, Carl A. Waldspurger, Kevin Walker, and William E. Weihl. Efficient and flexible value sampling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[27] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[28] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[29] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.*, 2012.

[30] João C. M. Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *ACM EuroSys European Conf. on Computer Systems*, 2012.

[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. 2012.

[32] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Workshop on Hot Topics in Operating Systems*, 2001.

[33] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

[34] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[35] Andy Chou, Jun-Feng Yang, Bejamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. 2001.

[36] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conf.*, 2008.

[37] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[38] Cristina Cifuentes and Bernhard Scholz. Parfait — designing a scalable bug checker. In *Static Analysis Workshop*, 2008.

[39] The Clang compiler. http://clang.llvm.org/.

[40] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In *Working Conf. on Reverse Engineering*, 2009.

[41] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Symp. on Operating Systems Principles*, 2007.

[42] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Symp. on Operating Systems Principles*, 2007.

[43] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering. In *USENIX Security Symp.*, 2007.

[44] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Conf. on Computer and Communication Security*, 2008.

[45] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[46] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[47] Mihai Dobrescu and Katerina Argyraki. Software Dataplane Verification. In *Symp. on Networked Systems Design and Implem.*, 2014.

[48] Dtrace. http://www.sun.com/bigadmin/content/dtrace/index.jsp, 2011.

[49] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Workshop on Hot Topics in Operating Systems*, 2007.

[50] Peter Feiner, Angela Demke, and Brown Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[51] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[52] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.

[53] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symp.*, 2003.

[54] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We crashed, now what? In *Workshop on Hot Topics in Dependable Systems*, 2010.

[55] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[56] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Symp. on Principles of Programming Languages*, 1997.

[57] Patrice Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007.

[58] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[59] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[60] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com.

[61] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2. 2011.

[62] Isabelle. http://isabelle.in.tum.de/.

[63] Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

[64] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conf.*, 2006.

[65] Jungo. WinDriver device driver development tookit, version 9.0. http://www.jungo.com/windriver.html, 2007.

[66] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Symp. on Operating Systems Principles*, 2009.

[67] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Working Conf. on Reverse Engineering*, 2012.

[68] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *10th Intl. Conf. on Formal Methods in Computer-Aided Design*, 2010.

[69] James C. King. A new approach to program testing. In *Intl. Conf. on Reliable Software*, 1975.

[70] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

[71] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *USENIX Annual Technical Conf.*, 2002.

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

[73] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[74] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[75] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implem.*, 2012.

[76] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Symp. on Principles of Database Systems*, 2005.

[77] Chris Lattner. LLVM related publications. Official LLVM web site. Retrieved on 2010-12-04. http://llvm.org.

[78] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[79] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symp. on Operating Sys. Design and Implem.*, 2004.

[80] John Levon and Philippe Elie. Oprofile. http://oprofile.sourceforge.net, 1998.

[81] Lua: A lightweight embeddable scripting language. http://www.lua.org/, 2010.

[82] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[83] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[84] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Symp. on Operating Sys. Design and Implem.*, 2000.

[85] Microsoft security advisory #944653: Vulnerability in Macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[86] Microsoft Windows Driver Kit. http://www.microsoft.com/whdc/devtools/WDK, 2009.

[87] Microsoft. WHDC: Develop hardware for windows. http://www.microsoft.com/whdc, 2011.

[88] Microsoft. Windbg. http://msdn.microsoft.com/en-us/windows/hardware/gg463009, 2011.

[89] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[90] MITRE. CVE-2010-0232: Microsoft Windows NT #GP trap handler allows users to switch kernel stack, 2010. http://cve.mitre.org.

[91] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1), January 1993.

[92] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symp. on Security and Privacy*, 2007.

[93] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *Intl. Conf. on Software Testing Verification and Validation*, 2009.

[94] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[95] NDISwrapper. http://ndiswrapper.sourceforge.net, 2008.

[96] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Construction*, 2002.

[97] Vince Orgovan and Mike Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[98] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *ACM EuroSys European Conf. on Computer Systems*, 2008.

[99] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

[100] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Linux Symposium*, 2005.

[101] Project UDI. Uniform Driver Interface. http://udi.certek.com/, 2008.

[102] Corina Păsăreanu, Peter Mehlitz, David Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[103] Tero Pulkkinen, Karl Nelson, Esa Pulkkinen, Murray Cumming, and Martin Schulze. libsigc++ — The Typesafe Callback Framework for C++. `http://libsigc.sourceforge.net/`, 2011.

[104] R. Beuchat, P. Ienne et al. FPGA4U. http://fpga4u.epfl.ch/.

[105] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In *Symp. on Operating Sys. Design and Implem.*, 2012.

[106] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, January 1997.

[107] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *Symp. on Operating Systems Principles*, 2009.

[108] Raimondas Sasnauskas, Philipp Kaiser, Russ Lucas Jukić, and Klaus Wehrle. Integration testing of protocol implementations using symbolic distributed execution. In *Intl. Workshop on Rigorous Protocol Engineering*, 2012.

[109] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[110] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Working Conf. on Reverse Engineering*, 2002.

[111] Koushik Sen. Concolic testing. In *Intl. Conf. on Automated Software Engineering*, 2007.

[112] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Symp. on the Foundations of Software Eng.*, 2005.

[113] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conf.*, 2012.

[114] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer - Data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[115] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *5thIntl. Workshop on Systems Software Verification*, 2010.

[116] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. *Binary Translation*, volume 36. February 1993.

[117] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Intl. Conf. on Information Systems Security*, 2008.

[118] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. HAIL: a language for easy and correct device access. In *Intl. Conf. on Embedded Software*, 2005.

[119] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), 2006.

[120] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[121] The PaX team. Address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`.

[122] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[123] Valgrind, 2011. http://valgrind.org/.

[124] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Symp. on Operating Systems Principles*, 1993.

[125] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Symp. on Operating Systems Principles*, 2013.

[126] David Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[127] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[128] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[129] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent model checking

of unmodified distributed systems. In *Symp. on Networked Systems Design and Implem.*, 2009.

[130] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Symp. on Operating Sys. Design and Implem.*, 2006.

[131] Heng Yin, Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Conf. on Computer and Communication Security*, 2007.

[132] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2007.

[133] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security Symp.*, 2014.

[134] Anna Zaks and Rajeev Joshi. Verifying multi-threaded C programs with SPIN. In *Intl. SPIN Workshop*, 2008.

[135] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

VITALY CHIPOUNOV

BUILDING TOOLS FOR AUTOMATED ANALYSIS OF PROPRIETARY SOFTWARE SYSTEMS

School of Computer and Communication Sciences
EPFL – IC – DSLAB, Station 14
Building INN, Room 319
1015 Lausanne, Switzerland

vitaly.chipounov@epfl.ch
http://people.epfl.ch/vitaly.chipounov
Tel. +41 (0) 21 693 8189

CURRENT RESEARCH

I built S2E, a platform for multi-path in-vivo analysis of complex software systems, at the Dependable Systems Laboratory, led by Prof. George Candea. S2E empowers developers to build practical analysis tools such as comprehensive performance profilers, tools for reverse engineering of proprietary software, and bug finders for both kernel-mode and user-mode binaries. The S2E platform is automated, scalable, and allows for flexible precision depending on the analysis. This is achieved by two new techniques: *selective symbolic execution*, a way to automatically minimize the amount of code that has to be executed symbolically given a target analysis, and *relaxed execution consistency models*, a way to make principled performance/precision trade-offs in complex analyses.

S2E is a virtual machine augmented with symbolic execution. Users install and run an unmodified x86 software stack in S2E, including programs, libraries, the OS kernel, and drivers. Symbolic execution then automatically explores hundreds of thousands paths through the system, allowing users to check desired properties even in corner-case situations. Unlike existing analysis tools, S2E does not require to write code in special languages or model the environment.
With this platform, I developed a tool – RevNIC – that reverse-engineers proprietary closed-source device drivers to synthesize new, safer, and portable device drivers for different operating systems and architectures. RevNIC takes a binary device driver, explores it automatically across multiple paths to witness all the ways in which the driver communicates with the hardware, and synthesizes an equivalent driver that captures this interaction.

We also successfully used S2E to implement a novel bug finder – DDT – that uncovered many bugs in various Microsoft-certified drivers that have been shipping for years with Windows. S2E combines our novel selective symbolic execution approach with state-of-the-art virtualization techniques to scale multi-path exploration to systems composed of millions of lines of code.

S2E is publicly available[1] and has an active community of more than 150 users, including several research institutions.

EDUCATION

2008-now	**Ph.D. Student in Computer Science** *EPFL, Switzerland*
2008	**M.S. in Computer Science** *EPFL, Switzerland*
2008	**Minor in Management of Technology and Entrepreneurship** *EPFL, Switzerland*
2006	**B.S. in Computer Science** *EPFL, Switzerland*
2005	**Exchange year as Erasmus student** *Karlsruhe Institute of Technology (KIT), Germany*
2003	**Baccalauréat National (highest honors)** *Lycée International de Ferney-Voltaire, France*

1 http://s2e.epfl.ch

PEER-REVIEWED PUBLICATIONS

JOURNAL PAPER

The S2E Platform: Design, Implementation, and Applications
Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea
In ACM Transactions on Computer Systems (TOCS), vol. 30, num. 1, 2012.

PEER-REVIEWED CONFERENCE PAPERS

S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems
Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea
In Proceedings of the 16th ASPLOS Conference, 2011 **BEST PAPER AWARD**

Testing Closed-Source Binary Device Drivers with DDT
Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea
USENIX Annual Technical Conference, 2010

Reverse Engineering of Binary Device Drivers with RevNIC
Vitaly Chipounov and George Candea
In Proceedings of the 5th ACM SIGOPS/EuroSys European Conference, 2010

PEER-REVIEWED WORKSHOP PAPERS

The Case for System-level Backtracking
Edouard Bugnion, Vitaly Chipounov, and George Candea
In Workshop on Hot Topics in Operating Systems, May 2013

Enabling Sophisticated Analysis of x86 Binaries with RevGen
Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea
In Proceedings of the 7th Workshop on Hot Topics in System Dependability, 2011

Selective Symbolic Execution
Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea
In Proceedings of the 5th Workshop on Hot Topics in System Dependability, 2009

Reverse-Engineering Drivers for Safety and Portability
Vitaly Chipounov and George Candea
In Proceedings of the 4th Workshop on Hot Topics in System Dependability, 2008

PATENT

System and method for in-vivo multi-path analysis of software systems
George Candea, Vitaly Chipounov, and Volodymyr Kuznetsov (US# 61/405,224 – pending)

MEDIA COVERAGE

The Achilles' Heel of Your Computer
In MIT Technology Review, June 30, 2010

Software spürt schlechte Gerätetreiber auf
In Pressetext, July 1, 2010

AWARDS

Intel Doctoral Student Honor Programme Award 2013
Recipient of a $30,000 award

Silver Prize at the World Open Source Challenge 2012
DDT, the automated device driver testing tool

Best paper award at *ASPLOS 2011*
S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems

WORK EXPERIENCE

TEACHING

Software Engineering, 2009-2013
In charge of the pool of teaching assistants, Android/Eclipse tool chain setup on the computer labs, preparing and grading assignments, and giving several lectures.

System-oriented Programming, 2008-2010
Assisting undergraduate students in the learning of the Unix environment, shell scripting and C programming. Responsible for organizing exams and supervising pools of teaching assistants.

Real-Time Embedded Systems, 2008
Responsible for the design, implementation, and integration in Altera Quartus/NIOS IDE of hardware and software components used during the course (MMC/SD Card reader, FAT driver, VGA controller)

Computer Architecture, 2007
Responsible for assisting the students in the implementation of a NIOS-compatible soft-core processor for the FPGA4U development board and the grading of a large number of exams.

Information Technology Project, 2007
Managing and evaluating student groups for the implementation of a peer-to-peer file sharing project.

EXTERNAL REVIEWER FOR SYSTEMS CONFERENCES

SOSP (Symposium on Operating Systems Principles): 2011, 2013
DSN (Annual IEEE/IFIP International Conference on Dependable Systems and Networks): 2011
EuroSys (ACM SIGOPS/EuroSys European Conference on Computer Systems): 2008, 2011
ASPLOS (ACM Conf. on Architectural Support for Programming Langs. and OSes): 2010
HotOS (Workshop on Hot Topics in Operating Systems): 2009, 2011, 2013
USENIX (USENIX Annual Technical Conference): 2009, 2011

MISCELLANEOUS

CONTESTS

Finalist of the French National Informatics Contest[2] (2003, 2004, 2005)

LANGUAGE SKILLS

French: native, **Russian**: native, **English**: fluent (Cambridge Certificate in Advanced English),
German: working technical knowledge, **Spanish**: basic knowledge

2 http://www.prologin.org