Improving Scalability of Symbolic Execution for Software with Complex Environment Interfaces

THÈSE N^{O} 6719 (2015)

PRÉSENTÉE LE 13 JUILLET 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DES SYSTEMES FIABLES

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ștefan BUCUR

acceptée sur proposition du jury:

Prof. J. R. Larus, président du jury
Prof. G. Candea, directeur de thèse
Prof. V. Adve, rapporteur
Prof. J. Kinder, rapporteur
Prof. W. Zwaenepoel, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

Abstract (German)

Manuelles Testen von Software ist aufwändig und fehleranfällig. Dennoch ist es die unter Fachleuten beliebteste Methode zur Qualitätssicherung. Die Automatisierung des Testprozesses verspricht eine höhere Effektivität insbesondere zum Auffinden von Fehlern in Randfällen. *Symbolische Softwareausführung* zeichnet sich als automatische Testtechnik dadurch aus, dass sie keine falsch positiven Resultate hat, mögliche Programmausführungen abschliessend aufzählt, und besonders interessante Ausführungen prioritisieren kann. In der Praxis erschwert jedoch die sogenannte Path Explosion – die Tatsache, dass die Anzahl Programmausführungen im Verhältnis zur Programmgrösse exponentiell ansteigt – die Anwendung von Symbolischer Ausführung, denn Software besteht heutzutage oft aus Millionen von Zeilen Programmcode.

Um Software effizient symbolisch zu testen, nutzen Entwickler die Modularität der Software und testen die einzelnen Systemkomponenten separat. Eine Komponente benötigt jedoch eine Umgebung, in der sie ihre Aufgabe erfüllen kann. Die Schnittstelle zu dieser Umgebung muss von der symbolischen Ausführungsplattform bereitgestellt werden, und zwar möglichst effizient, präzis und komplett. Dies ist das *Umgebungsproblem*. Es ist schwierig, das Umgebungsproblem ein für alle mal zu lösen, denn seine Natur hängt von der gegebenen Schnittstelle und ihrer Implementierung ab.

Diese Doktorarbeit behandelt zwei Fälle des Umgebungsproblems in symbolischer Ausführung, welche zwei Extreme im Spektrum der *Schnittstellenstabilität* abdecken. (1) Systemprogramme, welche mit einem Betriebssystem interagieren, dessen Semantik stabil und gut dokumentiert ist (z.B. POSIX); (2) Programme, welche in höheren, dynamischen Sprachen wie Python, Ruby oder JavaScript geschrieben sind, deren Semantik und Schnittstellen sich laufend weiterentwickeln.

Der Beitrag dieser Doktorarbeit zum Lösen des Umweltproblems im Fall von stabilen Betriebssystemschnittstellen ist die Idee, das Modell des Betriebssystems in zwei Teile zu trennen: Ein Kern von primitiven Funktionen, welche direkt in die Hostumgebung integriert sind, und darauf aufbauend eine vollständige Emulation des Betriebssystems innerhalb der zu testenden Gastumgebung. Bereits zwei primitive Funktionen genügen, um eine komplexe Schnittstelle wie POSIX zu unterstützen: Threads mit Synchronisation, und Adressräume mit gemeinsam genutzten Spe-

icher. Unser Prototyp dieser Idee ist die symbolische Ausführungsplattform *Cloud9*. Ihr genaues und effizientes Modell der POSIX-Schnittstelle deckt Fehler in Systemprogrammen, Webservern und verteilten Systemen auf, welche anderweitig schwer zu reproduzieren sind. Cloud9 ist unter http://cloud9.epfl.ch verfügbar.

Für Programme in dynamischen Sprachen stellt diese Arbeit die Idee vor, den Interpreter der Programmiersprache als „ausführbare Spezifikation" zu verwenden. Der Interpreter läuft in einer maschinennahen symbolischen Testumgebung (z.B. auf der Ebene von X86) und führt das zu testende Programm aus. Das Gesamtsystem wird dadurch zu einer symbolischen Testumgebung auf der Ebene der dynamischen Sprache. Um die Komplexität zu bewältigen, die durch das Ausführen des Interpreters entsteht, wird in dieser Arbeit die klassenuniforme Pfadanalyse (CUPA, class-uniform path analysis) eingeführt, eine Heuristik zur Prioritisierung von Pfaden. CUPA gruppiert Pfade in Äquivalenzklassen basierend auf Zielvorgaben der Analyse. Wir verwirklichten diese Ideen in unserem Prototyp *Chef*, einer symbolischen Ausführungsumgebung für interpretierte Sprachen. Chef generiert bis zu 1000 Mal mehr Testfälle für populäre Python- und Luapakete als die naïve Ausführung des Interpreters. Chef ist unter http://dslab.epfl.ch/proj/chef verfügbar.

Keywords: Symbolische Ausführung, Programmumgebungen, Systemsoftware, interpretierte Programmiersprachen.

Abstract

Manual software testing is laborious and prone to human error. Yet, among practitioners, it is the most popular method for quality assurance. Automating the test case generation promises better effectiveness, especially for exposing corner-case bugs. *Symbolic execution* stands out as an automated testing technique that has no false positives, it eventually enumerates all feasible program executions, and can prioritize executions of interest. However, path explosion—the fact that the number of program executions is typically at least exponential in the size of the program—hinders the applicability of symbolic execution in the real world, where software commonly reaches millions of lines of code.

In practice, large systems can be efficiently executed symbolically by exploiting their modularity and thus symbolically execute the different parts of the system separately. However, a component typically depends on its environment to perform its task. Thus, a symbolic execution engine needs to provide an environment interface that is efficient, while maintaining accuracy and completeness. This conundrum is known as *the environment problem*. Systematically addressing the environment problem is challenging, as its instantiation depends on the nature of the environment and its interface.

This thesis addresses two instances of the environment problem in symbolic execution, which are at opposite ends of the spectrum of *interface stability*: (1) system software interacting with an operating system with stable and well-documented semantics (e.g., POSIX), and (2) high-level programs written in dynamic languages, such as Python, Ruby, or JavaScript, whose semantics and interfaces are continuously evolving.

To address the environment problem for stable operating system interfaces, this thesis introduces the idea of splitting an operating system model into a core set of primitives built into the engine at host level and, on top of it, the full operating system interface emulated inside the guest. As few as two primitives are sufficient to support a complex interface such as POSIX: threads with synchronization and address spaces with shared memory. We prototyped this idea in the *Cloud9* symbolic execution platform. Cloud9's accurate and efficient POSIX model exposes hard-to-reproduce bugs in systems such as UNIX utilities, web servers, and distributed systems. Cloud9 is available at http://cloud9.epfl.ch.

For programs written in high-level interpreted languages, this thesis introduces the idea of using the language interpreter as an "executable language specification". The interpreter runs inside a low-level (e.g., x86) symbolic execution engine while it executes the target program. The aggregate system acts as a high-level symbolic execution engine for the program. To manage the complexity of symbolically executing the entire interpreter, this thesis introduces Class-Uniform Path Analysis (CUPA), an algorithm for prioritizing paths that groups paths into equivalence classes according to a coverage goal. We built a prototype of these ideas in the form of *Chef*, a symbolic execution platform for interpreted languages that generates up to 1000 times more tests in popular Python and Lua packages compared to a plain execution of the interpreters. Chef is available at http://dslab.epfl.ch/proj/chef/.

Keywords: Symbolic execution, program environments, systems software, interpreted languages.

Acknowledgments

This work would not have happened without the people who have supported me throughout this journey.

I am deeply grateful to my advisor, George Candea, an exceptional model of strength, vision, and kindness. I am indebted for his patience and wisdom, for pushing me out my comfort zone to become independent in pursuing my goals and reaching for the stars. I feel very fortunate to having worked with him and I wouldn't have gotten this far without his mentorship.

I thank my outstanding thesis committee: Vikram Adve, Johannes Kinder, Willy Zwaenepoel, and Jim Larus. Their insightful comments and advice helped crystallize the vision of the thesis.

All the work presented here is the result of collaboration with many incredibly bright people. I am grateful to Johannes Kinder for his tenacious help with Chef. During our many insightful discussions, he would bring clarity to the most difficult problems. I thank Vitaly Chipounov for helping me navigate the fine mechanics of the S2E system. I am indebted to Vlad Ureche, who was instrumental in Cloud9's successful encounter with real-world software. I thank Cristian Zamfir for his advice and patience during the most critical parts of my PhD. I thank Volodymyr Kuznetsov for his insights and for being a model of resourcefulness. I thank Martin Weber for his work and for withstanding my mentoring. I am grateful to Liviu Ciortea for easing my way to symbolic execution. I thank Istvan Haller, Calin Iorgulescu, Ayrat Khalimov, and Tudor Cazangiu for their contributions to Cloud9.

My PhD experience was defined by the interactions I had with the phenomenal team in the Dependable Systems lab. I am truly grateful to Silviu Andrica, Radu Banabic, Vitaly Chipounov, Alex Copot, Horatiu Jula, Baris Kasikci, Johannes Kinder, Volodymyr Kuznetsov, Georg Schmid, Ana Sima, Jonas Wagner, Cristian Zamfir, and Peter Zankov for the inspiring moments and making the lab a fun place to be. I am especially thankful to Nicoletta Isaac, who always made sure everything in the lab ran smoothly.

I thank Ed Bugnion and Jim Larus for their inspiring advice and immensely useful lessons. I am thankful to Burak Emir and Lorenzo Martignoni, my mentors at Google, for their guidance and support.

I thank Jonas Wagner for his precious help with translating the thesis abstract to German. I

thank Amer Chamseddine, Loïc Gardiol, Benjamin Schubert, and Martin Weber for their valuable feedback on my presentation dry-runs and their availability on a short notice.

I am grateful to Cristian Cadar, Daniel Dunbar, and Dawson Engler for creating and open-sourcing KLEE. Beside serving as a foundation for my own work, I learned a lot about symbolic execution by studying and tinkering with its code.

I thank Google for generously supporting a significant part of my research through a fellowship. I thank EPFL for providing me a unique working environment and unparalleled resources to pursue my work. I thank ERC for their important financial support.

I thank my parents, Jana and Mihai Bucur, for their selfless support and encouragement to follow my dreams. Any of my accomplishments bears the mark of their upbringing. I thank my brother, Andrei Bucur, for his care and inspiration.

I thank Alexandra Olteanu for her tireless support and care, for her wisdom, and for being a pillar of strength throughout the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we describe the problem being addressed in this thesis and present a brief overview of the solution we propose. Subsequently, in Chapter 2, we describe in more depth the prior art and the work related to ours, in order to gain an understanding of the landscape in which our solution appeared. The rest of the thesis covers our solution.

1.1 Problem Definition

1.1.1 Limitations of Manual Software Testing

A crucial aspect of software development is ensuring that the written code behaves as intended. Today, this is particularly important, as software is playing an increasingly dominant role in our lives, from keeping our private data in the cloud to powering our ubiquitous smartphones.

Ideally, software would be written correctly by construction. This was the way software engineering was envisioned in the early days of computer science [40]; the code would be written together with the mathematical proof of its correctness.

However, the decades of software development experience that followed have painted a different picture. As it turns out, the software complexity—up to hundreds of millions of lines of code—and rapid pace of feature development have precluded a formal, rigorous approach. With the exception of a few safety-critical segments, such as avionics [15], automotive [20], or medical equipment, most of the software industry today relies on *testing* for its quality assurance [4].

Testing a program consists of exercising multiple different paths through it and checking whether "they do the right thing." In other words, testing is a way to produce partial evidence of correctness, and thus increase confidence in the tested software. Yet, test suites provide inadequate coverage of all the inputs a program could handle. For instance, the test suite of the Chromium browser contains about one hundred thousand tests [34]. This suite is thoroughly comprehensive

by industry standards, yet it represents only a small fraction of all possible inputs the browser may receive.

Test suites also tend to be tedious to write and maintain. Statistics show that, on average, developers spend as much as half of their time doing testing [68], and companies allocate up to a quarter of their IT budget for software testing [30].

The net result is that, despite the effort and resources invested in manual testing, bugs escape quality assurance and make it into production [79]. The industry average for bug density is a staggering 15 errors per 1000 lines of shipped code, while the most thorough quality assurance processes reach 0.1 errors per 1000 lines of code [68].

Today, this problem is amplified by having an increasing number of businesses online and hence vulnerable to remote attacks that exploit software vulnerabilities. For instance, in 2014, there were 19 vulnerabilities with CVEs reported on average *per day* [74]. These vulnerabilities are exploited to cause major disruptions [11] or leaks of sensitive data [11, 80]. On average, a data breach cost $3.8 million in 2015 [51], and the most prominent cases cost over $100 million [1].

Alas, with a few exceptions—e.g., seL4 [59], a recent effort of formally verifying an operating system kernel—switching to a formal development model is not feasible for most of the software written today. Despite recent advancements in formal techniques and tools, which have brought down the cost of building formally-proven software, it still takes on the order of person-years to develop a few thousand lines of verified code [59]. This rate is currently unsustainable for most commodity software. Therefore, the second best option today is to automate the software testing process itself.

1.1.2 Automated Testing: Black-box vs. White-box

The simplest (and most popular) form of automated testing consists of randomly generating program inputs and observing whether the program crashes [69, 35, 96, 92, 97]. The inputs are typically obtained by fuzzing [69], i.e., randomly mutating a known valid input, such as an image file or a productivity suite document. This form of testing is called “blackbox”, because the input generation does not take into account the structure of the program under test [14]. Despite their conceptual simplicity, fuzzers are effective at discovering bugs [96, 92, 97] (albeit shallow) and are currently the state of the practice in automated testing.

However, plain random fuzzers are ineffective at discovering corner-case bugs—bugs that manifest only under particular inputs in the program. Consider the simple example in Figure 1.1, where the program checks the input for the particular value 42 before performing a null-pointer access. A random fuzzer has *less than a one in a billion* chance of hitting the bug with each input generated. The vast majority of the generated inputs are therefore redundant.

```
void foo(int x) {
  . . .
  if (x == 42) {
    *((char *)0) = 0;
  }
  . . .
}
```

Figure 1.1: Example program for which random fuzzing is unlikely to uncover the memory error bug (the red line).

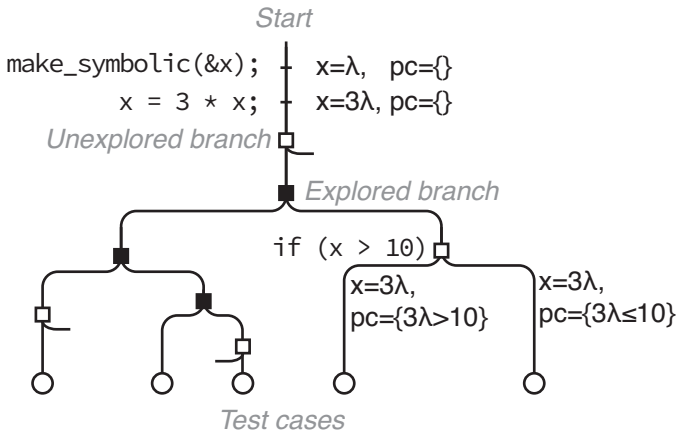

Figure 1.2: Symbolic execution tree for a simple example. Program states maintain symbolic values for variable `x` and carry a path condition (pc).

A better approach is "greybox" testing, i.e., to generate tests that follow a certain structure, as defined by a specification, such as protocol format [3], grammar [35], or input precondition [18]. However, this method would still likely miss bugs whose triggering inputs are not exposed in the input specification.

The most precise approach is to take the program structure itself into account when generating inputs. This form of testing is called "whitebox"; its goal is to generate inputs that take the program along previously unexplored execution paths. Symbolic execution is the most successful automated whitebox testing technique to be applied to commodity software, in terms of size of software and bugs found [17, 83, 28, 27].

1.1.3 Systematic Path Exploration Using Symbolic Execution

Symbolic execution works by executing a program with *symbolic* instead of concrete input values, to cover the entire input space and thus all possible program execution paths (Figure 1.2). For instance, a function `foo(int x)` is executed with a symbolic variable λ assigned to `x`. Statements using `x` manipulate it symbolically: `x := x * 3` updates the value of `x` to $3 \cdot \lambda$. During execution,

the assignment of variables to expressions is kept in a *symbolic store* of the program execution state.

Whenever the symbolic execution engine encounters a conditional branch, the execution state forks into two states, whose executions proceed independently. For each state, this process repeats for subsequent branches, turning an otherwise linear execution into a *symbolic execution tree*.

Each program state keeps the conjunction of the branch conditions taken as a *path condition*. The path condition is a formula over the symbolic program inputs, whose solutions are concrete input assignments (e.g., $\lambda = 42$) that take the program along the same execution path. The satisfiability of the formula is decided by a constraint solver, which is typically an external off-the-shelf tool [43, 39, 12]. The symbolic execution engine queries the solver whenever it needs to decide the feasibility of each branch condition, or when it generates a *test case* at the end of an execution path.

Symbolic execution is *complete*, as it systematically enumerates all program paths. It is also *sound*, because each execution path corresponds to a real execution, replayable using a test case. The two properties make symbolic execution highly effective at automatically generating high-coverage test suites [25], finding bugs [17, 31], and even debugging [98, 37, 54, 58].

Alas, symbolic execution is challenging to apply to non-trivial software. The number of paths in the symbolic execution tree is roughly exponential in the number of program branches, with loops further amplifying the problem. Even for moderately sized programs with hundreds of branches, the size of the execution tree becomes too large to be completely covered in a reasonable amount of time. This is commonly known as the "path explosion" problem [24, 25, 44, 16, 61]. In practice, given a limited time budget, most symbolic execution engines resort to prioritizing paths guided by a pluggable engine component called a *search strategy* [24, 66, 45, 25, 46].

1.1.4 The Environment Problem in Symbolic Execution

In practice, large systems can be efficiently executed symbolically by exploiting their modularity and thus symbolically execute the different parts of the system separately. However, a component typically depends on its environment to perform its task. This aspect is commonly encountered in the real world, where programs interact with the operating system, web applications use libraries, and so on. Thus, we refer to this software as *real-world*:

> In this thesis, we define **real-world software** as programs whose logic *depends* on external functionality, such as libraries or the operating system. We call this external functionality the *environment* of the program, with which the program interacts through an environment *interface*.

To execute real-world software, a symbolic execution engine needs to provide the interface of its environment. To reap the benefits of modularity, the engine should provide an interface that is more efficient than simply running the entire system symbolically. At the same time, the engine should maintain accuracy and completeness, in order to avoid missing program paths or introducing false positives. This conundrum is known as *the environment problem* [25]:

> The **environment problem** in symbolic execution is the trade-off between achieving *efficiency* and maintaining soundness and completeness in the target program when providing its environment interface.

While it also affects other program analysis techniques, such as static analysis [8] and software model checking [90], the environment problem is particularly challenging for symbolic execution [25, 77], because of the environment complexity combined with the accuracy and completeness requirements.

First, the size of the environment—code and state—may be much larger than the size of the program itself. For instance, consider a 50 lines of code system utility that prints on the terminal the contents of a file. To perform this task, the program calls the operating system for accessing files and printing on the screen, implemented in tens of thousands of lines of kernel and library code.

Second, the environment is often stateful and its state is closely coupled with the program state. In our previous example, the kernel keeps the open file and the terminal in a file table, whereas the program keeps file descriptors that index into the file table.

More sophisticated systems maintain even stronger relationships with their environments. For instance, the functionality of a web server depends on the connection state and semantics of network sockets, on the memory management logic, the concurrency model, etc. as provided by the operating system and the C library.

One simple approach for dealing with the environment problem is to let the calls into the environment go through concretely. For instance, the symbolic parameters of a system call could be forcefully assigned concrete values satisfying the path condition, before executing the system call. However, this approach risks introducing inconsistencies in the execution and miss feasible paths in the target program itself. For example, if opening a file with a symbolic name would succeed or fail depending on whether the file exists or not, concretizing to an existing file name would cause the symbolic execution to miss the error handling path in the program. For this example, a possible approach to simplifying the environment, while maintaining program coverage, is to employ fault injection at the environment interface: the symbolic execution engine forces the system call to both succeed and return a failure code, to exercise both successful and failing paths in the program.

Abstraction	Shallow	Deep
Examples	Library of arbitrary-precision numbers exposed as buffers of digits to C programs.	Native arbitrary-precision datatypes in dynamic languages (e.g., Python).
Encapsulation	Weak	Strong
Examples	Linux file system implemented as a kernel module.	Linux file system implemented using the FUSE user-space interface.
Stability	Stable	Frequently-changing
Examples	A web application that uses a standard web interface, such as CGI.	A web application that uses the fast-changing native API of its web server (e.g., Apache).

Table 1.1: Examples of environment interfaces, classified along main axes—abstraction and encapsulation—and stability in time.

In general, systematically addressing the environment problem is challenging, as its manifestations depend on the nature of the environment interface and its implementation.

To broadly structure the problem space, we classify in Table 1.1 the instances of the environment problem along three main axes of the environment interface: abstraction, encapsulation, and interface stability in time. Abstraction and encapsulation are two essential principles in system design. For environments, abstraction refers to hiding the environment state behind simpler entities. Encapsulation refers to restricting access to the environment state through narrow, well-defined interfaces. Finally, interface stability refers to the frequency of changes in the interface specification. For instance, well-documented and standardized interfaces tend to change less often.

For each combination of the three axes, a symbolic execution engine needs to approach the environment problem in different ways. To give a sense of the environment diversity, Table 1.1 provides concrete examples for each level of abstraction, encapsulation, and stability occurring in real-world software.

1.2 Solution Overview: A Tailored Approach to The Environment Problem

This thesis leverages the insight that the distribution of environment instances is skewed, with some types of interfaces being more common than others. Instead of attacking the entire range, this thesis addresses *two* of the most common instances of the environment problem: (1) software interacting with the operating system, and (2) high-level programs written in dynamic languages, such as Python, Ruby, and JavaScript. The operating system interface is used by virtually all systems

software, as it is the only way for a process to access resources in a modern operating system. For programs written in high-level dynamic languages, the environment is the language runtime—an interpreter or virtual machine. Dynamic languages are increasingly popular for developing web applications and for system integration. In January 2015, the top 4 languages used in open-source repositories on GitHub and on Stack Overflow were JavaScript, Java, PHP, and Python [75].

While both interfaces provide deep abstractions and strong encapsulation, they exhibit opposite characteristics in terms of stability and size.

On the one hand, an operating system interface is stable and well documented. Moreover, while the interface consist of hundreds of system calls, they operate with simple data types—integers and memory buffers—and their usage follows a power-law distribution [9], such that most software only uses a small fraction of them.

On the other hand, the interface of a dynamic language runtime is only partially specified and the specification changes frequently. Moreover, the language relies heavily on hundreds of built-in functions, such as string operations, data structure manipulation, and parsers, which are used thoroughly in most programs. These functions are not implemented in the language itself, but are part of the runtime implementation, typically written in C.

1.2.1 Symbolic Models for a Stable Operating System Interface

To address the environment problem for operating system interfaces, this thesis introduces the idea of using a split environment model: a core set of operating system primitives is built into the symbolic execution engine and the full operating system interface is emulated on top, as guest code.

Our insight is that as few as *two* primitives are sufficient to support complex operating system interfaces such as POSIX: threads with synchronization, and address spaces with shared memory. This results in a substantially simpler implementation in the symbolic execution engine, with opportunities of reusing existing symbolic execution components.

We prototyped our design in the *Cloud9* symbolic execution platform for POSIX programs. In under 7 KLOC, Cloud9 provides accurate and efficient models for files, network sockets, threads, processes, synchronization, IPC, signals, and other functions. We used Cloud9 to test complex system utilities such as curl, web servers such as Apache and lighttpd, and other networked services, such as memcached. As a result, Cloud9 uncovered new bugs, including a security vulnerability in memcached, and generated high-coverage test suites.

1.2.2 Using Interpreters as Specifications for Fast-changing Languages

For programs written in high-level dynamic languages, such as Python, Ruby, or JavaScript, writing a complete and accurate model is a significant engineering effort. The language semantics are complex, partially specified, and change frequently. Moreover, they rely heavily on functionality built into their runtime, so a symbolic execution engine ought to model it from scratch.

This thesis introduces the idea of using the language interpreter—the de facto standard of the language semantics—as an "executable language specification": the interpreter runs in a lower-level (e.g., x86) symbolic execution engine, while it interprets the target program. In turn, the aggregate system acts as a high-level symbolic execution engine for the target program.

The system automatically converts between the symbolic execution tree of the interpreter and that of the target program. It does so by partitioning the interpreter paths into segments corresponding to the program statements executed on the path. The interpreter paths with the same sequences of statements map to the same program path.

To circumvent the path explosion arising by executing the large interpreter implementations, which go up to hundreds of thousands of lines of code, we introduce Class-Uniform Path Analysis (CUPA), a family of path prioritization heuristics for maximizing a given coverage metric. CUPA works by grouping paths into equivalence classes, according to a coverage goal. The prioritization is done by uniformly choosing groups instead of paths. Any path selection bias introduced by program locations with higher path explosion is contained within one equivalence class, with the net result that the execution of paths in the interpreter is distributed more uniformly.

We prototyped these ideas in the Chef symbolic execution platform for interpreted languages. With Chef, we obtained engines for Python and Lua, which generated test suites and found bugs in popular library packages.

1.3 Thesis Roadmap

The rest of the thesis presents in detail the design, implementation, and evaluation of the two approaches to the environment problem.

Chapter 2 provides more background on symbolic execution and the environment problem, by surveying the related work and positioning our contributions with respect to it.

Chapter 3 expands on the idea of modeling the operating system interface. It presents the core abstractions that are built into the symbolic execution engine, then elaborates on the specifics of modeling the POSIX interface on top of them.

Chapter 4 presents the approach of using interpreters as executable language specifications. It presents an overview of the Chef system, then it goes into the details of converting between the

low-level symbolic execution of the interpreter and the high-level program execution.

Chapter 5 presents the ongoing work and longer-term vision of building a cloud-based testing service for cloud applications based on the techniques introduced in this thesis.

Chapter 6 provides the experimental evaluation of the two systems we built, along two main dimensions: the effectiveness in generating test suites and finding bugs, and the scalability to real-world software.

Chapter 7 ends the thesis with conclusions.

1.4 Previously Published Work

This thesis includes material previously published in conference and workshop papers:

- Chapter 3: Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *ACM European Conference on Computer Systems (EuroSys)*, 2011. [23]

- Chapter 4: Stefan Bucur, Johannes Kinder, and George Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. [22]

- Chapter 5: Stefan Bucur, Johannes Kinder, and George Candea. Making Automated Testing of Cloud Applications an Integral Component of PaaS. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2013. [21]

Chapter 2

Related Work

In this chapter, we review the existing work that touches on the environment problem in symbolic execution. We start with a brief overview of the symbolic execution engines that advanced the state of the art and made the technique applicable to real-world software (Section 2.1). We then systematize the approaches these tools took to address the environment problem (Section 2.2). The last part of the chapter covers the use of symbolic execution for high-level languages, which is one particular focus of our work (Section 2.3).

2.1 Symbolic Execution for Real-world Software Testing: A Brief History

Symbolic execution was introduced almost four decades ago as a test case generation technique [58, 19]. The first tools worked on domain-specific languages, supported basic numeric operations, and were mostly used for interactive debugging. The effectiveness of the technique on more complex programs was limited by the lack of efficient and expressive constraint solvers and by slow hardware.

The exponential increase in hardware performance and the availability of fast off-the-shelf constraint solvers [70, 42, 43, 39, 12] amplified the research in symbolic execution. Over the past decade, the new generation of symbolic execution engines produced test suites and found bugs on real-world software, ranging from small system utilities to large application suites.

In this section, we highlight some of the most important tools and discuss the techniques that expanded the applicability of symbolic execution to real-world software. Whenever appropriate, we touch on the environment problem; however, we cover it extensively later on, in Section 2.2.

Concrete + Symbolic Execution To run a real-world program, a symbolic execution engine must provide the interface of its environment and the semantics of the language features it uses. A first solution introduced by early engines, such as DART [45] and CUTE [85], is to execute the program normally, using concrete inputs, while maintaining the symbolic execution state only during the execution of the program code itself. When the symbolic semantics are not available, such as inside external library calls, the program execution would still be able proceed. By using this approach, DART and CUTE found crashes in small-to-moderate C programs, such as protocol implementations and data structure libraries.

This form of execution is also called concolic (*conc*rete + symb*olic*) [85], or "whitebox fuzzing" [46]. The symbolic execution space is explored through successive runs of the program with concrete inputs. The symbolic constraints collected at each run are used to generate new inputs, which are used in subsequent executions.

Specialized Solver Support While concolic execution completes each execution path, it can miss *other* feasible paths in the symbolic execution tree when the symbolic semantics of the program are not available. In particular, a major limitation of the early symbolic execution engines was the lack of support for reasoning precisely and efficiently about common expressions involving bitwise arithmetic and memory accesses. For instance, DART and CUTE used a solver only for linear integer constraints, which limited the scope of the symbolic analysis, because the solver theory did not capture the peculiarities of bitwise arithmetic (e.g., overflows), nor supported pointer arithmetic.

Subsequent symbolic execution engines took advantage of new breakthroughs in constraint solving and employed high performance off-the-shelf solvers, such as Z3 [39], STP [43], or CVC [12]. These solvers can reason efficiently about a large set of operations commonly encountered in program execution. For example, the EXE [26] symbolic execution engine was among the first to employ such a solver (STP), which was co-designed with EXE to accurately express machine operations in low-level languages such as C. EXE modeled the program memory as a flat array of bytes, with support for arbitrary pointer reads and writes, mixed with fixed-width integer operations. As a result, EXE targeted and found bugs in larger system software such as the `udhcpd` DHPC server, packet filters, and the `pcre` Perl regular expression library.

Low-level Symbolic Interpretation DART, CUTE, and EXE implement symbolic execution at the C source code level, by using CIL [72] to instrument the code with additional statements that maintain the symbolic state as the program executes. However, this approach is tedious. Even for a low-level language like C, the size of the language specification is about 200 pages[1], so supporting

[1] This figure refers to the C90 standard [52].

all language features symbolically is an extensive engineering effort. Moreover, the engineering cost would increase for languages with more features and richer data types, such as C++.

Instead, current state-of-the-art symbolic execution engines [46, 25, 86, 33, 31] take advantage of the fact that C, C++, and other languages are compiled to a lower-level representation, such as x86 binary or LLVM [62] bytecode, which is simpler to reason about. Two prominent examples are SAGE and KLEE.

SAGE [17, 46]—the best known use of symbolic execution in production—performs concolic execution at binary level. This enables SAGE to target large applications with deep execution paths, such as Windows format parsers and Microsoft Office applications. SAGE found thousands of Windows vulnerabilities at development time [17, 47].

KLEE [25] is arguably the most popular symbolic execution engine used in research, having served as a foundation for a wide range of other tools. KLEE works as a symbolic interpreter for LLVM IR bytecode, obtained by compiling source programs using an LLVM-based compiler such as Clang. KLEE was designed to target systems code, such as command line utilities (e.g., `ls` or `echo`). Since these programs call into the operating system, which is not available as LLVM IR, KLEE employs *models* that approximate the real operating system. The KLEE models replaced parts of the standard C library and were linked with the target program, providing basic support for file operations—the most common dependency among the targeted utilities. Unmodeled system calls were still allowed, by passing them to the host environment, executed concretely on behalf of the KLEE process. KLEE found bugs and generated test suites with over 90% statement coverage on average in the Coreutils suite [25]. It also found bugs in other systems software, such as Busybox and the HiStar kernel.

"In-vivo" Symbolic Execution The idea of targeting low-level representations, such as x86, in symbolic execution engendered a new approach to the environment problem, where the program and its environment are executed *together* inside the symbolic execution engine. This approach is particularly convenient for cases where the interface between the program and its environment is broad and difficult to model.

Full-VM symbolic execution engines, such as S2E [33] and BitBlaze [86], execute symbolically any part of the software stack "in-vivo", without the need of the program source code, nor operating system models. The program state effectively is the CPU state (registers, flags, etc.) and the contents of the physical memory, and the environment is the perhiperal hardware. For instance, to address the environment problem, S2E provides symbolic models for devices, for testing their drivers running in a kernel. Most notably, S2E discovered vulnerabilities in several Windows device drivers, some of which were Microsoft-certified [60].

Symbolic execution also found use in security testing [7, 31, 86], as a more precise approach

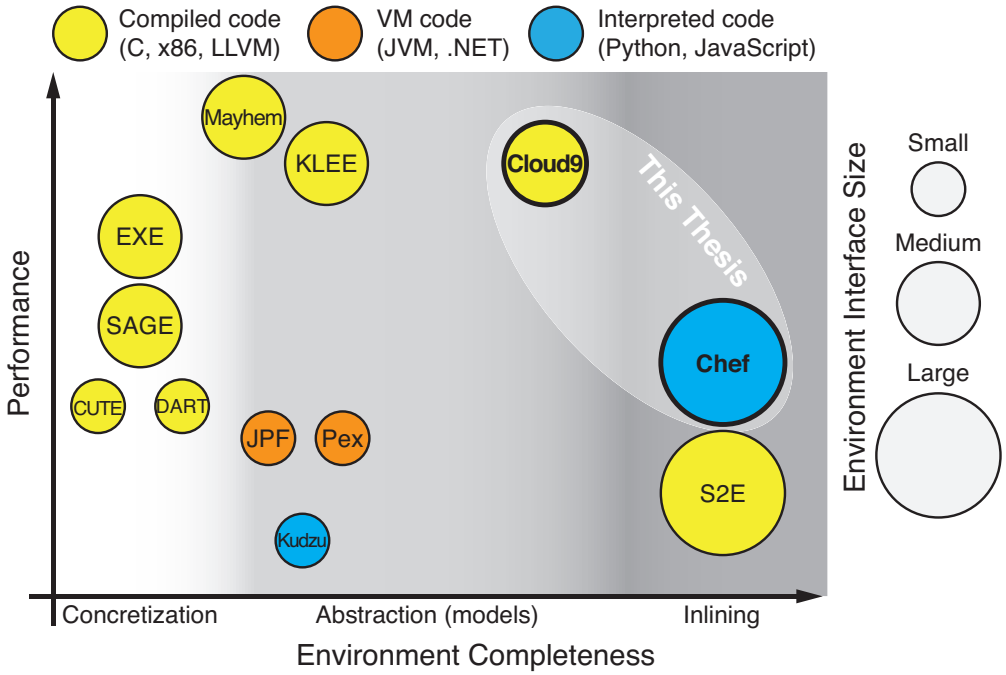

Figure 2.1: Qualitative comparison of most important symbolic execution engines with respect to the environment problem. X axis indicates completeness of symbolic environment, ranging from no symbolic support (concretization) to full environment inlining. Y axis indicates relative performance of engines in terms of paths throughput, as deduced from each engine's evaluation numbers.

compared to random fuzzing. Similarly to random fuzzing tools, symbolic execution-based security tools assume that bugs are more likely to be found in certain program parts, such as shallow parser components, or in code triggered by slight variations of valid inputs. For these tools, reaching such bugs is more important than achieving completeness, so the symbolic execution engines resort to simplifications to increase test case throughput and reach deeper executions in larger programs, at the expense of losing completeness. For example, the AEG [7] tool uses symbolic analysis to both find bugs and automatically generate exploits (get a shell) from the bugs. AEG uses heuristics that prioritize buggy paths and employs simple operating system models that minimize path explosion. The Mayhem [31] tool employs a simplified symbolic memory model where write addresses are concretized. Both tools found exploitable vulnerabilities in Linux and Windows programs.

2.2 Tackling the Environment Problem in Symbolic Execution

To efficiently handle real-world programs, symbolic execution engines need to minimize the time spent in the environment, while ensuring correct behavior in the program. Existing approaches

roughly fall into three categories, according to the degree of completeness in handling symbolic data going to the environment: (1) concretizing the calls to the environment (no symbolic support at all), (2) abstracting away the environment complexity through models, and (3) "inlining" the environment with the program.

Figure 2.1 positions the existing work with respect to this classification, while qualitatively indicating the size of the environment interface targeted by the engine and the relative performance attained by each engine in its class. There are several patterns emerging from this classification.

First, the less complete the environment support, the higher the engine performance tends to be. A simpler environment creates fewer execution paths and leads to simpler symbolic expressions in the program state. In turn, the path throughput in the target program increases. This effect is most visible in symbolic execution engines that resort to concretization or employ simple models, such as EXE [26], Mayhem [31], or KLEE [25].

Second, the completeness of the environment tends to be proportional to the complexity of the environment interface of the programs targeted. For example, S2E [33] executes device drivers together with the kernel, as modeling the latter would incur a significant engineering effort.

Third, the engine performance at a given environment completeness level (a vertical in Figure 2.1) depends on the targeted language and the constraint solver performance. Engines targeting compiled code (e.g., KLEE [25] or SAGE [46]) are typically faster than engines targeting high-level code (e.g., Kudzu [82] or Java PathFinder [5]). Similarly, newer engines that rely on faster solvers (e.g., EXE [26], SAGE [46], or KLEE [25]) fare better than the early engines (e.g., DART [45] or CUTE [85]).

Finally, Figure 2.1 shows that symbolic execution engines make a *trade-off* between the completeness of their environment and the performance they are willing to sacrifice. From this perspective, our contributions, embodied in the Chef and Cloud9 symbolic execution engines, advance the state of the art by improving this tradeoff. Cloud9 retains the performance of model-based environment handling, while providing an accurate operating system model that comes closer in completeness to an inlined approach. Chef benefits from the completeness of inlining the environment—the language interpreter—with the interpreted program, while significantly improving performance over naïve symbolic execution.

In the rest of this section, we discuss in more detail each of the three environment approaches. We use the program in Figure 2.2 as a running example to illustrate the tradeoffs made by each approach. The program opens a file using a symbolic name, reads data from it, then closes it and terminates. Despite being simple, this example exposes the challenges of the environment problem.

```c
#include <fcntl.h>
#include <unistd.h>

int main() {
  char name[] = “input.txt”;
  make_symbolic(name, sizeof(name));
  int fd = open(name, O_RDONLY);
  if (fd == -1) {
    return 1;
  }
  char buffer[16];
  int size = read(fd, &buffer, 16);
  close(fd);
  return 0;
}
```

Figure 2.2: Example of C program using the POSIX operating system interface to open and read a file with a symbolic name.

2.2.1 Under-approximation (Concretization)

A simple approach to reducing the time spent in the environment is to concretize the values that cross the program boundary [45, 46, 25, 26]. For instance, when the symbolic file name, say μ, is passed as an argument to the **open** call in Figure 2.2, the symbolic execution engine replaces the argument with a concrete example m. This causes the execution to proceed linearly inside the environment.

For programs that only send output to the environment (e.g., by calling **printf**), concretization works well. However, for programs that share state with the environment (e.g., maintain open file descriptors), concretization causes missed feasible paths later in the program execution, or introduces inconsistencies in the program state.

We illustrate these points on our example in Figure 2.2. First, the returned file descriptor of the concretized **open** system call is concrete, which causes the symbolic execution engine to exercise only one of the two possible outcomes—success or failure. Second, in order to maintain soundness, concretization introduces constraints in the path condition, which bind the symbolic arguments to concrete examples (in our case, $pc_{new} := pc_{old} \wedge (\mu = m)$). This means the symbolic file name becomes concrete for the rest of the execution path, precluding any symbolic branching on its value. Avoiding the extra constraint in the path condition would keep the file name symbolic, but introduces inconsistency in the program state: while the file descriptor returned by **open** is *either* valid or failed, the unconstrained symbolic file name would represent *both* valid and invalid file names. This may lead to test cases that increase the coverage in the program, at the expense of also introducing false positives, which manifest as duplicate test cases.

A concrete environment may also introduce inconsistencies if it is shared among all execution paths [25]. This is commonly the case with non-concolic ("purely-symbolic") execution, when the symbolic execution engine maintains all execution states as data structures in its address space. Calls to the environment would then be passed on to the environment of the engine itself, potentially causing cross-talking between different execution paths.

2.2.2 Abstraction (Modeling)

An alternative approach to simplifying the environment is to replace its concrete implementation with a simpler one—a model—that captures its essential behavior [25, 31, 7]. In our running example, a model of the `open` and `read` system calls should capture only the two traits used by the program: the ability to read data from the file and the failure when the file is not found. The former can be modeled as an in-memory buffer that is allocated when the file is open and copied to the user buffer in the `read` system call. The latter can be simulated in the model by branching the symbolic execution state and returning -1 in one path. For larger and more realistic programs, an accurate and complete model is significantly larger. Nonetheless, a model is still smaller than the original implementation—often by orders of magnitude [25]—as it drops non-functional requirements, such as performance and fault tolerance.

Executable Models The example model we described is *executable code* that runs together with the target program inside the symbolic execution engine, as guest code [25]. This approach benefits from the features available in the symbolic execution runtime. The model can directly access the program state, such as reading and manipulating call arguments using their native types, and directly returning values to the program. In addition, symbolic data implicitly flows through the model code. The file model in our example implicitly accepts symbolic file names in the `name` argument of the `open` system call.

However, some environment features are expensive to capture at the guest level, such as complex control flow abstractions like interrupts and concurrency. In those cases, symbolic execution engines provide models that are built into the engine implementation itself. For example, the S2E platform [33] provides models for symbolic hardware as built-in platform plugins. Despite their generality, built-in models are harder to write, since they explicitly handle the symbolic program state.

Functional Models Models may not only be executable, but also functional, expressed as predicates [8, 64] or abstract types with theories [81, 82, 64]. For instance, the model of a function can be provided as a function summary, consisting of pre-condition and post-condition predicates over

the program state [64, 8]. In our running example, the summary of the `open` system call could be $(fd = -1) \vee (fd \geq 3)$. Function summaries may be more efficient than executable code, as they avoid branching the program state and result in more concise symbolic expressions.

Another form of functional modeling is to treat the environment calls as uninterpreted functions [81, 82]. The idea is to *lazily* record each environment call as an opaque operation during execution, and provide a decision procedure that interprets the operation at constraint solving time. A benefit of this approach is that specialized decision procedures may be significantly more efficient than executing symbolically the environment implementation. For example, symbolic execution of web applications [82, 63, 84] benefits from constraint solvers that support strings [39, 56, 99, 89], as strings are ubiquitous in this domain. Another benefit is that the interpretation step may end up being skipped if the function call is not relevant for satisfiability [81], as opposed to an executable model, which would be executed regardless of whether the results are used or not.

Although they are effective at reducing path explosion, models are expensive to write correctly and completely. As a result, they tend to be employed only for simple or stable environment semantics, or when the model accuracy is less important (e.g., security testing [7]).

2.2.3 Inlining (Whole-system Analysis)

Concretizing environment calls or writing models is unfeasible when the environment interface is large or maintains a complex state, strongly coupled to the program state. For example, once loaded, a device driver typically can interact with the entire operating system kernel. Concretizing all kernel calls would drastically reduce the exploration space, while modeling all kernel features would be expensive, especially for kernels whose internals change often, such as Linux.

For these cases, an approach that preserves correctness and completeness is to bundle the environment with the program and execute it symbolically as one target. Full-VM symbolic execution engines implicitly provide this approach [32, 86]. For symbolic execution engines that execute programs in isolation [25, 46], the environment is explicitly compiled in the target format and linked with the program.

However, this approach reduces the effectiveness of symbolic execution on the original target program, due to the path explosion in the entire system, which may be orders of magnitude larger than the program. Existing work employs several techniques to mitigate path explosion in the environment. For example, S2E [32] introduces execution consistency models, which are principles of converting between symbolic and concrete values when crossing the program boundaries, in order to control the path explosion in the environment.

Another approach is to manually or automatically change the environment implementation

to reduce its complexity. For example, the KLOVER symbolic execution engine for C++ programs [64] alters the implementation of the `std::string` STL class to avoid branching inside operators such as equality. In general, the environment code can also be automatically compiled in a form sensible to symbolic execution [93].

2.2.4 Developer Control of Symbolic Environments

The developer interface to a symbolic execution engine plays an important role in controlling its effectiveness and efficiency. The best known sources of developer inputs are the search selection strategies and the injection of symbolic data in the system. For example, KLEE runs command-line utilities with symbolic arguments defined using a special syntax, in line with the rest of the target arguments. For instance, `ls -l -sym-arg 1 3` runs `ls` with the second argument set as symbolic, between 1 and 3 characters. This syntax can be used to define symbolic arguments or symbolic files.

The developer control over the symbolic input was encapsulated in concepts such as parameterized unit tests [88], which extend regular unit tests with parameters marked as symbolic inputs during symbolic execution. Similarly, QuickCheck [35] allows writing input specifications in Haskell, which are used to instantiate inputs during random testing.

2.3 Symbolic Execution for High-level Languages

2.3.1 Virtual Machine-based Languages: Java and C#

Beyond low-level system software, a significant amount of software is written in higher-level languages, offering features such as garbage collection, reflection, and built-in data structures.

Some of these languages, such as Java and C#, are compiled to a lower-level bytecode representation and executed in a virtual machine. The bytecode format is standardized and well-documented, which facilitates the development of dedicated symbolic execution engines for their runtimes. For example, the Java PathFinder project [90, 5, 91] provides a model checker and symbolic execution framework for Java bytecode. Similarly, Pex [87] is a symbolic execution engine for .NET that has recently been distributed as part of the Visual Studio IDE.

The high-level environments pose additional challenges for symbolic execution related to the complexity of their data and execution model. For example, symbolic program inputs can be both scalar and object-based. Java PathFinder handles symbolic object inputs using a technique called generalized symbolic execution [55], where symbolic objects are lazily initialized in response to the member accesses encountered during program execution. Pex takes a similar approach for

handling symbolic inputs in the .NET runtime.

2.3.2 Dynamic Interpreted Languages

The high-level languages that are never compiled, but source-interpreted, such as Python, JavaScript, Ruby, or Lua, are significantly more challenging for symbolic execution. These languages have complex semantics that are under-specified, their features evolve rapidly from one version to another, and they rely on large built-in support libraries [53, 81, 78]. Building and maintaining a symbolic execution engine for these languages is a significant engineering effort. As a result, the existing engines target only domain-specific subsets of their languages.

Supporting Symbolic Semantics Existing work addresses the problem of providing language semantics for symbolic execution in three major ways: by writing a symbolic interpreter for the language statements [29], executing the program concolically [81, 84], and by requiring program cooperation [36].

Writing a symbolic interpreter from scratch involves reconstructing the semantics for all the language constructs used by the target programs. For example, the NICE-PySE [29] symbolic execution engine, which is part of the NICE framework for testing OpenFlow applications, interprets the internal Python bytecode instructions generated by the interpreter when executing a program. NICE-PySE is a Python application itself and uses the language reflection mechanisms to obtain, instrument and interpret the internal representation of the program.

The downside of writing a complete interpreter from scratch—including a symbolic execution engine—is that whenever a construct is not supported, the interpreter would halt. To mitigate this, other engines take the concolic execution approach, where they run the real interpreter in concrete mode, and maintain in parallel the symbolic semantics of the language statements. On the path segments where the symbolic semantics are not available, the execution can still make progress concretely, keeping the program state sound. CutiePy [81] uses the tracing API of the Python interpreter to maintain the symbolic state in lock-step with the concrete execution. The Jalangi dynamic instrumentation framework for JavaScript [84] rewrites the target JavaScript program to insert statements for maintaining the symbolic state, similar to other symbolic execution engines for C programs [45, 85, 26].

In certain applications, symbolic execution is used to execute a model of a larger system, in the vein of model checking. In this case, the model uses the specific API provided by the engine; the high-level language acts as a DSL, whose features are fully supported by the engine. For example, the symbolic execution engine of the scalability testing tool Commuter [36] is entirely built in Python and offers an API for symbolically modeling operating system calls using the

Python language.

Effectiveness of Symbolic Execution for Interpreted Languages The existing symbolic execution engines for interpreted languages have shown promise for finding bugs, in areas where these languages are increasingly popular, such as web applications [82, 6, 57].

For instance, the Kudzu [82] symbolic execution engine for JavaScript was used to detect code injection vulnerabilities. The Apollo [6] engine targets PHP code to detect runtime and HTML errors, while Ardilla [57] discovered SQL injection and cross-site scripting vulnerabilities in PHP applications. This potential is also confirmed by our findings with Chef, whose engines for Python and Lua discovered hangs and unexpected exceptions in popular packages (Section 6.5).

2.4 Summary

Symbolic execution has been successfully used for finding bugs and generating high-coverage test suites. The advancements in constraint solving and the approaches to the environment problem have been increasingly extending the reach of symbolic execution to larger and more complex real-world software. Our work distinguishes itself as attaining a better tradeoff between the completeness of the environment provided to programs and the symbolic execution engine performance.

Chapter 3

Primitives for Symbolic Models for Stable Operating System Interfaces

Operating systems expose an interface that is typically stable and well documented. This engenders a *modeling* approach to the environment problem, which only involves the one-time effort to produce such a model. However, modern operating systems are complex and provide a wide set of abstractions to programs, such as processes and threads, IPC, networking, and files. Modeling these abstractions is challenging.

In this chapter, we present an approach to modeling the operating system interface that relies on *splitting* the model in a set of most primitive operating system abstractions, on top of which a full operating system model can be implemented with reasonable effort (Section 3.1). We leverage the operating system model to expose hard-to-reproduce scenarios in program execution by providing a *symbolic test* interface for developers (Section 3.2). Finally, we report on our experience applying these principles when building a POSIX operating system model with support for threads, processes, sockets, pipes, polling, and more (Section 3.3).

3.1 Splitting the Operating System Environment

In this section, we first present the design space of an operating system model (Section 3.1.1). We then present our approach of a split model (Section 3.1.2) and how we leverage it using symbolic tests.

3.1.1 The Design Space of Operating System Models

The goal of a symbolic model is to simulate the behavior of a real execution environment, while maintaining the necessary symbolic state behind the environment interface. The symbolic execu-

tion engine can then seamlessly transition back and forth between the program and the environment.

Writing and maintaining a model can be laborious and prone to error [33]. However, for operating system interfaces, which are typically stable and well documented, investing the time to model them becomes worthwhile.

Ideally, a symbolic model would be implemented as code that gets executed by the symbolic execution engine (i.e., guest-level code), substituting the implementation of the operating system and the C library. Such a model can be substantially faster. For instance, in the Linux kernel, transferring a packet between two hosts exercises the entire TCP/IP networking stack and the associated driver code, amounting to over 30 KLOC. In contrast, our POSIX model achieves the same functionality in about 1.5 KLOC. Requirements that complicate a real environment/OS implementation, such as performance and extensibility, can be ignored in a symbolic model.

However, not all operating system abstractions can be directly expressed as guest code. In general, the problematic abstractions are those *incompatible* with the execution model of the symbolic execution engine. For example, providing support for multiple execution threads may not be achievable in the guest of a symbolic execution engine designed to run sequentially, unless the guest can manipulate the current stack and program counter. There are other abstractions incompatible with the typical sequential single-process execution model of a symbolic execution engine, such as processes, synchronization, and shared memory.

A possible alternative is to implement the operating system model inside the symbolic execution engine, where we can define any abstraction. However, this approach is undesirable. A model built into a symbolic execution engine is significantly more expensive to produce, because the model code has to handle explicitly symbolic data and the program state. For example, while a guest model of the `open` system call could directly use the string buffer of the file name passed as argument, a built-in model needs to explicitly invoke read operations for the string bytes, then extract the character values from the symbolic AST expressions.

3.1.2 A Split Operating System Model

Our key idea is to take the best from both worlds and provide an operating system model that is *split* between a set of built-in primitives and guest-level code. The built-in primitives model only the minimal operating system abstractions that would be more expensive or impossible to model at the guest level. In turn, the guest-level code implements a complete operating system interface on top of these primitives. In analogy to the system calls exposed by an operating system to user programs, the symbolic execution engine provides its primitives to the guest through a set of "symbolic system calls".

The built-in primitives provide the operating system abstractions that depend on the execution model of the symbolic execution engine. The execution model includes aspects such as the program memory model and the control flow mechanisms. For performance reasons, these aspects are typically encoded in the logic of the symbolic execution engine.

We identified two abstractions that are prevalent in operating systems and that should be built into the symbolic execution engine: *multithreading* and *address spaces*. Multithreading is best provided as a built-in primitive that is integrated with the control flow mechanisms of the symbolic execution engine. Similarly, providing multiple isolated address spaces is best provided by the memory model of the symbolic execution engine. For example, the functionality needed to support address spaces (e.g., cloning) is shared with the requirements of cloning the execution state after a symbolic branch.

Many other common abstractions do not need to be built into the symbolic execution engine, but can be emulated as guest code on top of threads and address spaces. Such derived abstractions include mechanisms for inter-process communication, such as sockets, pipes and files. Various synchronization mechanisms, such as mutexes, semaphores, condition variables, and barriers can be provided on top of basic multi-threading primitives that put a thread to sleep and wake it up later.

We designed the support for multithreading and address spaces according to two goals: (1) minimizing the complexity of the guest code using them, and (2) capturing all possible behaviors in the real operating system, including corner-case, hard-to-reproduce scenarios. The latter is especially relevant when using the symbolic execution engine to find bugs occurring in the program interaction with the operating system, as we later show in the evaluation (Section 6.5).

3.1.3 Built-in Primitives: Multithreading and Address Spaces

We next describe the design of multithreading support and address spaces in a symbolic execution engine.

Multithreading To provide support for multiple threads, the symbolic execution engine maintains in each execution state a set of per-thread stacks, holding the current program location, the execution call chain, and local variables. During symbolic execution, the execution alternates between the threads, governed by a thread scheduler built into the symbolic execution engine.

To simplify synchronization inside the guest model, we use a cooperative scheduler. An enabled thread runs uninterrupted (atomically), until either (a) the thread goes to sleep, (b) the thread is explicitly preempted, or (c) the thread is terminated with a symbolic system call.

The scheduler can be configured to schedule the next thread deterministically, or to fork the

Primitive Name	Description
`thread_create(&func)`	Create new thread that runs `func`
`thread_terminate()`	Terminate current thread
`thread_preempt()`	Preempt the current thread
`create_wqueue()`	Create a new waiting queue
`thread_sleep(wq)`	Put current thread to sleep on waiting queue
`thread_notify(wq)`	Wake threads from waiting queue
`process_fork()`	Fork the current address space and thread
`make_shared(&buf, size)`	Share memory across address spaces
`get_context()`	Get the current context (process and thread ID)

Table 3.1: Symbolic system calls for multithreading (first block), address spaces (second block). Third block contains introspection primitives. The multithreading block is further divided into thread lifecycle management, explicit scheduling, and synchronization.

execution state for each possible next thread. The latter case is useful when looking for concurrency bugs. At the same time, it can be a significant source of path explosion, so it can be selectively disabled when not needed.

The symbolic execution engine can detect hangs in the system, such as deadlocks, when a thread goes to sleep and no other thread can be scheduled.

Address Spaces In symbolic execution, program memory is typically represented as a mapping from memory locations (e.g., variables or addresses) to slots holding symbolic expressions. To provide support for address spaces, we built into the symbolic execution engine support for multiple such mappings per execution state.

Each thread in the execution state is bound to one address space and each address space with its threads forms a process in the execution state.

The symbolic execution engine provides a symbolic system call for *sharing* slots among multiple memory mappings. This mechanism provides the foundation for implementing shared memory across multiple processes. Shared memory can be used by the guest model to provide multiple forms of inter-process communication, such as sockets, files, and pipes. For example, a socket can be modeled as a pair of memory buffers, one of each direction, shared between the client and the server processes.

Symbolic System Calls Table 3.1 shows the symbolic system calls that the engine provides to the guest to support multithreading and address spaces. We detail below the most important system calls.

Threads are created in the currently executing process by calling `thread_create`. For instance,

the POSIX threads (pthreads) model makes use of this primitive in its own `pthread_create()` routine. When `thread_sleep` is called, the symbolic execution engine places the current thread on a specified waiting queue, and an enabled thread is selected for execution. Another thread may call `thread_notify` on the waiting queue and wake up one or all of the queued threads.

Cloning the current address space is available to the guest through the `process_fork` primitive, which is used, for instance, to model the POSIX `fork()` call. A memory location can be marked as shared by calling `make_shared`; it is then automatically mapped in the address spaces of the other processes in the execution state. Whenever a shared object is modified in one address space, the new version is automatically propagated to the others.

3.2 Symbolic Tests

Software systems typically have large "hand-made" test suites. Writing and maintaining these suites requires substantial human effort, for two main reasons: (1) the developers need to devise a comprehensive set of concrete inputs to cover the program behaviors and (2) setting up the environment (i.e., the operating system) to expose all relevant program interactions is difficult. We aim to reduce both burdens, while improving the quality of testing, by introducing *symbolic tests*.

A symbolic test is a piece of code executed by the symbolic execution engine, which sets up the target program, creates symbolic inputs, and configures the operating system environment. A symbolic test encompasses many similar concrete test cases into a single symbolic one—each symbolic test a developer writes is equivalent to many concrete ones. Furthermore, symbolic tests can easily uncover corner cases, as well as new, untested functionality.

A symbolic test controls the operating system model to explore conditions that are hard to produce reliably in a concrete test case, such as the occurrence of faults, concurrency side effects, or network packet reordering, dropping, and delay.

In this section, we present the developer interface for writing symbolic tests and illustrate it with a use case.

3.2.1 Testing Platform Interface

The symbolic testing API (Tables 3.2 and 3.3) allows tests to programmatically control events in the environment of the program under test. A test suite needs to simply include a header file and make the requisite calls.

Symbolic Data and Streams The generality of a test case can be expanded by introducing bytes of symbolic data. This is done by calling `make_symbolic` to mark data symbolic, a wrapper around

Function Name	Description
`make_symbolic`	Mark memory regions as symbolic
`fi_enable` `fi_disable`	Enable/disable the injection of faults
`set_max_heap`	Set heap size for symbolic `malloc`
`set_scheduler`	Set scheduler policy (e.g., round-robin)

Table 3.2: API for setting global behavior parameters.

Extended Ioctl Code	Description
`SIO_SYMBOLIC`	Turn a file or socket into a source of symbolic input
`SIO_PKT_FRAGMENT`	Enable packet fragmentation on a stream socket
`SIO_FAULT_INJ`	Enable fault injection for operations on descriptor

Table 3.3: Extended `ioctl` codes to control environmental events on a per-file-descriptor basis.

the symbolic execution engine's primitive for injecting fresh symbolic variables in the program state.

In addition to wrapping this call, we added several new primitives to the testing API. Tables 3.2 and 3.3 show how these primitives are provided in a POSIX environment. Symbolic data can be written/read to/from files, can be sent/received over the network, and can be passed via pipes. Furthermore, the `SIO_SYMBOLIC` `ioctl` code (Table 3.3) turns on/off the reception of symbolic bytes from individual files or sockets.

Network Conditions Delay, reordering, or dropping of packets causes a network data stream to be fragmented. Fragmentation can be turned on or off at the socket level, for instance, by using one of the `ioctl` extensions. Section 6.5.3 presents a case where symbolic fragmentation proved that a bug fix for the lighttpd web server was incomplete.

Fault Injection Operating system calls can return an error code when they fail. Most programs can tolerate such failed calls, but even high-quality production software misses some [67]. Such error return codes are simulated in a symbolic test whenever fault injection is turned on.

Symbolic Scheduler The built-in multithreading primitive provides multiple scheduling policies that can be controlled for purposes of testing on a per-code-region basis. Currently, it supports a round-robin scheduler and two schedulers specialized for bug finding: a variant of the iterative context bounding scheduling algorithm [71] and an exhaustive exploration of all possible scheduling decisions.

3.2.2 Usage Example: Testing Custom Header Handling in Web Server

Consider a scenario in which we want to test the support for a new **`X-NewExtension`** HTTP header, just added to a UNIX web server. We show how to write tests for this new feature.

A symbolic test suite typically starts off as an augmentation of an existing test suite; in our scenario, we reuse the existing boilerplate setup code and write a symbolic test case that marks the extension header symbolic. Whenever the code that processes the header data is executed, the symbolic execution engine forks at all the branches that depend on the header content. Similarly, the request payload can be marked symbolic to test the payload-processing part of the system:

```
char hData[10];
cloud9_make_symbolic(hData);
strcat(req, "X-NewExtension: ");
strcat(req, hData);
```

The web server may receive HTTP requests fragmented in a number of chunks, returned by individual invocations of the **`read()`** system call—the web server should run correctly regardless of the fragmentation pattern. To test different fragmentation patterns, one simply enables symbolic packet fragmentation on the client socket:

```
ioctl(ssock, SIO_PKT_FRAGMENT, RD);
```

To test how the web server handles failures in the environment, we configure the symbolic test to selectively inject faults when the server reads or sends data on a socket by placing in the symbolic test suite calls of the form:

```
ioctl(ssock, SIO_FAULT_INJ, RD | WR);
```

We can also enable/disable fault injection globally for all file descriptors within a certain region of the code using calls to **`fi_enable`** and **`fi_disable`**. For simulating low-memory conditions, we provide a **`set_max_heap`** primitive, which can be used to test the web server with different maximum heap sizes.

3.3 Case Study: A Symbolic POSIX Interface Model

We used the symbolic system call interface to build a model for the POSIX interface, with support for symbolic tests. In this section, we describe the key design decisions involved in building the model, and we illustrate the use of the symbolic system call interface. This also serves as an example for building additional models on top of the symbolic system call interface.

```
typedef struct {
  wlist_id_t wlist;
  char taken;
  unsigned int owner;
  unsigned int queued;
} mutex_data_t;

int pthread_mutex_lock(pthread_mutex_t *mutex) {
  mutex_data_t *mdata = ((mutex_data_t**)mutex);
  if (mdata->queued > 0 || mdata->taken) {
    mdata->queued++;
    cloud9_thread_sleep(mdata->wlist);
    mdata->queued--;
  }
  mdata->taken = 1;
  mdata->owner = pthread_self();
  return 0;
}

int pthread_mutex_unlock(pthread_mutex_t *mutex) {
  mutex_data_t *mdata = ((mutex_data_t**)mutex);
  if (!mdata->taken ||
      mdata->owner != pthread_self()) {
    errno = EPERM;
    return -1;
  }
  mdata->taken = 0;
  if (mdata->queued > 0)
    cloud9_thread_notify(mdata->wlist);
  return 0;
}
```

Figure 3.1: Example implementation of pthread mutex operations in the POSIX environment model.

The POSIX model uses shared memory structures to keep track of all system objects (processes, threads, sockets, etc.). The two most important data structures are stream buffers and block buffers, analogous to character and block device types in UNIX. Stream buffers model half-duplex communication channels: they are generic producer-consumer queues of bytes, with support for event notification to multiple listeners. Event notifications are used, for instance, by the polling component in the POSIX model. Block buffers are random-access, fixed-size buffers, whose operations do not block; they are used to implement symbolic files.

The symbolic execution engine maintains only basic information on running processes and threads: identifiers, running status, and parent–child information. However, the POSIX standard mandates additional information, such as open file descriptors and permission flags. This information is stored by the model in auxiliary data structures associated with the currently running threads and processes. The implementations of **`fork()`** and **`pthread_create()`** are in charge of initializing these auxiliary data structures and making the appropriate symbolic system calls.

Modeling synchronization routines is simplified by the cooperative scheduling policy: no locks are necessary, and all synchronization can be done using the sleep/notify symbolic system calls, together with reference counters. Figure 3.1 illustrates the simplicity this engenders in the implementation of pthread mutex lock and unlock.

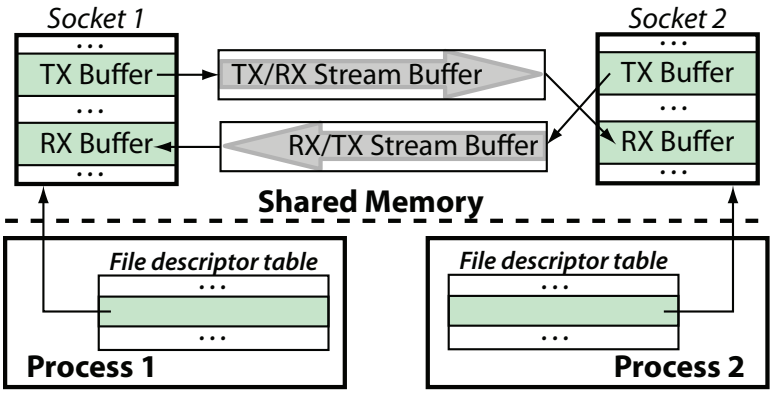

Figure 3.2: TCP network connection model using TX and RX buffers implemented as stream buffers.

To support files, we reused most of the file model semantics available in the KLEE symbolic execution engine [25]. In particular, one can either open a symbolic file (its contents comes from a symbolic block buffer), or a concrete file, in which case a concrete file descriptor is associated with the symbolic one, and all operations on the file are forwarded as external calls on the concrete descriptor.

In addition to file objects, the POSIX model adds support for networking and pipes. Currently, the TCP and UDP protocols are supported over IP and UNIX network types. Since no actual hardware is involved in the packet transmission, we can collapse the entire networking stack into a simple scheme based on two stream buffers (Figure 3.2). The network is modeled as a single-IP network with multiple available ports—this configuration is sufficient to connect multiple processes to each other, in order to simulate and test distributed systems. The model also supports pipes through the use of a single stream buffer, similar to sockets.

The POSIX model supports polling through the `select()` interface. All the software we tested can be configured to use `select()`, so it was not necessary to implement other polling mechanisms. The `select()` model relies on the event notification support offered by the stream buffers that are used in the implementation of blocking I/O objects (currently sockets and pipes).

The constraint solver used in most symbolic execution engines operates on bit vectors; as a result, symbolic formulas refer to contiguous areas of memory. In order to reduce the constraint solving overhead, we aim to reduce the amount of intermixing of concrete and symbolic data in the same memory region. Thus, the POSIX model segregates concrete from symbolic data by using static arrays for concrete data and linked lists (or other specialized structures) for symbolic data. We allocate into separate buffers potentially-symbolic data passed by the tested program through the POSIX interface.

In order to enable testing the systems presented in the evaluation section (Section 6.4), we had to add support for various other components: IPC routines, `mmap()` calls, time-related functions,

etc. Even though laborious, this was mostly an engineering exercise, so we do not discuss it further.

Finally, in some cases, it is practical to have the host OS handle parts of the environment via *external calls*. These are implemented by concretizing the symbolic parameters of a system call before invoking it from symbolically executing code. Unlike [45, 25, 26], we allow external calls *only* for stateless or read-only system calls, such as reading a system configuration file from the `/etc` directory. This restriction ensures that external concrete calls do not clobber other symbolically executing paths.

3.4 Summary

Operating systems expose a complex stateful interface to user programs. In this chapter, we showed a way to provide an operating system environment for symbolic execution by employing a split operating system model. A core set of primitives built into the symbolic execution engine serves as a base, on top of which a full operating system interface is emulated inside the guest. As few as two primitives are sufficient to support complex operating system interfaces: threads with synchronization and address spaces with shared memory. We showed how to use the core primitives to provide an accurate model of the POSIX interface. Our POSIX model includes extensions that developers can use in symbolic tests to control non-deterministic operating system events, such as thread scheduling and network flow control, in order to increase the coverage in the target programs.

Chapter 4

Using Interpreters as Specifications for Fast-changing Languages

In the previous chapter, we showed how to efficiently create accurate models for operating system interfaces, which are stable and well documented. For interfaces with unstable and incomplete specifications, such as those of dynamic languages, such as Python, Ruby, or JavaScript, the environment problem mandates a different approach. In this chapter, we present Chef, a symbolic execution platform for interpreted languages that relies on using the language interpreter as an "executable specification".

4.1 System Overview

Chef is a platform for language-specific symbolic execution. Provided with an interpreter environment, which acts as an executable language specification, Chef becomes a symbolic execution engine for the target language (see Figure 4.1). The resulting engine can be used like a hand-written one, in particular for test case generation. When supplied with a target program and a symbolic test case (also called test driver or test specification in the literature), the Chef engine outputs a set of concrete test cases, as shown in Figure 4.1.

Example We illustrate the functionality of Chef using the Python example in Figure 4.2 (left). The function `validateEmail` receives an e-mail address string and raises an exception if the address is malformed. We symbolically execute the function using a Chef engine obtained by plugging in it the Python interpreter.

The Chef engine receives the Python program together with a symbolic test that marks the `email` input argument as symbolic.

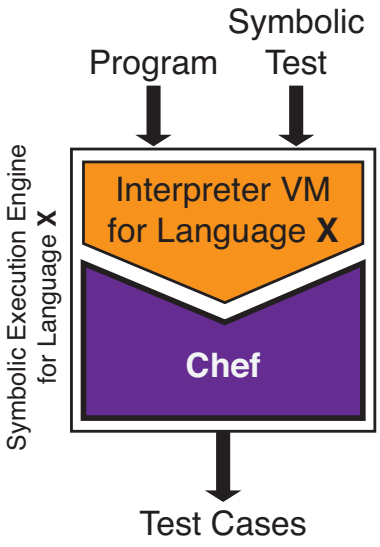

Figure 4.1: Overview of Chef's usage.

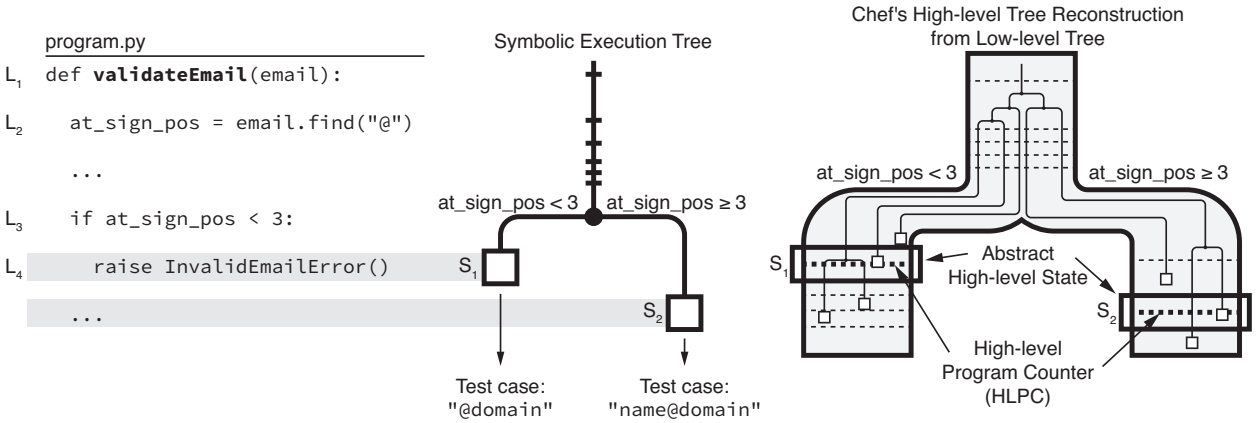

Figure 4.2: Example of Python code that validates an e-mail address given as a string (left) with corresponding symbolic execution tree (center) and mapping of interpreter paths to program paths (right). Marks on the execution path indicate program statements.

Chef constructs the symbolic execution tree of the program, as shown in Figure 4.2 (center). The paths in the tree are sequences of program locations, corresponding to a statement or bytecode instruction. Branches can occur explicitly at control flow statements (as in our example), or implicitly through exceptions.

Chef prioritizes execution states according to a pluggable search strategy (see Section 1.1.3). There are two execution states in our example: S_1, which executes the exception path, and S_2, which executes the normal path. A strategy may randomly select states to uniformly cover the set of possible paths, or it may favor the states that are triggering exceptions in the program, such as S_1.

At the end of each execution path, Chef generates a concrete string value for the `email` argument, which constitutes a test case. Each test case takes the Python program along the same execution path, when replayed in the same interpreter.

The Interpreter as Language Specification Chef is built on top of the S2E analysis platform [33], which symbolically executes the interpreter environment *at binary level*. The interpreter environment is a virtual machine that bundles the interpreter, a testing library, the operating system, and other user programs. For example, to symbolically execute the program in Figure 4.2, Chef runs the interpreter by invoking `./python program.py` inside the virtual machine.

The resulting engine is a correct symbolic execution engine for the target language *as defined by the interpreter*. It is fully precise and theoretically complete, i.e., it will not explore infeasible paths and will eventually explore all paths.[1]

4.2 Chef's Architecture as Adapter Between Engine Interfaces

Chef acts as an adapter between two symbolic execution engine interfaces: the user-facing *high-level* engine interface that receives the target program and the internal *low-level* binary engine that runs the interpreter.

The interface of a symbolic execution engine consists of a program-facing part and a strategy-facing part. Through the program-facing part, the symbolic execution engine receives the program and the symbolic test and outputs test cases. Through the strategy-facing part, the engine sends the active execution states to an external state selection strategy and receives the choice of the next state to run (see Section 1.1.3).

Figure 4.3 shows the architecture of Chef, consisting of the base and an interpreter module.

[1]The usual limitations of symbolic execution engines apply: completeness holds only under the assumption that the constraint solver can reason about all generated path conditions, and it is understood that exhaustive exploration is usually impractical in finite time.

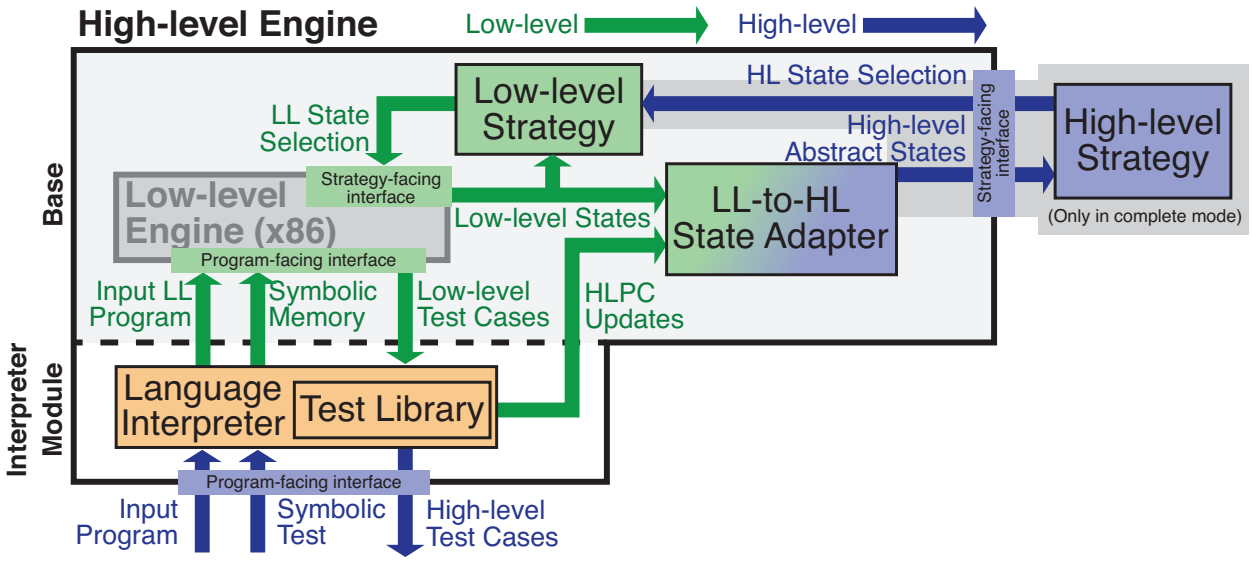

Figure 4.3: The architecture of Chef.

The base contains the *low-level* binary symbolic execution engine (S2E) that runs the interpreter, the low-level state selection strategy, and an *adapter* that creates high-level program paths based on the observed low-level interpreter paths. An external high-level strategy communicates with the base to provide the next high-level state to run.

The interpreter module contains the language interpreter, which provides the implicit semantics of the language.

Both the inner and the outer interfaces of Chef expose the program-facing and the strategy-facing part. The inner engine operates with machine-level abstractions: it runs a binary program—the interpreter—and marks memory bytes as symbolic inputs. During symbolic execution, it generates paths through the interpreter binary, prioritized by the low-level strategy. The outer engine operates with corresponding high-level abstractions: it runs an interpreted program and marks objects, instead of bytes, as symbolic. During execution, it generates paths through the high-level program.

The interpreter module *by construction* adapts the program-facing part of the two interfaces, as shown in Figure 4.3. Reusing the interpreter provides substantial savings in terms of development effort. For instance, the Python interpreter has over 400 KLOC; a hand-written symbolic execution engine would reimplement from scratch the same language semantics in an implementation of the same order of magnitude.

The main task left for Chef is to provide the low-level strategy for choosing the interpreter paths that maximize Chef's effectiveness as a high-level symbolic execution engine. We discuss this aspect in the next section.

4.3 From Low-level to High-level Symbolic Execution Semantics

The High-level Symbolic Execution Tree Consider again the example in Figure 4.2. On line L_2, the `find` method executes as a single statement at the Python level, but internally, the interpreter forks hundreds of alternate states, for each possible path in the implementation of the substring searching algorithm. Some of these paths lead the program to take the branch at line L_4 and raise the exception, while the others continue the execution normally. Chef groups the low-level interpreter paths into their corresponding high-level program paths (right side of Figure 4.2) and reports a single test case per high-level path.

To group the low-level paths into high-level paths, Chef tracks the program location—the *high-level program counter* (HLPC)—of each low-level interpreter state. The HLPC values are opaque to Chef. Their actual contents depend on the language and the interpreter. For example, for interpreters that compile programs to an intermediate bytecode representation, the HLPC can be the address in memory of each bytecode instruction (Section 4.6.1).

The low-level paths that run through the same sequence of HLPC values map to the same high-level path. This results in a high-level symbolic execution tree that "envelops" the low-level symbolic execution tree of the interpreter (Figure 4.2).

Abstract High-level States The expansion of the high-level tree is governed by the high-level strategy, which expects a set of high-level states and returns the chosen state. There is one high-level state per path, reflecting the progress of the execution on that path. However, it is the *low-level* execution states that determine the high-level paths, as previously explained. Moreover, there could be *several* low-level states on the same path. Intuitively, each low-level state executes a "slice" of the entire high-level path, and may reside at any HLPC location on the path.

To adapt between the two interface levels, Chef introduces the notion of *abstract high-level state*. Each high-level path has exactly one abstract high-level state, which resides at a HLPC location on the path. When a new high-level path branches, its abstract high-level state starts at the beginning of the path. The state subsequently advances to the next HLPC when it is selected for execution, and it is terminated when it reaches the end of the path.

An abstract high-level state may only advance to the next HLPC when the underlying low-level states on the path have executed enough of the program statement. Chef provides two policies that govern this relationship, called *state completeness models*. They are discussed in detail in the next section.

We call an abstract high-level state *blocking* when it cannot make progress on the path according to the state completeness model used. When the abstract state is blocking, the low-level

strategy selects for execution low-level states that unblock the abstract state.

The Low-level State Selection Workflow By using abstract high-level states, the selection of low-level states becomes a two-stage process, as shown in the architecture diagram (Figure 4.3). First, the low-level engine reports its execution states to an *adapter* component, which maintains the high-level symbolic execution tree and the abstract high-level states. The state adapter reports the abstract high-level states to the high-level strategy, which decides the next high-level state—and therefore path—to explore. If the state completeness model allows the state to advance on the path, the state adapter updates the HLPC directly. If not, the high-level strategy communicates its choice to the low-level strategy, which selects a low-level state on the path that helps unblock the abstract high-level state.

This state selection process both (a) provides an adapter mechanism for selecting *low-level* interpreter states for execution based on a *high-level* strategy, and (b) still preserves the high-level abstractions by hiding the interpreter and its states.

The abstract high-level states only contain the current HLPC and a path condition. Unlike the low-level states, they do not include program data, as its semantics are language-dependent. Instead, the program data is implicitly encoded and distributed across the low-level states on the path and manipulated by the interpreter.

4.4 Trading Precision for Efficiency with State Completeness Models

The state adapter uses the state completeness model to determine whether an abstract high-level state blocks or can be advanced to the next HLPC on its path. Chef provides two completeness models: *partial* and *complete*. The two models provide different trade-offs between the precision of controlling the high-level execution and the expected path throughput in the high-level symbolic execution engine.

Before we present the two models, we introduce the notion of *local path condition* and *path segments* (Figure 4.4).

Local Path Condition We define the local path condition of a high-level path at a given HLPC location as the disjunction of the path conditions of all low-level states that traversed the HLPC location, *at the moment the interpreter started executing the statement at the given HLPC*. Each statement on the high-level path has a local path condition.

The local path condition of a HLPC location is *complete* if all low-level states on the path are

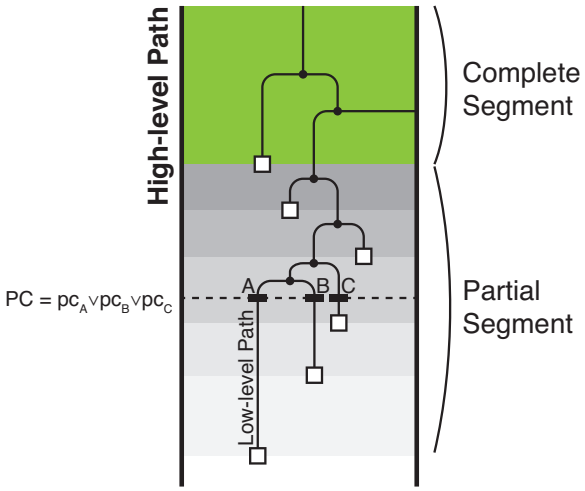

Figure 4.4: High-level execution path, segmented according to completeness of exploration by the low-level paths.

located after the statement. This means that there is no further interpreter execution that would reach the HLPC location. Hence, a complete local path condition describes *all* inputs that take the program along the given high-level path.

Conversely, if there exist low-level states that are located before the HLPC location on the path, its local path condition is *partial*. The local path condition shown in Figure 4.4 is partial, because there are three states before it. Any of them could reach the program at that location and augment the possible program executions on the path.

Path Segments We divide a high-level execution path in two segments, according to the completeness of the local path conditions of its HLPC locations:

- The *complete segment* consists of all locations traversed by all low-level states on the path. The segment starts at the beginning of the path and ends at the HLPC of the least advanced low-level state on the path (the green block in Figure 4.4).

- The *partial segment* consists of the rest of the path, which contains the locations traversed by at least one, but not all, low-level states on the path. The segment starts with the least advanced low-level state on the path and ends at the leading state on the path. Figure 4.4 shows the partial segment in nuances of gray, according to the degree of local path completeness.

Each HLPC location on the high-level path transitions from partial to complete. In case a high-level path consists of a single low-level path, the transition goes directly to complete and the partial segment is empty.

The State Completeness Models The partial and complete state models are defined with respect to the path segments on which the abstract high-level states are allowed to reside.

In the partial model, the abstract high-level states are allowed to reside anywhere on the high-level execution path, i.e., on both its complete and partial segment. On the other hand, in the complete model, the abstract high-level states are only allowed on complete segments.

This difference creates substantial divergences in the properties of the two models, which we discuss next.

4.4.1 Partial High-level Execution States

In the partial model, the abstract high-level states are allowed to reside anywhere on the high-level execution path. Therefore, the states block only at the last HLPC location on the path.

The low-level strategy unblocks the abstract high-level states by selecting for execution the leading low-level state on the path. Therefore, the cost of unblocking the state is relatively low: only one low-level state from the entire path is required, which amounts to simpler symbolic expressions and faster solver queries.

On the other hand, in the partial model, the execution of the abstract high-level state may miss feasible high-level branches. This is because the local path condition of the abstract state is incomplete, and therefore may not include the low-level executions that diverge into other high-level paths at branching points.

As a result, the high-level strategy can only exercise control over high-level paths *already discovered*. New high-level paths are discovered only as the low-level strategy repeatedly selects low-level states for execution on the existing paths.

4.4.2 Complete High-level Execution States

In the complete model, the abstract high-level states can only reside on the complete path segment. The states block at the first low-level state encountered on the path, before entering the partial segment.

The low-level strategy unblocks the abstract high-level states by executing all low-level states that reside on the next statement on the path. Therefore, compared to the partial mode, the cost of unblocking the state is significantly higher.

However, in the complete model, the feasibility of all branching instructions traversed by the abstract high-level states are decided, as the local path conditions are complete. In turn, the high-level execution strategy has knowledge of all paths forking off the currently discovered ones, and can influence the exploration ordering.

High-level State Model	**Partial**	**Complete**
Location (HLPC)	Complete and partial segments	Complete segments only
Path Condition	Incomplete	Complete
Strategies	Only at low level	Both high- and low-level
Efficiency (state cost)	One low-level state	All low-level states
Precision	Low (no branch feasibility)	Full control over states

Table 4.1: Comparing partial and complete high-level state models in Chef.

4.4.3 Choosing Between the Two Models

Table 4.1 summarizes the trade-offs between the partial and complete state models. On the one hand, the partial state model lacks precision, as it does not expose to the high-level engine interface any pending high-level states. However, it gains efficiency, as it uses as little as one low-level state to explore a high-level path, once discovered. On the other hand, the complete state model provides full precision, as it is able to determine all feasible branches along each explored path, at the expense of a more costly path exploration.

Both state models are useful for symbolic execution. Fundamentally, both models handle the same low-level state information coming from the underlying symbolic execution engine. The difference relates to how the data is organized into execution paths and test cases.

The partial state model works best for exploratory automated test case generation. The low-level strategy aims to discover new high-level paths, which are then executed efficiently using a single low-level state, which gives the test case for the path.

The precise state model is more appropriate for directed test case generation and exhaustive testing. The high-level strategy works with the low-level strategy to focus the exploration on particular high-level paths. At the same time, the decoupling between the two strategies opens up optimization opportunities at the low-level, such as state merging, which increases the efficiency of exhaustive testing.

4.5 Path Prioritization in The Interpreter

4.5.1 Class-Uniform Path Analysis (CUPA)

Consider using symbolic execution for achieving statement coverage on a program containing a function with an input-dependent loop. At each iteration, the loop forks one additional state (or exponentially many in the number of iterations, if there are branches in the loop). A strategy that selects states to explore uniformly is therefore biased toward selecting more states from this function, at the expense of states in other functions that fork less but contribute equally to the

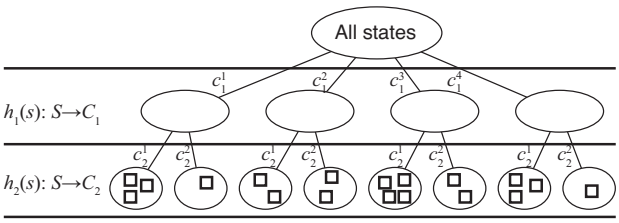

Figure 4.5: CUPA state partitioning. Each level corresponds to a state classification scheme. Child nodes partition the parent node according to the classification at their level.

statement coverage goal.

Class-Uniform Path Analysis (CUPA) reduces this bias by grouping states into classes and then choose uniformly among classes instead of states. For instance, in the above example, the grouping criterion of each state could be its current function. CUPA then first selects uniformly a function, then picks at random a state inside that function. This way, functions generating many states are still selected with equal probability to others.

In general, CUPA organizes the state queue into a hierarchy of state subsets rooted at the entire state queue (see Figure 4.5). The children of each subset partition the subset according to the *state classification scheme* at their level. A classification scheme is defined as a function $h : S \to C$, where $h(s)$ maps each state s into a class value c. States of the same parent with the same class value are sorted into the same child. CUPA selects a new state for exploration by performing a random descent in the classification tree, starting from the root. When reaching a leaf, the strategy takes out a random state from the state set and returns it to the symbolic execution engine for exploration. By default, all sibling classes on each level have equal probability of being picked, but they can be assigned weights if required.

A CUPA strategy is parameterized by the number N of levels in the tree and a classification function h_i for each level $i = 1 \ldots N$. Chef uses two instantiations of CUPA: one optimized for covering high-level paths (Section 4.5.2) and one for covering the high-level CFG, i.e., statements (Section 4.5.3).

4.5.2 Path-optimized CUPA

The goal of the path-optimized CUPA is to discover high-level paths through the program. The strategy achieves this by mitigating the bias towards high-level instructions that fork more low-level states than others, such as string operations or native calls. We instantiate a two-level CUPA strategy using the following classes:

1. The location of the state in the high-level symbolic execution tree. This is the occurrence of the state's high-level program counter (HLPC) in the unfolded high-level CFG, referred to

as the dynamic HLPC. We choose the dynamic HLPC to give each high-level path reaching the HLPC the same chance to fork and subsequently diverge.

2. The low-level x86 program counter of the state. This classification reduces the selection bias of "hot spots" of path explosion within a single complex instruction, such as a native function call.

4.5.3 Coverage-optimized CUPA

Coverage-optimized CUPA aims to discover those high-level paths that increase the basic block coverage. Based on a coverage-optimized strategy introduced by the KLEE symbolic execution engine [25], we developed a CUPA instance that partitions states according to their minimum distance to branches leading to uncovered code. Alas, dynamic language interpreters do not generally have a static CFG view of the program, so code that has not been covered yet is not accessible to the search strategy. The high-level CFG of the target program is dynamically discovered along each execution path. On this CFG, we employ heuristics that (1) identify the instruction opcodes that may branch, and (2) weigh the state selection toward states that are closer to these potential branching points.

First, Chef identifies the branching opcodes by collecting all high-level instructions that terminate a basic block with an out-degree in the CFG of at least 2 (i.e., cause branching in the control flow). We then eliminate the 10% least frequent opcodes, which correspond to exceptions or other rare control-flow events. Second, Chef identifies the potential branching points as those instructions in the CFG that have a branching opcode (as previously identified) but currently only one successor. Finally, Chef computes for each execution state the distance in the CFG to the closest such potential branching point.

Having computed this information, we instantiate a two-level CUPA strategy with the following classes:

1. The static HLPC of the state in the high-level CFG. On this level, each class is weighted by $\frac{1}{d}$, where d is the distance in the inferred high-level CFG to the closest potential branching point, making states at locations close to a target more likely to be selected.

2. The state itself (so each partition has a single element). On this level, the states are weighted by their *fork weight*.

Fork weight is computed by counting the number of consecutive forks at the same low-level program counter (i.e., at an input-dependent loop in machine code). States $1,\ldots,n$ forking from the same path at the same location get weights $p^n, p^{n-1}, \ldots, 1$, where $p < 1$ de-emphasizes states

API Call	Description
`log_pc(pc, opcode)`	Log the interpreter PC and opcode
`start_symbolic()`	Start the symbolic execution
`end_symbolic()`	Terminate the symbolic state
`make_symbolic(buf)`	Make buffer symbolic
`concretize(buf)`	Concretize buffer of bytes
`upper_bound(value)`	Get maximum value for expression on current path
`is_symbolic(buf)`	Check if buffer is symbolic
`assume(expr)`	Assume constraint

Table 4.2: The Chef API used by the interpreters running inside the S2E VM.

forked earlier ($p = 0.75$ in our implementation). The last state to fork at a certain location thus gets maximum weight, because alternating the last decision in a loop is often the quickest way to reach different program behavior (e.g., to satisfy a string equality check).

4.6 Packaging the Interpreter for Symbolic Execution

Integrating an interpreter in Chef requires preparing a virtual machine that includes the interpreter binary, a symbolic test library, and a launcher. To communicate with the interpreter runtime, Chef provides an API (Table 4.2) that will be explained along with its use.

Preparing the Interpreter Binary In principle, Chef can run unmodified interpreters for major popular languages, such as Python and JavaScript. Chef automatically obtains from the running interpreter the stream of high-level program counters (HLPCs) for each execution path, required by the state adapter to reconstruct the high-level symbolic execution tree (Section 4.6.1).

In practice, however, applying a number of optimizations in the interpreter implementation significantly improves symbolic execution performance (Section 4.6.2).

The Symbolic Test Library The unit of symbolic execution in Chef is the *symbolic test*. A symbolic test is a class, function, or other language-specific construct that encapsulates the symbolic input setup, the invocation, and the property checking of the feature targeted by the test. A symbolic test is similar to a unit test, except for an extra set of primitives for creating symbolic values.

The symbolic tests are written in the same language as the target, and require runtime support in the form of a language-specific *symbolic test library*. The symbolic test library provides the developer interface for writing symbolic tests. In addition, it translates the requests for marking language values as symbolic into the Chef API primitives for marking memory buffers as symbolic (the `make_symbolic` call, together with the `assume` call for defining conditions over the input). For

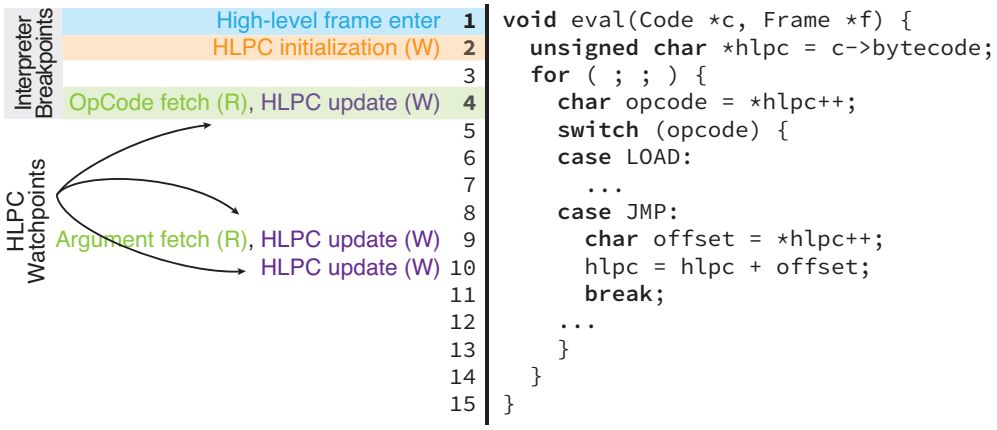

Figure 4.6: Structure of interpretation loop in most common interpreters. Lines 1, 2, and 4 are three key HLPC points. High-level control flow changes at Lines 4, 9, and 10 are detected at runtime by watchpoints on HLPC address identified at Line 2.

example, when a symbolic test requests a 10-character symbolic string, the symbolic test library creates a concrete 10-character string object, then it finds its buffer address in the internal representation and marks the buffer as symbolic. As a result, the symbolic test library is typically implemented as a native extension module, so it gains access to the internal data structures of the interpreter.

The execution of the interpreter in the virtual machine is bootstrapped by a launcher. The launcher receives the location of the symbolic test (e.g., the module file and the class name, for Python) and runs the interpreter binary with the appropriate arguments such that it starts executing the symbolic test. The launcher is typically a small web server running inside the guest, listening for commands from Chef.

4.6.1 Automated High-level Control Flow Detection in Interpreters

We defined a high-level path as a sequence of statements identified by high-level program counters (HLPCs). In this section, we present the concrete semantics of the HLPC and how Chef obtains it automatically from a running interpreter.

Standard Interpretation Model In general, obtaining the high-level program counter in a running interpreter is undecidable, because the boundaries between program statements are a high-level property of the language, and the separation may not reflect in the low-level implementation. Even at the high level, statement boundaries may be ambiguous due to functional constructs, such as lambda functions and list comprehensions, or data definitions mixed with control flow, such as the class definitions in Python.

In practice, however, most interpreters follow a common structure that *can be detected automatically*. Instead of directly interpreting the source program, interpreters first compile it to an intermediate bytecode representation, which is interpreted instruction by instruction. The bytecode is typically kept in per-function memory buffers. When a function is called, the interpreter pushes a frame on its high-level stack, which contains a pointer variable to the next instruction in the function bytecode buffer. An interpretation function receives the function bytecode and the stack frame, then executes the bytecode instructions one by one, while updating the HLPC variable of the frame (the right half of Figure 4.6). We confirmed the generality of this approach by inspecting the source code of the most popular interpreters for Python, JavaScript (Mozilla's SpiderMonkey and WebKit's JavaScriptCore), Ruby, PHP, and Lua.

For these interpreters, we define the Chef high-level statements to be bytecode instructions, and their HLPC to be their address in the bytecode buffer. As a result, to track the HLPC of the execution state, Chef monitors updates to the HLPC variable of the top high-level stack frame.

Tracking the HLPC Variable in the Interpreter To obtain the address of the top HLPC variable on the stack, Chef monitors three key locations in the interpretation function that summarize its HLPC update behavior: (1) the address of the function itself, (2) the address of the statement that initializes the HLPC variable, and (3) the address of the statement that fetches the next opcode in the bytecode buffer. The locations are highlighted on the left half of Figure 4.6. Chef uses them as follows.

First, Chef maintains a stack of HLPC frames that track the interpreter's own high-level frames. When the interpreter enters the interpretation function, Chef pushes a new HLPC frame on the current execution state. A HLPC frame contains the address of the HLPC variable and its last value in the corresponding high-level frame of the interpreter.

Second, when the interpreter executes the HLPC initialization statement, Chef stores in the HLPC frame the memory address written to by the statement as the address of the HLPC variable.

Third, when the interpreter executes the opcode fetching statement, Chef marks the beginning of a new high-level statement at the address given by the current HLPC value.

During the execution of the interpretation function, Chef sets a watchpoint on the HLPC variable in the current frame. When the variable is updated—e.g., when the next instruction is fetched, or after a branch or loop iteration—Chef correspondingly updates the HLPC value and the high-level symbolic execution tree, as needed.

Constructing the Interpreter HLPC Summary Before an interpreter is used for symbolic execution, Chef constructs its three-location HLPC summary. The summary is constructed once for each interpreter binary.

First, Chef records all memory writes in the interpreter, when running a special developer-provided "calibration" script. For each write, Chef records the interpreter location (x86 PC), the memory address, and the value written.

The role of the calibration script is to create HLPC update patterns that are recognizable among other memory writes. The calibration script should run long enough that the HLPC patterns become clearly distinguishable.

To this end, Chef uses a linear sequence of instructions to create a linear HLPC update pattern. We assume that the interpreter compiles a linear program (no branches) to a linear sequence of bytecode instructions.

After all memory writes are collected, Chef groups them by address and discards the groups whose values are not monotonically increasing. Among the remaining groups, Chef discards those with fewer writes than the number of statements in the calibration program. For this step, we assume that each statement corresponds to one or more bytecode instructions. Finally, Chef discards the groups whose write value deltas are larger than the size of a bytecode instruction. We empirically determined an upper bound threshold of 1KB.

At the end, there should be exactly one remaining group, whose address refers to the HLPC variable of the frame of the recorded execution.

From the HLPC variable, Chef obtains the three-location summary. The HLPC initialization location is the x86 PC of the first write operation in the remaining group. The HLPC initialization location then leads to the address of the interpretation function, either by dynamically tracking the low-level program stack, or by using debug symbols in the interpreter binary. Finally, the opcode fetch point corresponds to the first memory read of the HLPC variable, inside the interpretation function.

In case the calibration ends with no remaining memory write group, or more than one group remaining, the calibration fails. This could happen, for instance, when the calibration is attempted on a non-conforming interpreter, or when the calibration script is too short. For all interpreters we tested (Python, JavaScript, and Lua), a 100-statement calibration script was sufficient.

In principle, defining program statements at an intermediate representation risks missing paths at the high-level. This would happen, for instance, if the interpreter translated code with complex control flow to a linear bytecode sequence. However, in our experience, we noticed that the translated bytecode follows closely the structure of the program. In particular, interpreters perform little or no optimization on the bytecode.

Manually Annotating the High-level Program Counter When the interpreter structure diverges from our assumptions, Chef provides a fallback option of manually annotating the interpreter with the HLPC information, by placing calls to `log_pc` (Table 4.2).

4.6.2 Interpreter Optimizations

In order to maximize performance, interpreters make heavy use of special cases and sophisticated data structures. Unfortunately, these features hurt the performance of symbolic execution by amplifying path explosion and increasing the complexity of symbolic formulas [93].

We identify a number of optimizations that preserve the interpretation semantics but significantly improve symbolic execution performance. The optimizations use the Chef API in the last block of rows in Table 4.2.

Neutralizing Hash Functions Hash functions are especially common in interpreters, due to the internal use of hash tables for associative data structures (e.g., Python dictionaries or Lua tables). However, they are generally a problem in symbolic execution: a symbolic value added to a hash table (a) creates constraints that essentially ask the constraint solver to reverse a hash function, which is often hard, and (b) causes the exploration to fork on each possible hash bucket the value could fall into. A simple and effective optimization is to *neutralize the hash function*, i.e., replace it with a degenerate one returning a single constant. This change honors the usual contracts for hash functions (equal objects have equal hashes) and will turn hash lookups into list traversals.

Avoiding Symbolic Pointers Input-dependent pointers (also referred to as symbolic pointers) may point to multiple locations in the program memory, so a pointer dereference operation would have to be resolved for each possible location. In practice, symbolic execution engines deal with this situation in one of two ways: (a) fork the execution state for each possible concrete value the symbolic pointer can take; or (b) represent the dereference symbolically as a read operation from memory at a symbolic offset and let the constraint solver "deal" with it. Both ways hurt symbolic execution, either by causing excessive path explosion or by burdening the constraint solver.

While there is no generic way to avoid symbolic pointers other than concretizing their values (the `concretize` API call) at the price of losing completeness, there are specific cases where they can be avoided.

First, *the size of a buffer can be concretized* before allocation. A symbolic size would most likely cause a symbolic pointer to be returned, since a memory allocator computes the location of a new block based on the requested size. To avoid losing completeness, a symbolic execution-aware memory allocator can determine a (concrete) upper bound on the requested size and use that value for reserving space, while leaving the original size variable symbolic. This way, memory accesses to the allocated block would not risk being out of bounds. Figure 4.7 shows how the Chef API is used to wrap a call to the `malloc` function in the standard C library.

Second, *caching and "interning" can be eliminated*. Caching computed results and value interning (i.e., ensuring that a single copy of each possible value of a type is created) are common

```
void *malloc(size_t size) {
  if (is_symbolic(&size, sizeof(size))) {
    size_t upper_size = upper_bound(size);
    return old_malloc(upper_size);
  }
  return old_malloc(size);
}
```

Figure 4.7: Example of a symbolic execution-aware `malloc` function wrapper created using the Chef API. If the allocation size is symbolic, the wrapper determines its upper bound and issues a concrete request to the underlying implementation.

ways to improve the performance of interpreters. Alas, when a particular value is computed, its location in memory becomes dependent on its value. If the value was already in the cache or in the interned store, it is returned from there, otherwise a new value is computed. During symbolic execution, this logic becomes embedded in the value of the returned pointer, which becomes symbolic. Disabling caching and interning may hurt the native performance of the program, but it can give a significant boost when running inside a symbolic execution engine.

Avoiding Fast Paths A common way to speed-up the native performance of a function is to handle different classes of inputs using faster specialized implementations ("fast paths"). For example, a string comparison automatically returns false if the two strings have different lengths, without resorting to byte-wise comparison.

Fast paths may hurt symbolic execution because they cause symbolic branches in the code checking for the special input conditions. *Eliminating short-circuited returns* can reduce path explosion. Instead of returning to the caller as soon as it produced an answer, the function continues running and stops on an input-independent condition. For example, when comparing two strings of concrete length, a byte-wise string comparison would then traverse the entire string buffers in a single execution path, instead of returning after the first difference found.

4.7 Summary

Implementing and maintaining a symbolic execution engine is a significant engineering effort. It is particularly hard for interpreted dynamic languages, due to their rich semantics, rapid evolution, and lack of precise specifications. Chef provides an engine platform that is instantiated with a language interpreter, which implicitly defines the complete language semantics, and results in a correct and theoretically complete symbolic execution engine for the language. A language-agnostic strategy for selecting paths to explore in the interpreter allows the generated engine to systematically explore and test code in the target language effectively and efficiently. Chef is available at http://dslab.epfl.ch/proj/chef.

Chapter 5

Towards a Cloud Testing Service for PaaS

In this chapter, we present our ongoing work on a cloud-based automated testing service for PaaS applications. We first present the opportunities that computing clouds provide to developers, as well as the challenges developers face to test and secure their applications (Section 5.1). We then give an overview of the usage of our PaaS-based testing service (Section 5.2). The foundation of our service is a parallelization algorithm for symbolic execution that is the first to demonstrate linear scalability up to hundreds of cluster nodes (Section 5.4). On top, we introduce layered symbolic execution for addressing path explosion in systems composed of several stacked layers, such as protocol stacks (Section 5.3). Finally, we show how all pieces fit together in our S2E-based platform (Section 5.5).

5.1 The Opportunity of the Cloud Application Model

Modern consumer software is increasingly relying on the "cloud application" model, where a browser- or mobile device-based client interacts with a functionally rich cloud service. This model is prevalent in major systems like Facebook and GMail, as well as in smartphone and tablet "apps" like Instagram, Siri, or Dropbox. For both developers and users, the economics are highly attractive: cloud-based applications offer ubiquitous access, transparent scaling, and easy deployment at low cost.

The hidden cost is that cloud apps introduce security, privacy, and availability risks. They store and process critical information remotely, so the impact of failures is higher in this model than for single-user desktop apps [76]. This increases the importance of testing for cloud applications.

Rapid advances in development and deployment tools have significantly lowered the barrier to entry for developers, but these tools lack similarly advanced support for testing. Platform-as-a-service (PaaS) offerings, such as Google App Engine or Microsoft Azure, provide easy-to-use interfaces with automated deployment fully integrated into development environments. However,

testing tools for apps running on PaaS are still immature, test automation is limited, and developers are left with the laborious and error-prone task of manually writing large numbers of individual test cases.

Integration tests, which complement unit tests by checking that all the components of a fully deployed cloud application work together correctly, are especially tedious to write and set up in such an environment. PaaS-based cloud applications typically use frameworks with a layered communication architecture and perform some processing at each of the layers. Writing a full integration test then requires to carefully craft an HTTP request that successfully passes through all application and framework layers and triggers the desired behavior, while being transformed from one representation to another at each step (e.g., from an HTTP request to JSON to language objects).

Spending precious developer time on testing for improving security and reliability is particularly unattractive in a viciously competitive environment. Promoted by the ability to deploy virtually instantly and at low cost, the pressure is to use features to quickly acquire a large user base. Security and reliability testing is a long-term investment and does not pay off immediately, thus it is often deferred for later. High-profile failures of cloud applications [65, 73] are thus likely to become a common occurrence, unless we can make testing easy and cheap.

We argue that *testing* must become at least as easy as *deploying* a new app. PaaS has made the latter easy, leaving the former just as hard to do as before. A dedicated testing service must therefore become an integral component of PaaS. Recent commercial test automation systems (CloudBees, Skytap, SOASTA, etc.) relieve developers of managing their own test infrastructure, but still require them to write test suites; we aim to further relieve developers of having to write individual tests. Just as modern PaaS APIs spare developers from having to manage the building blocks of their applications, so should they spare developers from manually writing test cases, and instead offer means to automatically test apps based on only minimal input from developers.

The recent progress in automated test case generation and, in particular, symbolic execution takes an important step in this direction. For a typical PaaS application like the example in Figure 5.1, a test case generator could use symbolic execution to automatically find HTTP packets that drive the execution to different parts of the code. Unfortunately, these tools are challenging to apply to cloud apps: the number of execution paths successfully crossing all application layers is dwarfed by the vast number of possible error paths (dark vs. light arrows in Figure 5.1). Yet, most error paths are irrelevant to testing the high-level logic of the app itself (i.e., the innermost layer), because they test code that is part of the PaaS.

We leverage the modularity of each layer in modern web application stacks and introduce *layered parameterized tests* (LPTs) for integration testing of PaaS-based cloud applications (Section 5.2). LPTs describe families of integration tests (in the spirit of parameterized unit tests [88])

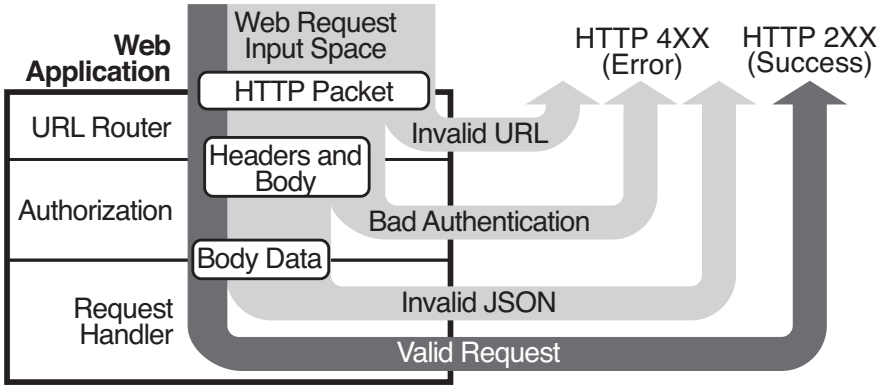

Figure 5.1: A sample cloud application. Clients communicate with the server through web requests that traverse several layers inside the server before reaching the request handler logic. From the total input space, most possible requests trigger errors in one of the processing layers (light arrows). Only a small fraction of requests successfully traverses all layers and is handled inside the app (dark arrow).

across several application layers. We rely on developer-provided *onion objects* to describe the layering of the data abstractions in the application; onion objects encode the multiple interpretations of input data as, e.g., an HTTP request, a JSON object, etc.

For the automatic generation of thorough test cases from LPTs, we introduce *layered symbolic execution* (LSE), an automated program analysis that is tailored for the layered structure of cloud applications (Section 5.3).

Symbolic execution is a resource-intensive task, so we horizontally scale it in the cloud by introducing the first symbolic execution parallelization algorithm to demonstrate linear scalability in the number of nodes (Section 5.4).

Finally, we present a design and early prototype for a PaaS-integrated parallel testing service based on LSE (Section 5.5).

5.2 A PaaS Test Interface

We introduce an automated testing service integrated in PaaS. Developers write layered parameterized tests (LPTs) and upload them with the cloud application to be executed by the test service. The service uses LPTs to automatically generate application inputs (e.g., web requests and persistent data) that exercise the application layers of interest. The developer writes LPTs by specifying the structure of the application inputs and a target property to be checked.

Developer Workflow Figure 5.2 illustrates the workflow of a developer using our testing service. The developer writes layered parameterized tests using a platform-provided testing API (step 1).

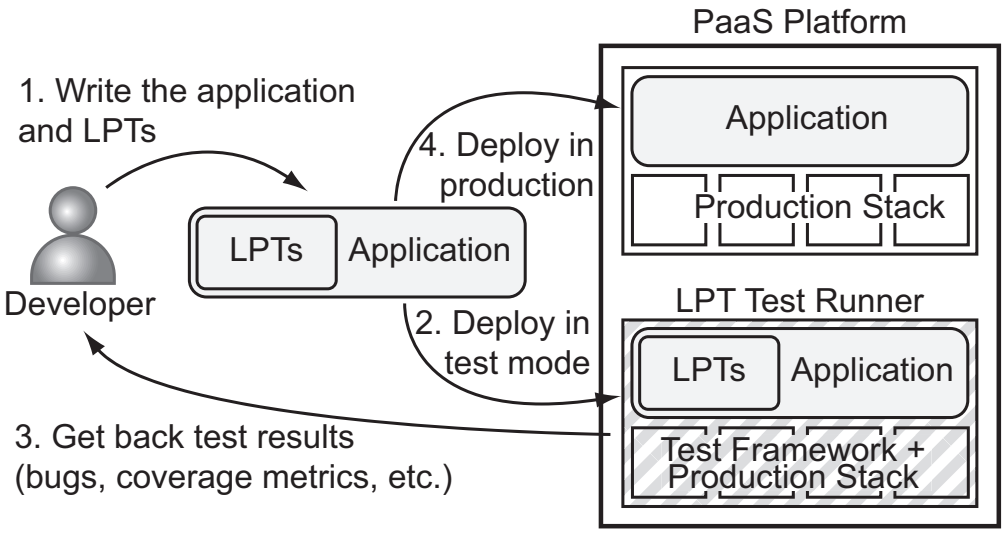

Figure 5.2: Development flow for using a PaaS-integrated testing service.

She then deploys the app in test mode, which invokes the LPT test runner of the PaaS (step 2). This test runner is responsible for generating and running the individual test cases from the LPT, and it returns test results back to the developer (step 3). The develop-deploy-test cycle continues until the code is ready to be deployed in production (step 4).

Layered Parameterized Tests An LPT specifies a family of executions (or, equivalently, classes of inputs) plus a set of properties (expressed as assertions) that are expected to hold for all these executions. The specified family of executions can be large or even infinite; still, the test runner can often efficiently check whether the property is guaranteed to hold for the entire family (we give more details on the symbolic execution-based mechanism in Section 5.3). A traditional unit test can be seen as a special case of an LPT for which the input is fixed and any provided assertions are checked on a single execution only.

LPTs are defined by developers using a platform-provided API in the implementation language of the application (e.g., Python). The testing API builds on the popular xUnit testing paradigm [13] and extends it with constructs to specify the structure of families of application inputs. The API is easily integrated with existing testing fixtures and frameworks that developers use today.

We illustrate the structure of an LPT and how it is used by the test runner using the example LPT `TestUpload` shown in Figure 5.3, which tests the upload functionality of a photo management application. Under the `/upload` URL, the application accepts POST requests containing a JSON-encoded dictionary describing photo information. Figure 5.4 shows an example HTTP request for the application.

We assume that the app follows the structure given in Figure 5.1, that it is written in Python using a popular web framework like Django [41] or WebApp [94], and that it is deployed on

```
1  from onion import LPT, instrumentLayer
2  from onion import str_, int_, dict_, http_, json_
3
4  app = PhotoUploadApp()
5
6  class TestUpload(LPT):
7      def setUp(self):
8          payload = dict_(
9              [("name", str_()), ("size", int_())],
10             layer="payload")
11         body = json_(payload, layer="body")
12         request = http_(body, layer="request")
13         self.defineInput(request, name="img_upload")
14
15     def runTest(self):
16         request = self.getInput("img_upload")
17         instrumentLayer(request, "request")
18         response = app.sendRequest(request)
19         self.assertNotEqual(response.status, 500)
20
21 class UploadHandler(RequestHandler):
22     def post(self):
23         instrumentLayer(self.request.body, "body")
24         payload = self.decodeJSON(self.request.body)
25         instrumentLayer(payload, "payload")
26
27         # ... process the request ...
```

Onion Object

Figure 5.3: An LPT example in Python for a photo management application. Highlighted code is the test code written by developers. Names in bold denote the LPT-specific API added to the standard Python unit test framework.

a PaaS infrastructure like Google App Engine [48] or Heroku [49]. The web framework takes care of dispatching any POST request to the **`/upload`** application URL to the **`post`** method of the **`UploadHandler`** class (lines 21–27). The **`TestUpload`** LPT checks that, for arbitrary upload requests, the server never returns an internal error (HTTP 500) in its response.

The test runner uses the LPT to automatically generate application input according to the following procedure:

1. The test runner invokes the **`setUp`** method (line 7), which declares the application inputs and their structure as *onion objects* (described below) using the **`defineInput`** call at line 13.

2. Based on the onion objects, the test runner generates a default input for the application.

3. The test runner invokes **`runTest`**, which retrieves the concrete input generated by the test runner with a call to **`getInput`** (line 16). In our example, the input is a web request, which is then sent to the application (line 18). Behind the scenes, the web framework dispatches the request to the **`post`** method of **`UploadHandler`** (line 22). When the handler finishes handling the request, a response object with a status code and body is returned. In our example, the LPT checks that no internal error (HTTP 500) occurred during request handling (line 19).

```
POST /upload HTTP/1.1
Host: photo.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Content-Type: application/json
Content-Length: 104446

{
  "name": "cat.jpg",
  "size": 104385,
  "data": "gZnJvbSBvdGhlciBhbmltYW[...]"
}
```

Figure 5.4: An example HTTP request for the upload feature of the photo management application. The text in bold denotes input processed by the application at various layers.

4. Based on the information collected during the execution of **`runTest`** (described below), the test runner uses the onion objects to generate a new application input (available to the LPT through the **`getInput`** call) and goes back to step 3 for a new iteration. Any assertion failures triggered by the generated inputs are reported to the developers.

Note that the generation and execution of multiple inputs is well suited for parallelization across multiple nodes in the cloud. This allows to leverage the availability of additional nodes to reduce the total testing time.

The test runner uses symbolic execution to generate new inputs (described in more detail in Section 5.3). To generate inputs that exercise the application at specific layers, the test runner needs:

- the unwrapped application inputs for the current execution at the different application layers, provided by developers through annotations in the application source code; and
- information about the input structure, provided by the LPT's onion objects.

Annotating Application Layers A web request traverses several processing layers in an application. First, it is received as an HTTP packet string; second, it is decoded into a URL, a set of headers, and request a body; third, the body contents is decoded and processed. Depending on the application framework, processing can involve additional layers, e.g., for converting JSON representations to language objects.

The application layers process data at corresponding layers of the input data (the bold parts of the HTTP request in Figure 5.4). For instance, the application typically maps the URL to a request handler, checks the headers for authentication information, and processes the body contents in the request handler code.

To expose the application input to the LPT as it is being processed at each layer, developers annotate the variables holding the input data structures in the application source code. Three layers have been declared in Figure 5.3: the HTTP request at line 17, the request body at line 23, and the JSON payload extracted from the body at line 25. The `instrumentLayer` call attaches a layer name to a variable. Similar to assertion statements, the call is active when executed as part of a test invocation, but disabled in production, where the LPTs are not used. For a typical web stack, only about three layers have to be annotated for each request handler, keeping the required effort on the developer side low.

Onion Objects An *onion object* is a data structure that describes the representations of the application input as it traverses multiple processing layers. The onion object (i) enables more convenient assertion-writing by directly exposing the data layers, and (ii) enables automated test generation to focus on specific layers of the application. Onion objects are needed to specify the application inputs for onion tests, but they can also be used to store output as the cloud application constructs a response in layers.

The framed area in Figure 5.3 shows the onion object for our running example. The structure consists of a set of *onion nodes* (the identifiers ending in an underscore) connected in a nested structure. There is one onion node for each layer and one for each input structure or value that is supposed to be generated automatically by the test engine. The abstraction level is declared using the `layer` parameter passed to the node constructor and matches one of the layers annotated in the code. Structures and values can be nested within the same layer. For example, the dictionary structure on lines 8–10 has constant keys and wildcard values of type `str_` and `int_`, which mimic the standard string and integer types.

Checking Properties LPTs express application properties through standard xUnit assertion statements (line 19 in the example). Through the dynamic test generation mechanism explained in Section 5.3, the test runner actively attempts to generate inputs that cause an assertion to be violated. Each generated test input not failing the assertion serves as a witness for an entire equivalence class of inputs that cannot violate the assertion. When an assertion does fail, the input that caused the failure is reported back to the developer.

To allow input variables at each layer to be used in assertions, each onion node offers a `value` property that refers to the value matched in the current test execution (not shown in the example).

5.3 Layered Symbolic Execution

In this section, we introduce *layered symbolic execution* (LSE), an LPT execution technique that focuses on covering a particular application layer. LSE uses symbolic execution—a test case generation technique that observes the program structure—to generate inputs in the representation of the layer of interest (e.g., HTTP headers or a JSON payload). Each generated layer-level input is then assembled back into application-level input based on the structure encoded in the onion object, in order to form an integration test case.

Naïve application of dynamic test generation to execute the LPT for a cloud app is of little use: First, the path exploration can end up exploring many different paths within the framework code, but might test only a single path within the application layer over and over again. Second, the path conditions will encode many branches due to the multiple layers of parsing logic, making symbolic execution of cloud apps prohibitively expensive. Third, if the exploration is unaware of the connections between abstraction layers, blindly negating just single branch conditions will produce many infeasible paths before finding a new valid test input.

LSE and Onion Objects LSE relies on onion objects to mark input variables as symbolic and generate new values based on the alternate path conditions. To this end, each onion object exposes a number of operations:

- `instrument(var)` instruments the variable `var` for symbolic execution, i.e., injects a fresh symbolic input value in dynamic test generation. The variable is expected to match the structure described by the onion object.
- `reconstruct(var, val)` applies an assignment of value `val` to variable `var` that is demanded by the satisfying assignment representing a new path. In doing the assignment, the function performs the necessary modifications to other variables to respect the cross-layer invariants.
- `getDefault()` returns a default value for the object node. It is used for generating the initial test case or any padding values required by invariants (e.g., changing the content length field of an HTTP request requires to extend the actual contents).

For example, applying the `instrument` method of a string onion object on a string variable in Python marks as symbolic the string length and the contents of the character buffer. Then, during symbolic execution, the alternate path constraint yields new values for the length and for the buffer. The `reconstruct` method takes both values and creates a new Python string object.

The reconstruction method is essential for enforcing the object and cross-layer invariants of the input structure. For instance, the length of the reconstructed string would always match the size

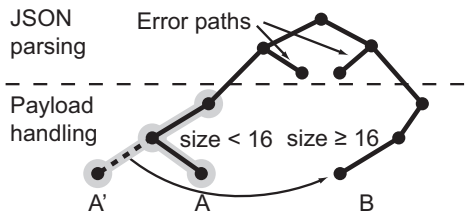

Figure 5.5: Exploring an execution tree using layered symbolic execution.

of its buffer (and avoid spurious overflows); the Content-length HTTP header would always match the size of the new request body, and so on.

LSE Algorithm LSE allows the test runner to focus on exploring paths inside inner application layers. Conceptually, LSE decouples the input layers to give the test runner the flexibility to freely explore an individual layer. When constructing a new application input, LSE reconnects the layers, taking care to respect cross-layer invariants (e.g., the value of a JSON field has to be present also in the HTTP packet). The LSE algorithm proceeds along the following steps:

1. Generate an initial valid input (i.e., a web request) using the `getDefault` call on the root node. The LPT can read this input by calling `getInput`.

2. Symbolically execute the program through the test (the `runTest` method), using symbolic inputs created by calling the `instrument` method on the onion object nodes corresponding to the layer of interest. Any existing symbolic expressions for these variables (which implicitly encode the parsing logic) are overwritten in this step, effectively decoupling the input at the current layer from the previous ones. This permits the symbolic execution engine to negate constraints inside the current layer without being constrained by the previous layers.

3. When the execution completes, negate a constraint in the path condition to obtain new values for the onion nodes.

4. Using the `reconstruct` function of the onion object node, assemble the new values back into a new complete program input (e.g., the HTTP request) for the next iteration.

Figure 5.5 illustrates an execution tree explored in an iteration of LSE. Consider an initial input for the example in Figure 5.3, where the value of the `size` field in the JSON request payload is 8 (Path *A* in the figure). At step 2 of the algorithm, a symbolic value is injected for `size`, together with the rest of the onion object wildcard fields (the highlighted segment of Path *A*). Now, if the tested path contains the conditional statement `if payload.size < 16`, the `then` branch of the statement is taken and the $size < 16$ constraint is recorded. At the end of the execution (step 3), if this constraint is negated to $size \geq 16$, a new value for `size` is generated, say 20 (the alternate

potential Path A'). Then, at step 4, the `reconstruct` functions assembles the new values of all leaves into a new HTTP packet to be sent to the app, which will cause the `else` branch of the `if` statement to be taken in the next execution (Path B). Note that Path A' is not globally feasible and never explored, but only transiently used to produce the feasible Path B.

Compared to a solution that only marks the variables at the layers of interest as symbolic, LSE is superior in two ways: (1) By obtaining the root input, it is able to run integration tests for a fully deployed application; (2) LSE supports data structures of variable sizes, e.g., arrays whose lengths are symbolic values, by regenerating the input structure at each new iteration.

5.4 Parallelizing Symbolic Execution on Commodity Clusters

Since symbolic execution is a resource-intensive task, our PaaS testing service parallelizes it on a cloud of commodity machines. LPTs submitted to the testing service generate symbolic execution jobs that are distributed across a number of cloud nodes that is proportional to the testing thoroughness specified by the user at job creation time.

The key design goal is to enable individual cluster nodes to explore the execution tree independently of each other. One way of doing this is to statically split the execution tree and farm off subtrees to worker nodes. Alas, the contents and shape of the execution tree are not known until the tree is actually explored, and finding a balanced partition (i.e., one that will keep all workers busy) of an unexpanded execution tree is undecidable. Besides subtree size, the amount of memory and CPU required to explore a subtree is also undecidable, yet must be taken into account when partitioning the tree. Since the methods used so far in parallel model checkers [50, 10] rely on static partitioning of a finite state space, they cannot be directly applied to the present problem. Instead, our approach partitions the execution tree *dynamically*, as the tree is being explored.

Dynamic Distributed Exploration The testing service cluster consists of worker nodes and a load balancer (LB). Workers run independent symbolic execution engines that explore portions of the execution tree and send statistics on their progress to the LB, which in turn instructs, whenever necessary, pairs of workers to balance each other's work load. Encoding and transfer of work is handled directly between workers, thus taking the load balancer off the critical path.

The goal is to dynamically partition the execution tree such that the parts are *disjoint* (to avoid redundant work) and together they *cover* the global execution tree (for exploration to be complete). We aim to minimize the number of work transfers and associated communication overhead. A fortuitous side effect of dynamic partitioning is the transparent handling of fluctuations in resource quality, availability, and cost, which are inherent to large clusters in cloud settings.

The system operates roughly as follows: The first component to come up is the load balancer.

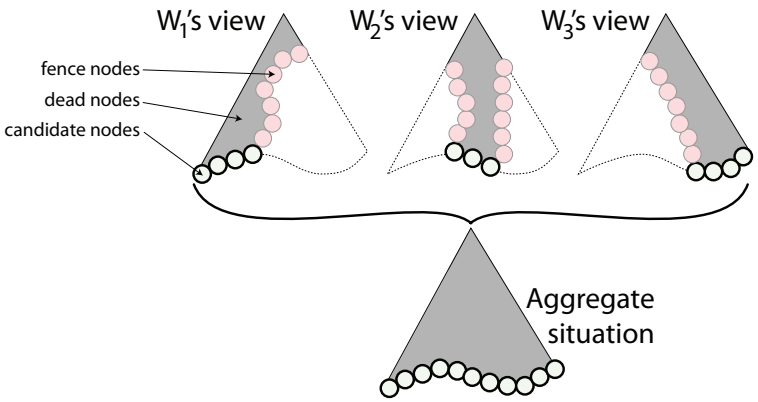

Figure 5.6: Dynamic partitioning of exploration in the testing service.

When the first worker node W_1 joins the cluster, it connects to the LB and receives a "seed" job to explore the entire execution tree. When the second worker W_2 joins and contacts the LB, it is instructed to balance W_1's load, which causes W_1 to break off some of its unexplored subtrees and send them to W_2 in the form of *jobs*. As new workers join, the LB has them balance the load of existing workers. The workers regularly send to the LB status updates on their load in terms of exploration jobs, along with current progress in terms of code coverage, encoded as a bit vector. Based on workers' load, the LB can issue job transfer requests to pairs of workers in the form $\langle$ source worker, destination worker, # of jobs $\rangle$. The source node decides which particular jobs to transfer.

5.4.1 Worker-level Operation

A worker's visibility is limited to the subtree it is exploring locally. As W_i explores and reveals the content of its local subtree, it has no knowledge of what W_j's ($i \neq j$) subtree looks like. No element in the system—not even the load balancer—maintains a global execution tree. Disjointness and completeness of the exploration (see Figure 5.6) are ensured by the load balancing algorithm.

As will be explained later, each worker has the root of the global execution tree. The tree portion explored thus far on a worker consists of three kinds of nodes: (1) internal nodes that have already been explored and are thus no longer of interest—we call them *dead* nodes; (2) *fence* nodes that demarcate the portion being explored, separating the domains of different workers; and (3) *candidate* nodes, which are nodes ready to be explored. A worker exclusively explores candidate nodes; it never expands fence or dead nodes.

Candidate nodes are leaves of the local tree, and they form the *exploration frontier*. The work transfer algorithm ensures that frontiers are disjoint between workers, thus ensuring that no worker duplicates the exploration done by another worker. At the same time, the union of all frontiers in the system corresponds to the frontier of the global execution tree. The goal of a worker W_i at every

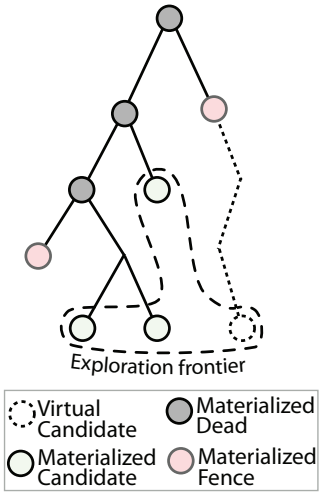

Figure 5.7: Example of a local worker tree.

step is to choose the next candidate node to explore and, when a bug is encountered, to compute the inputs, thread schedule, and system call returns that would take the program to that bug.

Worker-to-Worker Job Transfer When the global exploration frontier becomes poorly balanced across workers, the load balancer chooses a loaded worker W_s and a less loaded worker W_d and instructs them to balance load by sending n jobs from W_s to W_d. In the extreme, W_d is a new worker or one that is done exploring its subtree and has zero jobs left.

W_s chooses n of its candidate nodes and packages them up for transfer to W_d. Since a candidate node sent to another worker is now on the boundary between the work done by W_s and the work done by W_d, it becomes a fence node at the sender. This conversion prevents redundant work.

A job can be sent in at least two ways: (1) serialize the content of the chosen node and send it to W_d, or (2) send to W_d the path from the tree root to the node, and rely on W_d to "replay" that path and obtain the contents of the node. Choosing one vs. the other is a trade-off between time to encode/decode and network bandwidth: option (1) requires little work to decode, but consumes bandwidth (the state of a real program is typically at least several megabytes), while encoding a job as a path requires replay on W_d. We assume that large commodity clusters have abundant CPU but meager bisection bandwidth, so we chose to encode jobs as the path from the root to the candidate node. As an optimization, we exploit common path prefixes: jobs are not encoded separately, but rather the corresponding paths are aggregated into a job tree and sent as such.

When the job tree arrives at W_d, it is imported into W_d's own subtree, and the leaves of the job tree become part of W_d's frontier (at the time of arrival, these nodes may lie "ahead" of W_d's frontier). W_d keeps the nodes in the incoming jobs as *virtual* nodes, as opposed to *materialized* nodes that reside in the local subtree, and replays paths only lazily. A materialized node is one

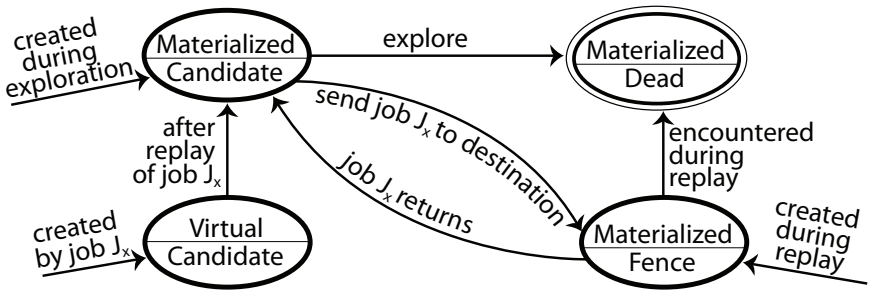

Figure 5.8: Transition diagram for nodes in a worker's subtree.

that contains the corresponding program state, whereas a virtual node is an "empty shell" without corresponding program state. In the common case, the frontier of a worker's local subtree contains a mix of materialized and virtual nodes, as shown in Figure 5.7.

As mentioned earlier, a worker must choose at each step which candidate node to explore next—this choice is guided by a *strategy*. Since the set of candidate nodes now contains both materialized and virtual nodes, it is possible for the strategy to choose a virtual node as the next one to explore. When this happens, the corresponding path in the job tree is replayed (i.e., the symbolic execution engine executes that path); at the end of this replay, all nodes along the path are dead, except the leaf node, which has converted from virtual to materialized and is now ready to be explored. Note that, while exploring the chosen job path, each branch produces child program states; any such state that is not part of the path is marked as a fence node, because it represents a node that is being explored elsewhere, so W_d should not pursue it.

Summary A node N in W_i's subtree has two attributes, $N^{\text{status}} \in \{\text{materialized, virtual}\}$ and $N^{\text{life}} \in \{\text{candidate, fence, dead}\}$. A worker's frontier F_i is the set of all candidate nodes on worker W_i. The worker can only explore nodes in F_i, i.e., dead nodes are off-limits and so are fence nodes, except if a fence node needs to be explored during the replay of a job path. The union $\cup F_i$ equals the frontier of the global execution tree, ensuring that the aggregation of worker-level explorations is complete. The intersection $\cap F_i = \emptyset$, thus avoiding redundancy by ensuring that workers explore disjoint subtrees. Figure 5.8 summarizes the life cycle of a node.

As suggested in Figure 5.8, once a tree node is dead, it has reached a terminal state; therefore, a dead node's state can be safely discarded from memory. This enables workers to maintain program states only for candidate and fence nodes.

5.4.2 Cluster-level Operation

Load Balancing When jobs arrive at W_d, they are placed conceptually in a queue; the *length* of this queue is sent to the load balancer periodically. The LB ensures that the worker queue lengths

stay within the same order of magnitude. The balancing algorithm takes as input the lengths l_i of each worker W_i's queue Q_i. It computes the average $\bar{l}$ and standard deviation σ of the l_i values and then classifies each W_i as underloaded ($l_i < max\{\bar{l} - \delta \cdot \sigma, 0\}$), overloaded ($l_i > \bar{l} + \delta \cdot \sigma$), or OK otherwise; δ is a constant factor. The W_i are then sorted according to their queue length l_i and placed in a list. LB then matches underloaded workers from the beginning of the list with overloaded workers from the end of the list. For each pair $\langle W_i, W_j \rangle$, with $l_i < l_j$, the load balancer sends a job transfer request to the workers to move $(l_j - l_i)/2$ candidate nodes from W_j to W_i.

Coordinating Worker-level Explorations Classic symbolic execution relies on heuristics to choose which state on the frontier to explore first, so as to efficiently reach the chosen test goal (code coverage, finding a particular type of bug, etc.). In a distributed setting, local heuristics must be coordinated across workers to achieve the global goal, while keeping communication overhead at a minimum. What we have described so far ensures that eventually all paths in the execution tree are explored, but it provides no aid in focusing on the paths desired by the global strategy. In this sense, what we described above is a *mechanism*, while the exploration strategies represent the *policies*.

Global strategies are implemented in the testing service using its interface for building *overlays* on the execution tree structure. We used this interface to implement distributed versions of all strategies that come with KLEE [25]; the interface is also available to the testing service. Due to space limitations, we do not describe the strategy interface further, but provide below an example of how a global strategy is built.

A coverage-optimized strategy drives exploration so as to maximize coverage [25]. In our testing service, coverage is represented as a bit vector, with one bit for every line of code; a set bit indicates that a line is covered. Every time a worker explores a program state, it sets the corresponding bits locally. The current version of the bit vector is piggybacked on the status updates sent to the load balancer. The LB maintains the current global coverage vector and, when it receives an updated coverage bit vector, ORs it into the current global coverage. The result is then sent back to the worker, which in turn ORs this global bit vector into its own, in order to enable its local exploration strategy to make choices consistent with the global goal. The coverage bit vector is an example of an overlay data structure in our testing service.

5.5 A Testing Platform for the Cloud

We deploy layered symbolic execution on a cluster of symbolic execution-aware virtual machines (the symbolic VMs). Unlike a regular (e.g., x86) virtual machine, a symbolic virtual machine can mark parts of its memory as symbolic and fork its state (CPU registers, memory, etc.) at symbolic

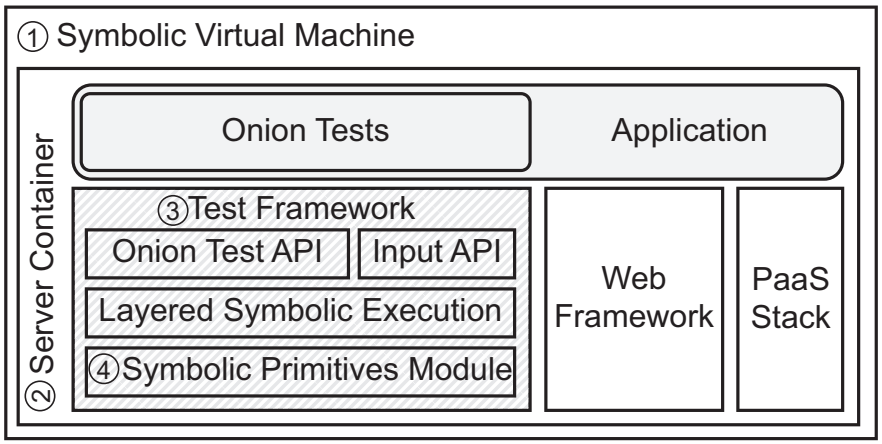

Figure 5.9: The PaaS test runner service.

branch instructions. A symbolic VM encapsulates the "entire universe" of the application, including the framework and even the language interpreter and operating system, enabling integration testing of the entire application stack.

The test-mode deployment then requires just the push of a button for the system to execute the layered parameterized tests, generate coverage statistics, and highlight any failing test cases.

This deployment model leverages the properties of PaaS in several ways: (1) By hiding the testing VMs behind a service interface, the PaaS system can faithfully reproduce the exact environment of production VMs inside the testing VMs without exposing its internals. (2) The testing task can be transparently scaled across multiple VMs by using parallel symbolic execution. (3) Since the application uses standard interfaces for accessing the PaaS components (storage, networking, etc.), the provider is able to substitute production-optimized implementations with testing-optimized stubs that offer a simplified behavior that is better suited to automated program analysis.

From the perspective of the PaaS provider, the test runner service consists of a set of symbolic VMs, operated separately from the production infrastructure. When an application is deployed in test mode, one of the symbolic VMs is allocated for testing: the application code and tests are copied to the guest, and the LSE algorithm is invoked.

Architecture Figure 5.9 illustrates the architecture of the symbolic VM environment. Inside the VM①, all application components are symbolically executed in the same low-level representation (e.g., x86 machine code or LLVM [62]). The components execute inside their own vanilla interpreters②. The test framework③ plays two roles: it implements (1) the APIs for LPTs and onion objects that developers use to write the testing code, and (2) the LSE algorithm that guides the test case generation.

Prototype We implemented a prototype of the symbolic VM that tests Python-based Google App Engine PaaS applications and is built on top of the S2E symbolic virtual machine.

In our implementation, the symbolic execution engine and the LSE logic live at different levels in the symbolic VM stack. The symbolic execution engine operates with low-level abstractions such as memory bytes. It resides on the host level, as an S2E plugin that exposes the core symbolic execution primitives to the guest as S2E system calls, e.g., to allow marking memory buffers as symbolic. The LSE algorithm operates on web application state (e.g., by accessing the onion objects), and is implemented in the guest as a native Python extension module. We implemented LSE on top of WebTest [95], a popular fixture library for testing Python web applications. The resulting system is extensible to other languages with limited engineering effort: since the symbolic execution logic is provided at the host level, only the test framework component needs to be implemented in the cloud app language.

Early experiences with the prototype are encouraging: for the full application stack of a simple cloud app, our prototype generates a test case every few seconds.

5.6 Summary

This chapter presented the vision of an automated testing platform that changes the way software testing has been traditionally done, by providing a cloud-based "in-vivo" testing service of an application in the same environment as the one used in production. The testing platform is based on symbolic execution. It leverages the modularity of modern web application stacks to scale symbolic execution, by using layered parameterized tests (LPTs). In LPTs, developers write onion objects that concisely describe the nested structure of the application input. LPTs are executed using layered symbolic execution (LSE), which navigates the complexity of the application stack and avoids exploring errors paths in irrelevant application layers. We parallelize the symbolic execution in clouds of commodity hardware, using an algorithm that demonstrates linear scalability.

Chapter 6

Evaluation

In this chapter, we evaluate the research ideas described in the thesis. We built prototypes for Cloud9 (Section 6.1) and Chef (Section 6.2), which we use to demonstrate the effectiveness and efficiency of their techniques. After presenting our methodology (Section 6.3), we structure the evaluation by answering the following questions:

1. Can Chef and Cloud9 symbolically execute real-world software (Section 6.4)?

2. Are the symbolic execution engines effective for bug finding and test suite generation (Section 6.5)?

3. Given the overhead of using the language interpreter, how efficient is Chef's test suite generation (Section 6.6)?

4. Does parallel symbolic execution, which our testing service builds upon, scale on commodity clusters (Section 6.7)?

6.1 The Cloud9 Prototype

We developed a Cloud9 prototype on top of the KLEE [25] symbolic execution engine. The prototype has 7 KLOC. The KLEE modifications to support the symbolic OS abstractions amount to roughly 2 KLOC, while the rest consists of the POSIX model built on top of the abstractions. We also implemented support for parallelizing symbolic execution on a cluster of nodes (Section 5.4), which we use to demonstrate the scalability of the parallelization algorithm (Section 6.7). The Cloud9 prototype is available at http://cloud9.epfl.ch.

Cloud9 builds upon the KLEE symbolic execution engine, and so it inherits from KLEE the mechanism for replacing parts of the C Library with model code; it also inherits the external calls

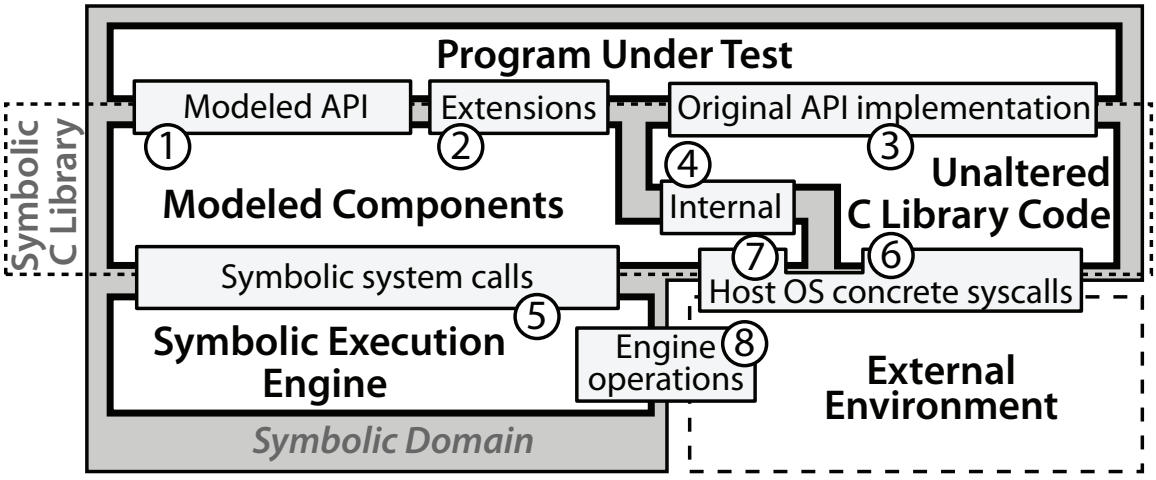

Figure 6.1: Architecture of the Cloud9 POSIX model.

mechanism. Cloud9 adds the symbolic system call interface and replaces parts of the C Library with the POSIX model. The resulting architecture is shown in Figure 6.1.

Before symbolic execution starts, the Cloud9 system links the program under test with a special symbolic C Library. We built this library by replacing parts of the existing uClibc library in KLEE with the POSIX model code. Developers do not need to modify the code of to-be-tested programs in any way to make it run on Cloud9.

In the C Library, we replaced operations related to threads, processes, file descriptors, and network operations with their corresponding model ①, and augmented the API with Cloud9-specific extensions ②. A large portion of the C Library is reused, since it works out of the box ③ (e.g. memory and string operations). Finally, parts of the original C Library itself use the modeled code ④ (e.g., Standard I/O `stdio` relies on the modeled POSIX file descriptors).

The modeled POSIX components interface with the SEE through symbolic system calls ⑤, listed in Table 3.1 from Section 3.1.3. Occasionally, the unmodified part of the C Library invokes external system calls ⑥, and the model code itself needs support from the host OS ⑦—in order to make sure the external calls do not interfere with the symbolic engine's own operations ⑧, such access is limited to read-only and/or stateless operations. This avoids problems like, for instance, allowing an external `close()` system call to close a network connection or log file that is actually used by the SEE itself.

KLEE uses copy-on-write (CoW) to enable memory sharing between symbolic states. We extend this functionality to provide support for multiple address spaces. We organize the address spaces in an execution state as *CoW domains* that permit memory sharing between processes. A memory object marked as shared by calling `cloud9_make_shared` is automatically mapped in the address spaces of the other processes within the CoW domain. Whenever a shared object is modified in one address space, the new version is automatically propagated to the other members

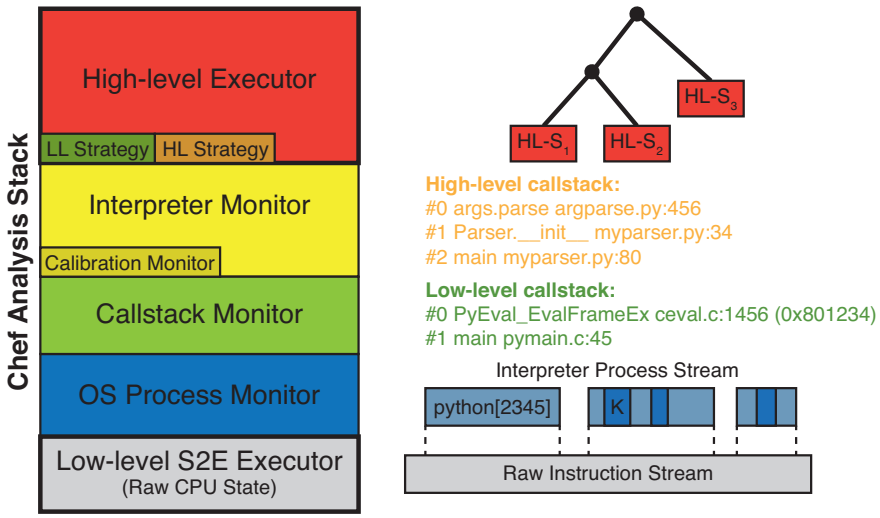

Figure 6.2: The implementation of Chef, as a stack of S2E analysis modules that refine the raw stream of x86 instructions into a high-level symbolic execution view.

of the CoW domain.

6.2 The Chef Prototype and Case Studies

In this section, we present our Chef prototype, along with our experience preparing two symbolic execution engines, one for Python and the other for Lua.

Implementation We implemented Chef on top of the S2E analysis platform [33], which is a symbolic virtual machine that executes symbolically entire guests. The implementation consists of a stack of dynamic analysis modules that refine the raw stream of x86 instructions at the CPU level into high-level program statements forming a symbolic execution tree (Figure 6.2).

First, the OS Monitor module breaks down the raw stream of x86 instructions into processes, and separates the user from the kernel mode. The module cooperates with an instrumented Linux kernel in the VM, which reports all running threads, their creation, and termination. To detect context switches, the OS Monitor tracks the value of the x86 CR3 page table register. To detect user/kernel mode switches, the OS Monitor tracks the privilege level (ring) in the CPU. The analysis modules above the OS Monitor look only at the user-mode instructions of the interpreter process.

Next, the Callstack Monitor tracks the `call` and `ret` instructions in the interpreter to maintain the low-level call stack. On top, the Interpreter Monitor uses the low-level call stack and the interpreter HLPC slice information (Section 4.6.1) to maintain the high-level HLPC stack. The Interpreter Monitor also computes the HLPC slice when the interpreter runs in calibration mode.

Component	Python	Lua
Interpreter core size (C LoC)	427,435	14,553
Symbolic optimizations (C LoC)	274 (0.06%)	233 (1.58%)
Native extensions (C LoC)	1,320 (0.31%)	154 (1.06%)
Test library (Python/Lua LoC)	103	87
Developer time (person-days)	5	3

Table 6.1: Summary of the effort required to support Python and Lua in Chef. The first row is the interpreter size without the standard language library. The next row shows changes in the interpreter core, while the following two constitute the symbolic test library. The last row indicates total developer effort.

Finally, the top High-level Executor module aggregates the HLPC information from all low-level execution states to maintains the high-level symbolic execution tree.

Case Studies We used Chef to generate symbolic execution engines for Python (Section 6.2.1) and Lua (Section 6.2.2). Table 6.1 summarizes the effort to set up the two interpreters for Chef. The necessary changes to the interpreter amount to 274 lines of code for Python and 233 for Lua. The total developer time was 5 person-days for Python and 3 person-days for Lua, which is orders of magnitude smaller than the effort required for building a complete symbolic execution engine from scratch.

6.2.1 Symbolic Execution Engine for Python

Interpreter Preparation We instrumented the CPython interpreter 2.7.3 for use with Chef, according to the guidelines presented in Section 4.6.2.

Python programs are composed of modules, corresponding to Python source files. Before executing a module, the interpreter compiles its source into an interpreter-specific bytecode format, i.e., each source statement is translated into one or more lower-level primitive instructions. The instructions are grouped into blocks, corresponding to a single loop nesting, function, method, class, or global module definition. Chef automatically detects the HLPC variable pointing to the bytecode blocks, by using the HLPC slice reconstruction procedure in Section 4.6.1. We cross-checked the correctness of the obtained slice by looking it up in the interpreter source code, via the debug symbols in the binary.

We performed several optimizations on the Python interpreter (Section 4.6.2): we neutralized the hash functions of strings and integers, which are the most common objects; we concretized the memory sizes passed to the garbage-collected memory allocator; and we eliminated interning for small integers and strings. Most optimizations involved only adding preprocessor directives for conditional compilation of blocks of code. We gathered the optimizations under a new

```
class ArgparseTest(SymbolicTest):
    def setUp(self):
        self.argparse = importlib.import_module("argparse")

    def runTest(self):
        parser = self.argparse.ArgumentParser()
        parser.add_argument(
            self.getString("arg1_name", '\x00'*3))
        parser.add_argument(
            self.getString("arg2_name", '\x00'*3))

        args = parser.parse_args([
           self.getString("arg1", '\x00'*3),
           self.getString("arg2", '\x00'*3)])
```

Figure 6.3: The symbolic test used to exercise the functionality of the Python **`argparse`** package.

`-with-symbex` flag of the interpreter's **`./configure`** script.

Symbolic Tests To validate the usefulness of the resulting symbolic execution engine, we use it as a test case generation tool. To this end, we implemented a symbolic test library as a separate Python package, used both inside the guest virtual machine, and outside, during test replay. Figure 6.3 is an example of a symbolic test class for the **`argparse`** command-line interface generator. It sets up a total of 12 symbolic characters of input: two 3-character symbolic arguments to configure the command-line parser plus another two to exercise the parsing functionality. We choose three characters for each command line argument to minimally cover the three types of arguments: short option (e.g., **`-h`**), long option (e.g., **`--x`**), and positional (e.g., **`xyz`**).

The test class derives from the library's **`SymbolicTest`** class, which provides two methods to be overridden: **`setUp`**, which is run once before the symbolic test starts, and **`runTest`**, which creates the symbolic input and can check properties. The symbolic inputs are created by calling the **`getString`** and **`getInt`** methods in the **`SymbolicTest`** API.

A symbolic test is executed by a symbolic test runner, which is also part of the library. The runner can work in either symbolic or replay mode. In *symbolic mode*, the runner executes inside the guest virtual machine. It creates a single instance of the test class, whose **`getString`** and **`getInt`** methods create corresponding Python objects and invoke the **`make_symbolic`** call to mark their memory buffers as symbolic. In *replay mode*, the runner creates one instance of the test class for each test case created by Chef. The **`getString`** and **`getInt`** methods return the concrete input assignment of the test case.

6.2.2 Symbolic Execution Engine for Lua

Lua is a lightweight scripting language mainly used as an interpreter library to add scripting capabilities to software written in other languages. However, it also has a stand-alone interpreter

and several Lua-only projects exist. We generated a symbolic execution engine for Lua based on version 5.2.2 of the Lua interpreter.

Interpreter Instrumentation Similar to Python, Lua programs are composed of one or more Lua source files, compiled into a bytecode format. The code is compiled into a set of functions that operate on a global stack of values. Each function is composed of a sequence of bytecode instructions, where each instruction is defined by an offset, opcode, and parameters. We automatically detect updates to the HLPC variable pointing to bytecode instructions. We cross-checked the correctness of the interpreter HLPC slice using the debug symbols in the binary.

We optimized the Lua interpreter for symbolic execution by eliminating string interning. In addition, we configured the interpreter to use integer numbers instead of the default floating point, for which S2E does not support symbolic expressions. This change was easy, because it was available as a macro definition in the interpreter's configuration header.

6.3 Methodology

Hardware Configuration For all our Cloud9 experiments, we ran Cloud9 in parallel symbolic execution mode in a heterogeneous cluster environment, with worker CPU frequencies between 2.3–2.6 GHz and with 4–6 GB of RAM available per core.

We performed all Chef experiments on a 48-core 2.3 GHz AMD Opteron 6176 machine with 512 GB of RAM, running Ubuntu 12.04. Each Chef invocation ran on 1 CPU core and used up to 8 GB of RAM on average.

Coverage Measurement Line or statement coverage remains widely used, even though its meaningfulness as a metric for test quality is disputed. We measure and report line coverage to give a sense of what users can expect from a test suite generated fully automatically by Cloud9 or a symbolic execution engine based on Chef. For Python, we rely on the popular `coverage` package, and for Lua we use the `luacov` package.

Since our Chef prototype only supports strings and integers as symbolic program inputs, we count only the lines of code that can be reached using such inputs. We report this number as "coverable LOC" in the fifth column of Table 6.3, and use it in our experiments as a baseline for what such a symbolic execution engine could theoretically cover directly. For example, for the `simplejson` library, this includes only code that decodes JSON-encoded strings, not code that takes a JSON object and encodes it into a string. Note that, in principle, such code could still be tested and covered by writing a more elaborate symbolic test that sets up a JSON object based on symbolic primitives [21].

System	Size (KLOC)	Type of Software
Apache httpd 2.2.16	226.4	Web servers
Lighttpd 1.4.28	39.5	
Ghttpd 1.4.4	0.6	
Memcached 1.4.5	8.3	Distributed object cache
Python 2.6.5	388.3	Language interpreter
Curl 7.21.1	65.9	Network utilities
Rsync 3.0.7	35.6	
Pbzip 2.1.1	3.6	Compression utility
Libevent 1.4.14	10.2	Event notification library
Coreutils 6.10	72.1	Suite of system utilities
Bandicoot 1.0	6.4	Lightweight DBMS

Table 6.2: Representative selection of testing targets that run on Cloud9. Size was measured using the `sloccount` utility.

Execution Strategies in Cloud9 On each worker, the underlying KLEE engine used the best execution strategies from [25], namely an interleaving of random-path and coverage-optimized strategies. At each step, the engine alternately selects one of these heuristics to pick the next state to explore. Random-path traverses the execution tree starting from the root and randomly picks the next descendant node, until a candidate state is reached. The coverage-optimized strategy weighs the states according to an estimated distance to an uncovered line of code, and then randomly selects the next state according to these weights.

6.4 Testing Targets

6.4.1 Testing Low-level Systems with Cloud9

Table 6.2 shows a selection of the systems we tested with Cloud9, covering several types of software. We confirmed that each system can be tested properly under our POSIX model. In the rest of this section, we focus our in-depth evaluation on several networked servers and tools, as they are frequently used in settings where reliability matters.

Due to its comprehensive POSIX model, Cloud9 can test many kinds of servers. One example is lighttpd, a web server used by numerous high-profile web sites, such as YouTube, Wikimedia, Meebo, and SourceForge. For lighttpd, Cloud9 proved that a certain bug fix was incorrect, and the bug could still manifest even after applying the patch (Section 6.5.3). Cloud9 also found a bug in curl, an Internet transfer application that is part of most Linux distributions and other operating systems (Section 6.5.1). Cloud9 also found a hang bug in the UDP handling code of memcached, a distributed memory object cache system used by many Internet services, such as Flickr, Craigslist,

Package	LOC	Type	Description	Coverable LOC	Exceptions	Hangs
Python						
argparse*	1,466	System	Command-line interface	1,174	4 / 0	—
ConfigParser*	451	System	Configuration file parser	145	1 / 0	—
HTMLParser*	623	Web	HTML parser	582	1 / 0	—
simplejson 3.10	1,087	Web	JSON format parser	315	2 / 0	—
unicodecsv 0.9.4	126	Office	CSV file parser	95	1 / 0	—
xlrd 0.9.2	7,241	Office	Microsoft Excel reader	4,914	5 / 4	—
Lua						
cliargs 2.1-2	370	System	Command-line interface	273	—	—
haml 0.2.0-1	984	Web	HTML description markup	775	—	—
sb-JSON v2007	454	Web	JSON format parser	329	—	✓
markdown 0.32	1,057	Web	Text-to-HTML conversion	673	—	—
moonscript 0.2.4-1	4,634	System	Language compiler	3,577	—	—
TOTAL	18,493			12,852		

Table 6.3: Summary of testing results for the Python and Lua packages used for evaluation. Items with (*) represent standard library packages. Exception numbers indicate total / undocumented exception types discovered.

Twitter, and Livejournal (Section 6.5.2).

In addition to the testing targets mentioned above, we also tested a benchmark consisting of a multi-threaded and multi-process producer-consumer simulation. The benchmark exercises the entire functionality of the POSIX model: threads, synchronization, processes, and networking.

We conclude that Cloud9 is practical and capable of testing a wide range of real-world software systems.

6.4.2 Testing Python and Lua Packages with Chef

We evaluated the symbolic execution engines for Python and Lua on 6 Python and 5 Lua packages, respectively, including system, web, and office libraries. In total, the tested code in these packages amounts to about 12.8 KLOC. We chose the latest versions of widely used packages from the Python standard library, the Python Package Index, and the Luarocks repository. Whenever possible, we chose the pure interpreted implementation of the package over the native optimized one (e.g., the Python `simplejson` package). The first five columns of Table 6.3 summarize the package characteristics; LOC numbers were obtained with the `cloc` tool [2].

The reported package sizes exclude libraries, native extension modules, and the packages' own test suites. However, the packages ran in their unmodified form, using all the language features and libraries they were designed to use, including classes, built-in data structures (strings, lists, dictionaries), regular expressions, native extension modules, and reflection.

All testing targets have a significant amount of their functionality written in the interpreted

language itself; we avoided targets that are just simple wrappers around native extension modules (written in C or C++) in order to focus on the effectiveness of Chef at distilling high-level paths from low-level symbolic execution. Nevertheless, we also included libraries that depend on native extension modules. For instance, all the testing targets containing a lexing and parsing component use Python's standard regular expression library, which is implemented in C. To thoroughly test these parsers, it is important to also symbolically execute the native regular expression library. For this, the binary symbolic execution capabilities of Chef are essential.

Symbolic Tests For each package, we wrote a symbolic test that invokes the package's entry points with one or more symbolic strings. Figure 6.3 in Section 6.2.1 is an example of such a symbolic test.

Each symbolic test ran for 30 minutes within Chef, after which we replayed the collected high-level tests on the host machine, in a vanilla Python/Lua environment, to confirm test results and measure line coverage. To compensate for the randomness in the state selection strategies, we repeated each experiment 15 times. In each graph we present average values and error margins as +/- one standard deviation.

For our experiments, we did not use explicit specifications, but relied on generic checks for finding common programming mistakes. For both Python and Lua, we checked for interpreter crashes and potential hangs (infinite loops). For Python—which, unlike Lua, has an exception mechanism—we also flagged whenever a test case led to unspecified exceptions being thrown. In general, one could find application-specific types of bugs by adding specifications in the form of assertions, as in normal unit tests.

6.5 Effectiveness for Bug Finding and Test Generation

In this section we present several case studies that illustrate how Cloud9 and Chef can explore and find new bugs.

6.5.1 Case Study #1: Curl

Curl is a popular data transfer tool for multiple network protocols, including HTTP and FTP. When testing it, Cloud9 found a new bug which causes Curl to crash when given a URL regular expression of the form "`http://site.{one,two,three}.com{`". Cloud9 exposed a general problem in Curl's handling of the case when braces used for regular expression globbing are not matched properly. The bug was confirmed and fixed within 24 hours by the developers.

This problem had not been noticed before because the globbing functionality in Curl was shadowed by the same functionality in command-line interpreters (e.g., Bash). This case study illustrates a situation that occurs often in practice: when a piece of software is used in a way that has not been tried before, it is likely to fail due to latent bugs.

6.5.2 Case Study #2: Memcached

Memcached is a distributed memory object cache system, mainly used to speed up web application access to persistent data, typically residing in a database.

Memcached comes with an extensive test suite comprised of C and Perl code. Running it completely on a machine takes about 1 minute; it runs 6,472 different test cases and explores 83.66% of the code. While this is considered thorough by today's standards, two easy Cloud9 test cases further increased code coverage. Table 6.4 contains a summary of our results, presented in more details in the following paragraphs.

Symbolic Packets The memcached server accepts commands over the network. Based on memcached's C test suite, we wrote a test case that sends memcached a generic, symbolic binary command (i.e., command content is fully symbolic), followed by a second symbolic command. This test captures all operations that entail a pair of commands.

A 24-worker Cloud9 explored in less than 1 hour all 74,503 paths associated with this sequence of two symbolic packets, covering an additional 1.13% of the code relative to the original test suite. What we found most encouraging in this result is that such exhaustive tests constitute first steps toward using symbolic tests to *prove* properties of real-world programs, not just to look for bugs. Symbolic tests may provide an alternative to complex proof mechanisms that is more intuitive for developers and thus more practical.

Symbolic Fault Injection We also tested memcached with fault injection enabled, whereby we injected all feasible failures in memcached's calls to the C Standard Library. After 10 minutes of testing, a 24-worker Cloud9 explored 312,465 paths, adding 1.28% over the base test suite. The fact that *line* coverage increased by so little, despite having covered almost 50× more paths, illustrates the weakness of line coverage as a metric for test quality—high line coverage should offer no high confidence in the tested code's quality.

For the fault injection experiment, we used a special strategy that sorts the execution states according to the number of faults recorded along their paths, and favors the states with fewer fault injection points. This led to a uniform injection of faults: we first injected one fault in every possible fault injection point along the original C test suite path, then injected pairs of faults, and so on. We believe this is a practical approach to using fault injection as part of regular testing.

Testing Method	Paths Covered	Isolated Coverage*	Cumulated Coverage**
Entire test suite	6,472	83.67%	—
Binary protocol test suite	27	46.79%	84.33% (+0.67%)
Symbolic packets	74,503	35.99%	84.79% (+1.13%)
Test suite + fault injection	312,465	47.82%	84.94% (+1.28%)

Table 6.4: Path and code coverage increase obtained by each symbolic testing technique on memcached. We show total coverage obtained with each testing method (*), as well as total coverage obtained by augmenting the original test suite with the indicated method (**); in parentheses, we show the increase over the entire test suite's coverage.

Hang Detection We tested memcached with symbolic UDP packets, and Cloud9 discovered a hang condition in the packet parsing code: when a sequence of packet fragments of a certain size arrive at the server, memcached enters an infinite loop, which prevents it from serving any further UDP connections. This bug can seriously hurt the availability of infrastructures using memcached.

We discovered the bug by limiting the maximum number of instructions executed per path to 5×10^6. The paths without the bug terminated after executing $\sim 3 \times 10^5$ instructions; the other paths that hit the maximum pointed us to the bug.

6.5.3 Case Study #3: Lighttpd

The lighttpd web server is specifically engineered for high request throughput, and it is quite sensitive to the rate at which new data is read from a socket. Alas, the POSIX specification offers no guarantee on the number of bytes that can be read from a file descriptor at a time. lighttpd 1.4.12 has a bug in the command-processing code that causes the server to crash (and connected clients to hang indefinitely) depending on how the incoming stream of requests is fragmented.

We wrote a symbolic test case to exercise different *stream fragmentation* patterns and see how different lighttpd versions behave. We constructed a simple HTTP request, which was then sent over the network to lighttpd. We activated network packet fragmentation via the symbolic `ioctl()` API explained in Section 3.2. We confirmed that certain fragmentation patterns cause lighttpd to crash (prior to the bug fix). However, we also tested the server right after the fix and discovered that the bug fix was incomplete, as some fragmentation patterns still cause a crash and hang the client (Table 6.5).

This case study shows that Cloud9 can find bugs caused by specific interactions with the environment which are hard to test with a concrete test suite. It also shows how Cloud9 can be used to write effective regression test suites—had a stream-fragmentation symbolic test been run after the fix, the lighttpd developers would have promptly discovered the incompleteness of their fix.

Fragmentation pattern (data sizes in bytes)	**ver. 1.4.12 (pre-patch)**	**ver. 1.4.13 (post-patch)**
1×28	OK	OK
$1 \times 26 + 1 \times 2$	crash + hang	OK
$2 + 5 + 1 + 5 + 2 \times 1 + 3 \times 2 + 5 + 2 \times 1$	crash + hang	crash + hang

Table 6.5: The behavior of different versions of lighttpd to three ways of fragmenting the HTTP request "GET /index.html HTTP/1.0CRLFCRLF" (string length 28).

6.5.4 Case Study #4: Bandicoot DBMS

Bandicoot is a lightweight DBMS that can be accessed over an HTTP interface. We exhaustively explored all paths handling the GET commands and found a bug in which Bandicoot reads from outside its allocated memory. The particular test we ran fortuitously did not result in a crash, as Bandicoot ended up reading from the libc memory allocator's metadata preceding the allocated block of memory. However, besides the read data being wrong, this bug could cause a crash depending on where the memory block was allocated.

To discover and diagnose this bug without Cloud9 is difficult. First, a concrete test case has little chance of triggering the bug. Second, searching for the bug with a sequential symbolic execution tool seems impractical: the exhaustive exploration took 9 hours with a 4-worker Cloud9 (and less than 1 hour with a 24-worker cluster).

6.5.5 Comparing Cloud9 to KLEE

Cloud9 inherits KLEE's capabilities, being able to recognize memory errors and failed assertions. We did not add much in terms of bug detection, only two mechanisms for detecting hangs: check if all symbolic threads are sleeping (deadlock) and set a threshold for the maximum number of instructions executed per path (infinite loop or livelock). Even so, Cloud9 can find bugs beyond KLEE's abilities because the POSIX model and symbolic tests allow Cloud9 to exercise additional interactions of the program with the operating system. The case studies showed that Cloud9 can explore conditions that are hard to produce reliably by running the concrete operating system, such as fragmentation patterns in network traffic and the occurrence of faults.

Our parallel Cloud9 prototype also has more total memory and CPU available, due to its distributed nature, so it can afford to explore more paths than KLEE. As we have shown above, it is feasible to offer proofs for certain program properties: despite the exponential nature of exhaustively exploring paths, one can build small but useful symbolic test cases that can be exhaustively executed.

6.5.6 Case Study #5: Exploratory Bug Finding in Python and Lua Packages

We now evaluate the effectiveness of the Chef-obtained symbolic execution engines for bug detection.

The specifications we used for our experiments are application-agnostic and only check for per-path termination within a given time bound and for the absence of unrecoverable crashes. The first specification checks whether a call into the runtime returns within 60 seconds. In this way, we discovered a bug in the Lua JSON package that causes the parser to hang in an infinite loop: if the JSON string contains the `/*` or `//` strings marking the start of a comment but no matching `*/` or line terminator, the parser reaches the end of the string and continues spinning waiting for another token. This bug is interesting for two reasons: First, comments are not part of the JSON standard, and the parser accepts them only for convenience, so this is a clear case of an interpreter-specific bug. Second, JSON encodings are normally automatically generated and transmitted over the network, so they are unlikely to contain comments; traditional testing is thus likely to miss this problem. However, an attacker could launch a denial of service attack by sending a JSON object with a malformed comment.

The second implicit specification checks that a program never terminates non-gracefully, i.e., the interpreter implementation or a native extension crashes without giving the program a chance to recover through the language exception mechanisms. In our experiments, our test cases did not expose any such behavior.

6.5.7 Case Study #6: Undocumented Exceptions in Python Packages

This scenario focuses on finding undocumented exceptions in Python code. Being memory-safe languages, crashes in Python and Lua code tend to be due to *unhandled exceptions* rather than bad explicit pointers. When such exceptions are not caught by the program, they propagate to the top of the stack and cause the program to be terminated prematurely. In dynamic languages, it is difficult to determine all the possible exceptions that a function can throw to the callee, because there is no language-enforced type-based API contract. Users of an API can only rely on the documentation or an inspection of the implementation. Therefore, undocumented exceptions are unlikely to be checked for in `try-except` constructs and can erroneously propagate further. They can then hurt productivity (e.g., a script that crashes just as it was about to complete a multi-TB backup job) or disrupt service (e.g., result in an HTTP 500 Internal Server Error).

We looked at all the Python exceptions triggered by the test cases generated using Chef and classified them into *documented* and *undocumented*. The documented exceptions are either exceptions explicitly mentioned in the package documentation or common Python exceptions that are part of the standard library (e.g., `KeyError`, `ValueError`, `TypeError`). Undocumented exceptions

are all the rest.

The sixth column in Table 6.3 summarizes our findings. We found four undocumented exceptions in **xlrd**, the largest package. These exceptions occur when parsing a Microsoft Excel file, and they are **BadZipfile**, **IndexError**, **error**, and **AssertionError**. These errors occur inside the inner components of the Excel parser, and should have either been documented or, preferably, been caught by the parser and re-raised as the user-facing **XLRDError**.

6.6 Efficiency of Test Generation with Chef

6.6.1 Impact of CUPA Heuristics and Interpreter Optimizations

We now analyze the impact of the CUPA heuristics (described in Section 4.5.1) and the interpreter optimizations (described in Section 4.6.2) on test generation effectiveness. Specifically, we measure the number of paths (respectively source code lines) covered by the test suite generated in 30 minutes for the packages in Table 6.3.

We compare the results obtained in 4 different configurations: (1) the baseline, consisting of performing random state selection while executing the unmodified interpreter, and then either use (2) the path- or coverage-optimized CUPA only, (3) the optimized interpreter only, or (4) both CUPA and the optimized interpreter. This way we measure the individual contribution of each technique, as well as their aggregate behavior.

Test Case Generation Figure 6.4 compares the number of test cases generated with each of the four Chef configurations, using the path-optimized CUPA (Section 4.5.2). We only count the *relevant* high-level test cases, that is, each test case exercises a unique high-level path in the target Python program.

For all but one of the 11 packages (6 Python plus 5 Lua), the aggregate CUPA + interpreter optimizations performs the best, often by a significant margin over the baseline. This validates the design premises behind our techniques.

The CUPA strategy and the interpreter optimizations may interact non-linearly. In two cases (Python's **xlrd** and **simplejson**), the aggregate significantly outperforms either individual technique. These are cases where the result is better than the sum of its parts. In the other cases, the result is roughly the sum of each part, although the contribution of each part differs among targets. This is visually depicted on the log-scale graph: for each cluster, the heights of the middle bars measured from level $1\times$ roughly add up to the height of the aggregate (left) bar.

In one case (Lua's **JSON**), the aggregate performs worse on average than using the interpreter optimizations alone. Moreover, the performance of each configuration is less predictable, as shown

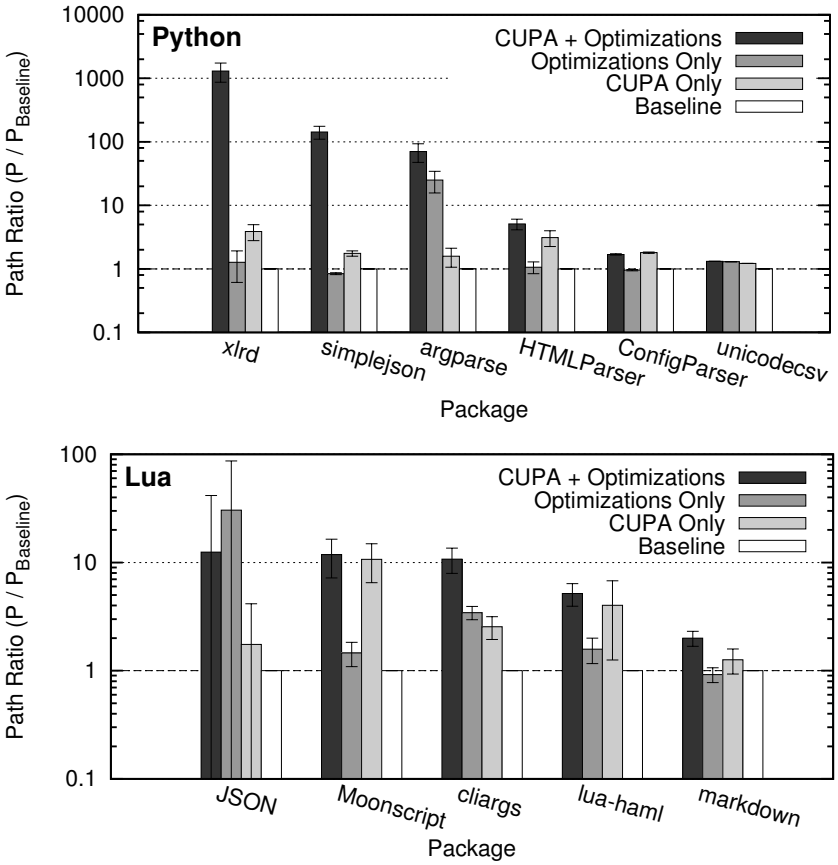

Figure 6.4: The number of Python and Lua test cases generated by coverage- and path-optimized CUPA relative to random state selection (logarithmic scale).

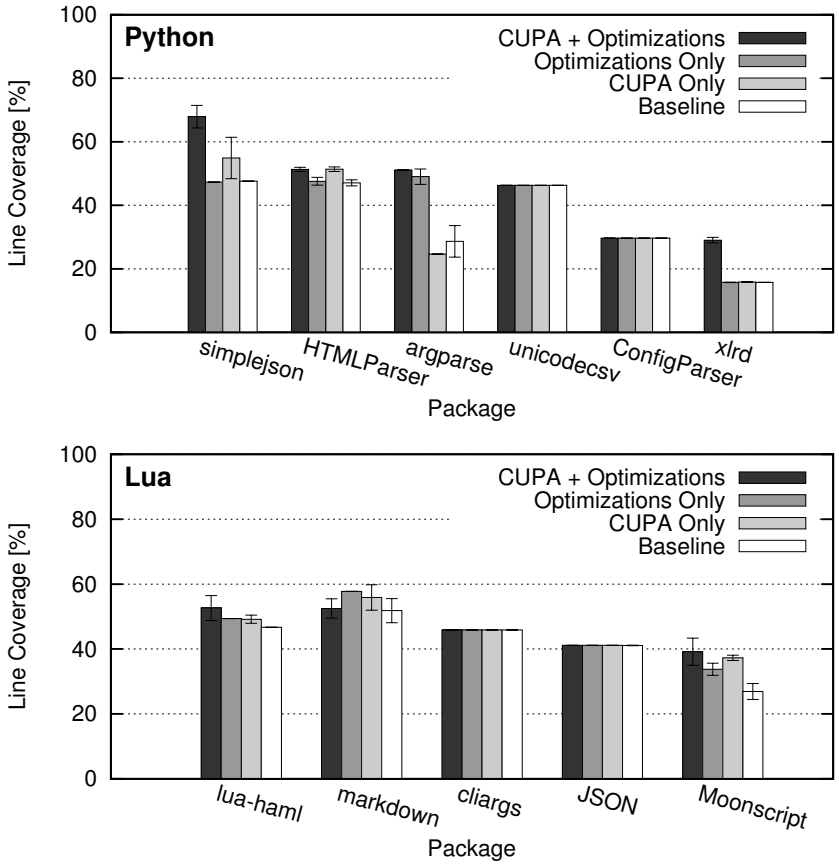

Figure 6.5: Line coverage for the experiments of Figure 6.4.

by the large error bars. This behavior is due to the generated tests that cause the interpreter to hang, as explained in Section 6.5. To detect hangs, the test runs for 60 seconds before switching to another test case. This acts as a "penalty" for the configurations that find more paths leading to the hang and also skews the distribution of path execution times, since the hanging paths take significantly longer than the normal (terminating) paths.

Line Coverage Figure 6.5 shows the line coverage achieved by each configuration, using CUPA optimized for line coverage (Section 4.5.3). In 6 out of 11 packages, the coverage improvement is noticeable, and for Python's **`simplejson`** and **`xlrd`**, the improvements are significant (80% and 40%).

Note that these coverage improvements are obtained using basic symbolic tests that do not make assumptions about the input format. We believe that tailoring the symbolic tests to the specifics of each package could improve these results significantly.

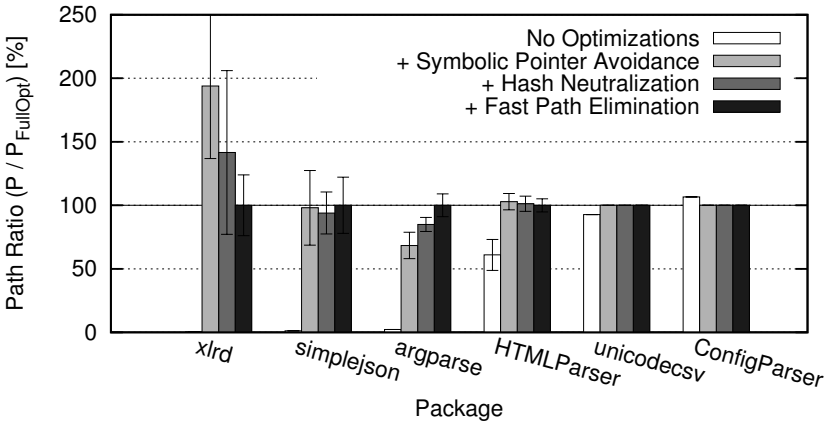

Figure 6.6: The contribution of interpreter optimizations for Python as number of high-level paths explored. Number of paths is relative to full optimizations (100%) for each package.

6.6.2 Breaking Down Chef's Interpreter Optimizations

We now analyze in more depth the impact of the interpreter optimizations by breaking them down into the three types mentioned in Section 4.6.2: avoiding symbolic pointers, hash neutralization, and fast-path elimination. We run again the symbolic tests for 30 minutes, using the path-optimized CUPA and four different interpreter builds, starting from the vanilla interpreter and adding the optimization types one by one. For each build and package, we count the number of high-level paths discovered by Chef.

Figure 6.6 shows the results for Python. The data is normalized such that the number of high-level paths for each target reaches 100%. For 3 out of 6 packages (**`simplejson`**, **`argparse`**, and **`HTMLParser`**), Chef's performance monotonically increases as more optimizations are introduced. For **`unicodecsv`** and **`ConfigParser`**, the optimizations do not bring any benefits or even hurt slightly.

However, in the case of **`xlrd`**, hash neutralization and fast path elimination seem to actually *hurt* symbolic execution, since the best performance is attained when only symbolic pointer avoidance is in effect. We explain this behavior by the fact that the different optimization levels cause the search strategy to explore different behaviors of the target package. **`xlrd`** is by far the largest Python package in our evaluation (7.2KLOC vs. the second largest of 1.4KLOC) and includes a diverse set of behaviors, each with its own performance properties.

This result suggests that, for large packages, a *portfolio* of interpreter builds with different optimizations enabled would help further increase the path coverage.

6.6.3 Comparing Chef Against Hand-Made Engines

We now evaluate the trade-offs in using a symbolic execution engine generated with Chef over building one "by hand".

Hand-Made Engines To our knowledge, no symbolic execution engine for Lua exists. For Python, we found three research tools, which we compare Chef to. (1) CutiePy [81] is a concolic engine based on a formal model of the Python language. It uses a custom CPython interpreter to drive a concrete execution, along with updating the symbolic state according to model semantics. (2) NICE-PySE [29] is part of the NICE framework for testing OpenFlow applications. We will refer to it as NICE, for brevity. It wraps supported data types into symbolic counterparts that carry the symbolic store, and uses Python's tracing mechanisms to implement the interpretation loop fully in Python. (3) The symbolic execution engine of the scalability testing tool Commuter [36] is also entirely built in Python. Its primary purpose is the construction of models that explicitly use an API of symbolic data types.

We perform our comparison along three aspects: language features supported, implementation faithfulness, and performance. The last two aspects are evaluated only against NICE, which, besides being open source, is most compatible with our symbolic data representation (based on STP [43]).

Language Feature Support Table 6.6 summarizes the language feature support for Chef, NICE, CutiePy, and Commuter[1]. We relied on information from the respective papers in all cases and additionally on the implementation in the cases of NICE and Commuter, which are available as open source.

We distinguish engines designed to support arbitrary Python code (the "Vanilla" label) and those where the symbolic data types are an API used by model code (the "Model" label). Engines in the "Model" category essentially offer a "symbolic domain-specific language" on top of the interpreted language. Chef, CutiePy, and NICE are "vanilla" engines, since their testing targets do not have to be aware that they are being symbolically executed. Commuter is a model-based engine, since its testing targets are bound to the symbolic API offered by the engine.

We grouped the supported language features into program state representation (the language data model and types) and manipulation (the operations on data). We divide data types into values (integers, strings and floating-point), collections (lists and dictionaries), and user-defined classes. The operations consist of data manipulation, basic control flow (e.g., branches, method calls), advanced control flow (e.g., exception handling, generators), and native method invocations (they

[1] We consider the features available at the moment of performing this evaluation (early 2014).

	Chef	CutiePy	NICE	Commuter
Engine type	Vanilla	Vanilla	Vanilla	Model
Data types				
Integers	●	●	●	●
Strings	●	○	○	◐
Floating point	○	○	○	○
Lists and maps	●*	◐	○	●
User-defined classes	●*	○	◐	◐
Operations				
Data manipulation	●	◐	◐	◐
Basic control flow	●	●	◐	●
Advanced control flow	●	●	○	●
Native methods	●	◐	○	○

●Complete ◐Partial ○Not supported

Table 6.6: Language feature support comparison for Chef and dedicated Python symbolic execution engines. Complete support with (*) refers to the internal program data flow and not to the initial symbolic variables.

are atomic operations at the high level). We also include in the comparison the ability to execute unsupported operations in concrete-only mode.

In a nutshell, CutiePy is able to complete correctly any execution in concrete mode by using the interpreter implementation directly. However, the symbolic semantics for each data type and native function must be explicitly provided by the developer, which makes CutiePy impractical to use with rich Python applications. NICE suffers from the additional limitation that it has to support each bytecode instruction explicitly, which makes the tool impossible to use beyond its target applications. Finally, Commuter provides a rich set of symbolic data types, including lists and maps, by taking advantage of Z3's extended support for arrays [38]. However, it supports only Python programs explicitly written against its API and does not handle native functions.

The engine generated by Chef offers complete symbolic support for almost all language features. Floating point operations are supported only concretely, due to lack of support in STP, the constraint solver used by S2E. For the same reasons, the symbolic program inputs can only be integers and strings. However, all data structures are supported during the execution.

Each half or empty bullet in Table 6.6 implies that significant engineering effort would be required to complete a feature. While useful for their evaluation targets, NICE and CutiePy are unable to handle a complex software package that makes use of Python's many language features.

Use as Reference Implementation When the need for performance justifies investing in a dedicated engine implementation, an engine created from Chef can serve as a reference implementation during development. One can find bugs in a symbolic execution engine by comparing its test cases

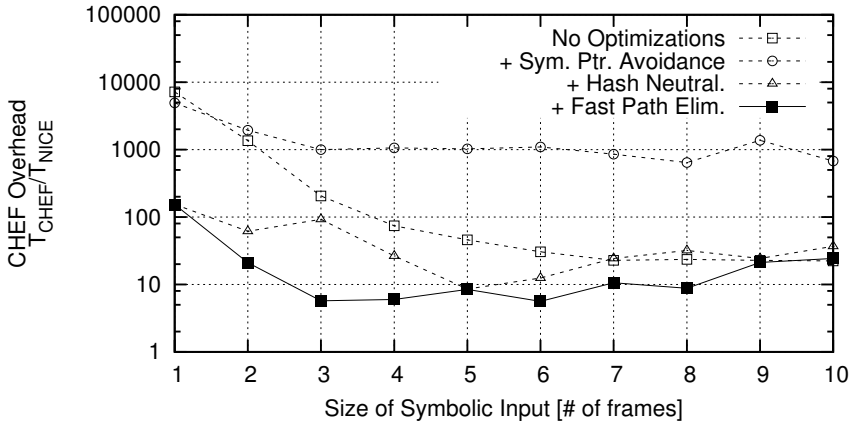

Figure 6.7: Average overhead of Chef compared to NICE, computed as ratio of average per-path execution times. The average divides total tool execution time by number of high-level paths generated.

with those generated by Chef. The process can be automated by tracking the test cases generated by the target engine along the high level paths generated by Chef to determine duplicates and missed feasible paths.

In this mode, we found a bug in the NICE implementation, which was causing it to generate redundant test cases and miss feasible paths. The bug was in the way NICE handled `if not <expr>` statements in Python, causing the engine to select for exploration the wrong branch alternate and end up along an old path. We are assisting the NICE developers in identifying and fixing any other such bugs.

In conclusion, the experiment provides evidence that a system combining an established low-level symbolic execution engine (e.g., S2E) with a reference interpreter implementation is more robust than a symbolic execution engine built from scratch.

Performance The downside of Chef is that the symbolic execution engines produced are slower than their hand-written equivalents. We quantify this drawback by applying Chef to the experimental setup of NICE, consisting of an OpenFlow switch controller program that implements a MAC learning algorithm. The controller receives as input a sequence of Ethernet frames and, in response, updates its forwarding table (stored as a Python dictionary). We use symbolic tests that supply sequences of between 1 and 10 Ethernet frames, each having the MAC address and frame types marked as symbolic.

Given the small size of the controller (less than 100 LOC), the number of execution paths is relatively small, and choosing low-level paths at random quickly discovers new high-level paths. Therefore, the search strategy has no impact (in the experiments we used path-optimized CUPA). However, the interpreter optimizations are crucial, since the controller code relies heavily on the

dictionary. As in Section 6.6.2, we use several interpreter builds with optimizations introduced one-by-one.

Figure 6.7 illustrates the overhead for each optimization configuration, as a function of number of Ethernet frames supplied. The overhead is computed as the ratio between the average execution times per high-level path of NICE and Chef. In turn, the execution time per high-level path is computed by dividing the entire execution time of each tool by the number of paths it produced.

The performance of each optimization configuration illustrates the sources of path explosion and slowdown in the vanilla interpreter. With no optimizations, symbolic keys in the MAC dictionary cause massive path explosion due to symbolic pointers. When avoiding symbolic pointers, performance drops even more due to symbolic hash computations. This penalty is reduced up to two orders of magnitude with hash neutralization. Finally, fast path elimination reduces the forking inside string key comparisons in the dictionary.

The shape of the final performance curve (the solid line) is convex. For 1 and 2 symbolic frames, the search space is quickly exhausted and the execution time is dominated by Chef's initialization costs, i.e., setting up the symbolic VM and executing the interpreter initialization inside the guest. This results in an execution overhead as high as 120×. For more symbolic frames, the initialization cost is amortized, and the overhead goes below 5×. However, as the number of frames increases, so does the length of the execution paths and the size of the path constraints, which deepens the gap between Chef's low-level reasoning and NICE's higher level abstractions. For 10 symbolic frames, the overhead is around 40×.

Despite Chef's performance penalty, the alternative of writing an engine by hand is daunting. It involves developing explicit models that, for a language like Python, are expensive, error-prone, and require continuous adjustments as the language evolves. Where performance is crucial, a hand-written engine is superior; however, we believe that Chef is a good match in many cases.

6.7 Scalability of Parallel Symbolic Execution in Cloud9

We evaluate Cloud9 using two metrics:

1. The time to reach a certain goal (e.g., an exhaustive path exploration, or a fixed coverage level)—we consider this an *external* metric, which measures the performance of the testing platform in terms of its end results.

2. The useful work performed during exploration, measured as the number of useful (non-replay) instructions executed symbolically. This is an *internal* metric that measures the efficiency of Cloud9's internal operation.

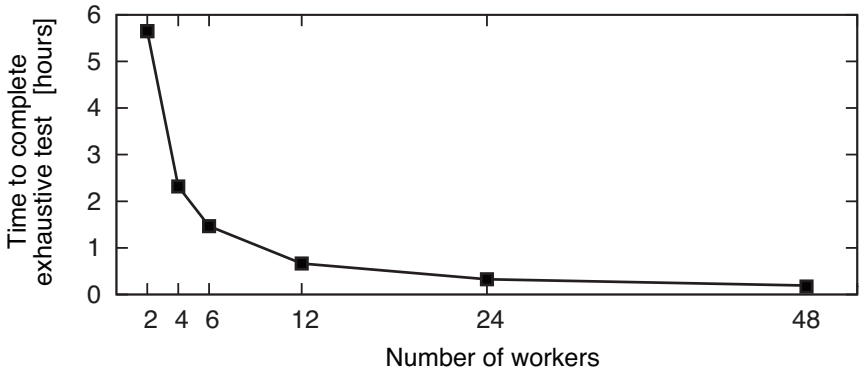

Figure 6.8: Cloud9 scalability in terms of the time it takes to exhaustively complete a symbolic test case for memcached.

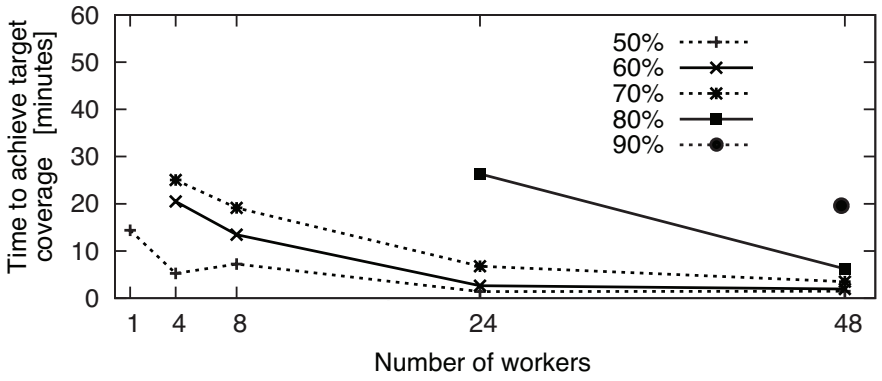

Figure 6.9: Cloud9 scalability in terms of the time it takes to obtain a target coverage level when testing `printf`.

A cluster-based symbolic execution engine *scales* with the number of workers if these two metrics improve proportionally with the number of workers in the cluster.

Time Scalability We show that Cloud9 scales linearly by achieving the same testing goal proportionally faster as the number of workers increases. We consider two scenarios.

First, we measure how fast Cloud9 can exhaustively explore a fixed number of paths in the symbolic execution tree. For this, we use a symbolic test case that generates all the possible paths involved in receiving and processing two symbolic messages in the memcached server. Figure 6.8 shows the time required to finish the test case with a variable number of workers: every doubling in the number of workers roughly halves the time to completion. With 48 workers, the time to complete is about 10 minutes; for 1 worker, exploration time exceeds our 10-hour limit on the experiment.

Second, we measure the time it takes Cloud9 to reach a fixed coverage level for the `printf` UNIX utility. `printf` performs a lot of parsing of its input (format specifiers), which produces

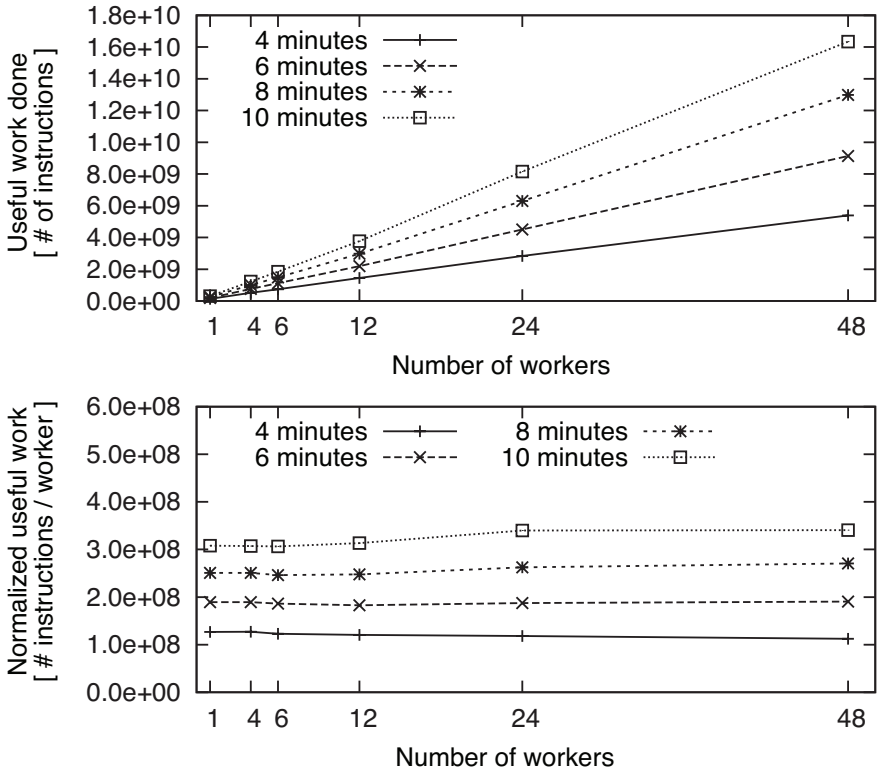

Figure 6.10: Cloud9 scalability in terms of useful work done for four different running times when testing memcached.

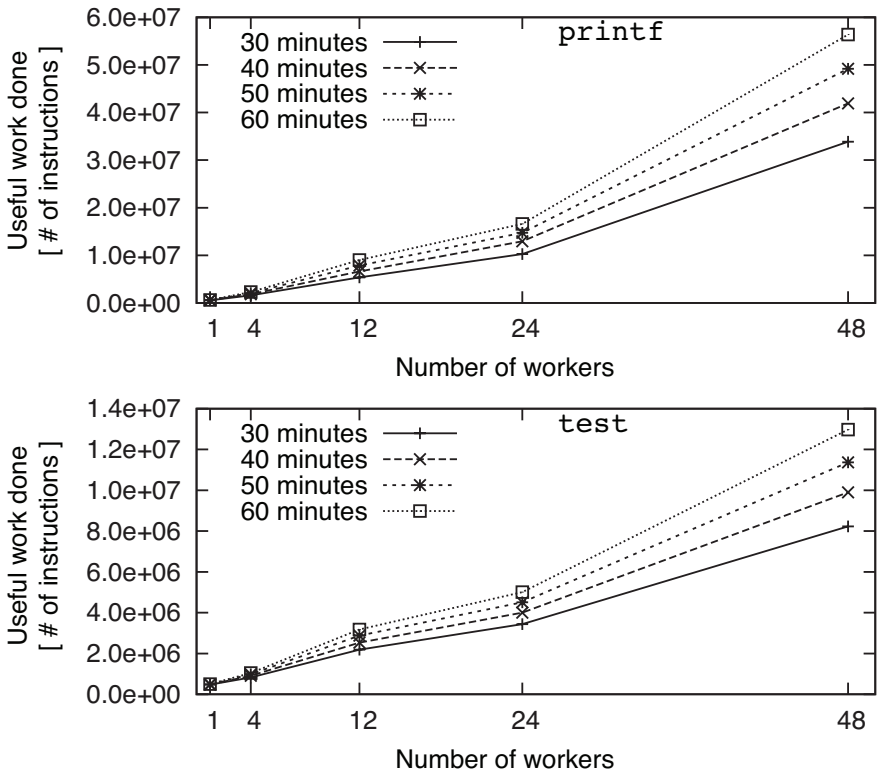

Figure 6.11: Cloud9's useful work on `printf` (top) and `test` (bottom) increases roughly linearly in the size of the cluster.

complex constraints when executed symbolically. Figure 6.9 shows that the time to achieve a coverage target decreases proportionally with the number of added workers. The low 50% coverage level can be easily achieved even with a sequential SEE (1-worker Cloud9). However, higher coverage levels require more workers, if they are to be achieved in a reasonable amount of time; e.g., only a 48-worker Cloud9 is able to achieve 90% coverage. The anomaly at 4 workers for 50% coverage is due to high variance; when the number of workers is low, the average (5 ± 4.7 minutes over 10 experiments) can be erratic due to the random choices in the random-path search strategy.

Work Scalability We now consider the same scalability experiments from the perspective of useful work done by Cloud9: we measure both the total number of instructions (from the target program) executed during the exploration process, as well as normalize this value per worker. This measurement indicates whether the overheads associated with parallel symbolic execution impact the efficiency of exploration, or are negligible. Figure 6.10 shows the results for memcached, confirming that Cloud9 scales linearly in terms of useful work done (top graph). The average useful work done by a worker (bottom graph) is relatively independent of the total number of workers in the cluster, so adding more workers improves proportionally Cloud9's results.

In Figure 6.11 we show the results for `printf` and `test`, UNIX utilities that are an order of magnitude smaller than memcached. We find that the useful work done scales in a similar way to memcached, even though the three programs are quite different from each other (e.g., `printf` does mostly parsing and formatting, while memcached does mostly data structure manipulations and network I/O).

In conclusion, Cloud9 scales linearly with the number of workers, both in terms of the time to complete a symbolic testing task and in terms of reaching a target coverage level.

6.8 Summary

In this chapter, we presented the Cloud9 and Chef symbolic execution platforms for systems interacting with the POSIX operating system and programs written in interpreted languages, respectively. We demonstrated that they can find bugs and generate test suites in wide range of real-world software, from UNIX utilities to web servers and popular Python and Lua packages. Cloud9's POSIX model based on the split model approach was crucial in uncovering bugs in the program interaction with the operating systems, such as bugs resulting from mishandling the fragmentation of a TCP stream. Chef's use of the interpreter as as executable specification provided complete and correct semantics of the language, which enabled it to target popular Python and Lua packages.

Finally, we showed promising results in our future plan of building a cloud-based distributed symbolic execution platform, by demonstrating linear scalability of our symbolic execution parallelization algorithm on a workload of UNIX utilities.

Chapter 7

Conclusion

Symbolic execution is an automated test generation technique that stands out for its soundness, completeness, and flexibility. Alas, it faces significant hurdles on its way to adoption for the large systems encountered in the real world, because of path explosion and the excessive complexity of symbolic formulae.

The system modularity and the natural boundaries between components permit symbolic execution to handle large systems by executing the different parts of the system separately. However, a symbolic execution engine has to find the right balance between efficiency, accuracy, and completeness to effectively provide the environment interface of a component—a conundrum known as the environment problem.

This thesis examines the trade-offs employed by existing symbolic execution engines and proposes new techniques to improve these trade-offs on one axis of the program environment design space: the stability of its interface.

Stable Operating System Interfaces On the one end of the axis, this thesis addresses the environment problem for software interacting with the operating system, which has a stable and well-documented interface.

This thesis shows that it is economical to provide an accurate and practically complete operating system model, by splitting it into a core set of primitives built into the symbolic execution engine and a guest-level model that provides the full operating system interface (Chapter 3). As few as two built-in primitives are sufficient to support complex operating system interfaces: threads with synchronization and address spaces with shared memory. Moreover, the operating system model engenders a more effective way for developers to test their software with the symbolic execution engine, by writing symbolic tests. Symbolic tests control the operating system model to explore conditions that are hard to produce reliably in a concrete test case.

We prototyped the split model approach and symbolic tests in the Cloud9 symbolic execution

platform, which exposes hard-to-reproduce bugs in systems such as UNIX utilities, web servers, and distributed systems. Cloud9 is available at http://cloud9.epfl.ch.

Fast-Changing Interpreted Languages On the other end of the axis, this thesis addresses the problem of building complete and correct symbolic execution engines for interpreted languages, such as Python, Ruby, or JavaScript. The environment of an interpreted program consists of the language semantics and the library of functions built into the interpreter. Building a symbolic execution engine by hand for a modern interpreted language is a significant engineering effort, due to their rich semantics, rapid evolution, and lack of precise specifications.

This thesis introduces the idea of using the language interpreter itself as an "executable specification" (Chapter 4). The idea is to run the interpreter executable inside a binary-level symbolic execution engine, while running the target program. The resulting system acts as a high-level symbolic execution engine for the program. To circumvent the path explosion arising in the interpreter, Class-Uniform Path Analysis (CUPA) groups the execution paths in the symbolic execution tree into equivalence classes that isolate the sources of path explosion in the interpreter. We prototyped these ideas in the form of Chef, a symbolic execution platform for interpreted languages that generates up to 1000 times more tests in popular Python and Lua packages compared to a plain execution of the interpreters. Chef is available at http://dslab.epfl.ch/proj/chef/.

Towards an Automated Software Testing Service Together, the two techniques introduced in this thesis enable effective symbolic execution for a wide range of software, from low-level system utilities, to web servers and interpreted programs. This software forms the modern web application stack, which consists of a high-level application logic written against a platform-as-a-service (PaaS) API, transparently scaled in the cloud across machines running the system software.

The work presented in this thesis enables the vision of an automated software testing service based on symbolic execution that targets applications running "in-vivo" in their production environments (Chapter 5). The testing service focuses on each application layer at a time, from the low-level web server to the high-level application logic, avoiding the combinatorial explosion of paths across layers. The service scales to large clusters of commodity hardware owing to a symbolic execution parallelization algorithm that is the first to demonstrate linear scalability.

Looking Forward This thesis teaches two lessons on making symbolic execution applicable to large software. On the one hand, a hand-written model outperforms a real implementation in a symbolic execution engine. A model captures only the essential functionality, leaving out complicating details, such as performance and extensibility. On the other hand, the implementation is what gets executed in practice, so a model for complex software with evolving semantics is likely

to be inaccurate and difficult to maintain.

Reconciling the two conflicting perspectives requires researching new symbolic execution techniques that run on real implementations, while providing the performance of domain-specific engines. We identify three major directions to pursue in order to achieve this research vision:

1. **Eliminate formula redundancy.** A symbolic execution engine spends most of its resources performing constraint solving [25]. Essentially, its role is to convert the target program and its properties to a representation that can be directly handled by a solver. Unfortunately, a straightforward symbolic execution algorithm that separately executes each program path will unnecessarily explore the same program statements multiple times and repeat the formulas sent to the solver. To minimize this redundancy, techniques such as state merging [61] and compositionality [44] bundle execution states as disjunctive formulas. More work is needed, though, to avoid creating formulas that are too complex for the constraint solver and that make the merging of states disadvantageous.

2. **Automatically substitute implementations with lazy abstractions.** Eliminating formula redundancy is not enough to scale symbolic execution to large systems. Even in the optimal case, the size of the formulae grows linearly in the program size and may take exponentially more time to solve. This places an upper bound on the size of the code that can be exhaustively executed with any given budget of CPU and memory. Hence, scaling the analysis beyond this limit can only be done by reducing the size of symbolic formulae through *abstraction*. The idea is to replace an operation in the system with an uninterpreted function that is handled by a specialized decision procedure. Operations on common data structures, such as strings, lists, and maps, are the most obvious candidates, but other common system primitives, such as memory allocators, can also be abstracted. The main challenges are (a) to automatically detect abstraction opportunities in systems and (b) to further expand the set of abstractions and decision procedures available in the solver.

3. **Automated abstraction discovery.** Today, an experienced developer can study a large code base and build an intuition on the role of each system component. To ultimately reach their full potential, program analysis tools should be able to perform this task on their own and orders of magnitude faster than a human. Achieving this goal is a cross-disciplinary effort, which spreads from traditional formal methods into machine learning techniques and artificial intelligence.

Bibliography

[1] Rachel Abrams. Target puts data breach costs at $148 million, and forecasts profit drop. *The New York Times*, August 2014.

[2] Al Danial. Cloc. `http://cloc.sourceforge.net/`.

[3] Pedram Amini and Aaron Portnoy. Sulley fuzzing framework. http://www.fuzzing.org/wp-content/SulleyManual.pdf, 2010.

[4] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[5] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF–SE: A symbolic execution extension to Java PathFinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[6] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Adam Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

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

[8] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[9] Radu Banabic. *Techniques for identifying elusive corner-case bugs in systems software*. PhD thesis, EPFL, 2015.

[10] Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable multi-core LTL model-checking. In *International SPIN Workshop (SPIN)*, 2007.

[11] Brooks Barnes and Michael Cieply. Intrusion on Sony unit prompts a shutdown of messaging systems. *The New York Times*, November 2014.

[12] Clark Barrett and Cesare Tinelli. CVC3. In *International Conference on Computer Aided Verification (CAV)*, 2007.

[13] Kent Beck. Simple Smalltalk testing: With patterns. http://www.xprogramming.com/testfram.htm.

[14] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[15] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[16] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[17] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Technical Report MSR-TR-2012-55, Microsoft Research, 2012.

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[19] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – A formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software (ICRS)*, 1975.

[20] Manfred Broy. Challenges in automotive software engineering. In *International Conference on Software Engineering (ICSE)*, 2006.

[21] Stefan Bucur, Johannes Kinder, and George Candea. Making automated testing of cloud applications an integral component of PaaS. In *Proc. 4th Asia-Pacific Workshop on Systems (APSYS 2013)*, 2013.

[22] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

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

[24] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[25] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[26] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[27] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *International Conference on Software Engineering (ICSE)*, 2011.

[28] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 2013.

[29] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[30] Capgemini. *World Quality Report*. 2014.

[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[32] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HOTDEP)*, 2009.

[33] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] Test results for the Chromium browser. https://test-results.appspot.com/testfile?testtype=layout-tests (retrieved on 05/27/2015).

[35] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming (ICFP)*, 2000.

[36] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[37] Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *ACM EuroSys European Conference on Computer Systems (EUROSYS)*, 2011.

[38] Leonardo de Moura and Nikolaj Bjorner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2009.

[39] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[40] Edsger Wybe Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[41] The Django project. https://www.djangoproject.com/.

[42] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2003.

[43] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV)*, 2007.

[44] Patrice Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[46] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[47] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *ACM Queue*, 2012.

[48] Google App Engine. https://developers.google.com/appengine/.

[49] The Heroku cloud application platform. https://www.heroku.com/.

[50] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In *International SPIN Workshop (SPIN)*, 2008.

[51] IBM. 2015 cost of data breach study. http://www-03.ibm.com/security/data-breach/, 2015.

[52] The ISO/IEC 9899:1990 C language standard. http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=17782, 1990.

[53] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[54] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with Portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[55] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[56] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[57] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering (ICSE)*, 2009.

[58] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

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

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

[61] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[62] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[63] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[64] Guodong Li, Indradeep Ghosh, and S. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *International Conference on Computer Aided Verification (CAV)*, 2011.

[65] LinkedIn passwords leaked by hackers. http://www.bbc.co.uk/news/technology-18338956, June 2012.

[66] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, 2007.

[67] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[68] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[69] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[70] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, 2001.

[71] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[72] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CCC)*, 2002.

[73] Jared Newman. Gmail bug deletes e-mails for 150,000 users. *PCWorld*, February 2011.

[74] National vulnerability database. https://nvd.nist.gov/.

[75] Stephen O'Grady. The RedMonk programming language rankings: January 2015. http://redmonk.com/sograd y/2015/01/14/language-rankings-1-15/, 2015.

[76] Soila Peret and Pryia Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[77] Corina S. Păsăreanu, Peter C. Mehlitz, David H. 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 *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[78] Python Software Foundation. *The Python Language Reference*. http://docs.python.org/3/reference/.

[79] RedHat security. http://www.redhat.com/security/updates/classification, 2005.

[80] Shane Richmond. Millions of internet users hit by massive Sony PlayStation data theft. *The Telegraph*, April 2011.

[81] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M. Clarke. Finding errors in Python programs using dynamic symbolic execution. In *International Conference on Testing Software and Systems (ICTSS)*, 2013.

[82] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[83] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[84] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2013.

[85] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[86] 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 *International Conference on Information Systems Security (ICISS)*, 2008.

[87] Nikolai Tillmann and Jonathan De Halleux. Pex – White box test generation for .NET. *Tests and Proofs (TAP)*, 2008.

[88] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[89] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[90] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 2003.

[91] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[92] Martin Vuagnoux. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany*, 2005.

[93] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -OVERIFY: Optimizing programs for fast verification. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2013.

[94] The webapp2 Python web framework. http://webapp-improved.appspot.com/.

[95] The WebTest Python testing framework. http://webtest.pythonpaste.org/en/latest/.

[96] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[97] Michal Zalewski, Niels Heinen, and Sebastian Roschke. Skipfish–Web application security scanner. http://code.google.com/p/skipfish/, 2011.

[98] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conference on Computer Systems (EUROSYS)*, 2010.

[99] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2013.

Stefan Bucur

École Polytechnique Fédérale de Lausanne (EPFL)
EPFL-IC-DSLAB, INN328, Station 14
1015 Lausanne, Switzerland

+41 788 84 36 33
stefan.bucur@epfl.ch
http://dslab.epfl.ch/people/bucur

Research Interests

Program analysis, operating systems, programming languages, compilers

I am interested in scaling precise program analysis to real-world software systems. My graduate work focused on enabling effective symbolic execution on systems that are large, span layers of abstraction, and interact with their environments.

Education

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland Sept. 2009 - Present
Ph.D. Candidate in Computer Science
Dependable Systems Laboratory, under the direction of Prof. George Candea.

"Politehnica" University of Bucharest, Romania 2004 - 2009
Dipl. Eng. in Computer Science
GPA: 10/10 (Valedictorian)
Thesis: *Automatic Code Formatting for Structured Languages*

Projects

Chef is a platform for obtaining symbolic execution engines for interpreted languages, such as Python, Ruby, or JavaScript. Chef reuses the interpreter as an executable language specification, thus eliminating the need for writing the engine from scratch. — http://dslab.epfl.ch/proj/chef/

Cloud9 is a parallel symbolic execution engine that scales on shared-nothing clusters of commodity hardware. Cloud9 features a symbolic POSIX environment model that enables it to target systems ranging from command line utilities to web servers and distributed systems. — http://cloud9.epfl.ch/

Work Experience

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland Sept. 2009 - present
Research Assistant *Lausanne, Switzerland*

I am working in the Dependable Systems Laboratory, under the direction of Prof. George Candea. Starting from 2012, my research was supported by a 3-year Google European Doctoral Fellowship in Software Dependability.

Google Inc. Nov. 2011 - Feb. 2012
Software Engineering Intern *Zurich, Switzerland*

I worked on deploying the Cloud9 parallel symbolic execution engine within Google and scaling it to test parts of the Chromium open source browser.

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland July - Sept. 2009
Student Intern *Lausanne, Switzerland*

I worked on the initial prototype of the Cloud9 parallel symbolic execution engine.

Adobe Systems March - June 2009
Student Intern *Bucharest, Romania*

I worked on an reusable and extensible automated source code formatting system, capable of applying the same set of formatting rules to multiple programming languages sharing a similar structure. We prototyped the algorithm as an Eclipse / Adobe Flex Builder plug-in.

Google Inc. May - August 2008
Contract Work as a Google Summer of Code 2008 Student

I added support for loading and linking dynamic ELF modules for the Syslinux boot-loader suite. My mentor was H. Peter Anvin, a Linux kernel maintainer and the main author of the Syslinux project.

Refereed Publications

[ASPLOS] **Prototyping Symbolic Execution Engines for Interpreted Languages.** Stefan Bucur, Johannes Kinder, and George Candea. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, March 2014

[APSYS] **Making Automated Testing of Cloud Applications an Integral Component of PaaS.** Stefan Bucur, Johannes Kinder, and George Candea. *Asia-Pacific Workshop on Systems (APSYS)*, Singapore, July 2013

[PLDI] **Efficient State Merging in Symbolic Execution.** Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. *Conf. on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012

[EuroSys] **Parallel Symbolic Execution for Automated Real-World Software Testing.** Stefan Bucur, Vlad Ureche, Cristian Zamfir, George Candea. *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys),* Salzburg, Austria 2011.

[SOCC] **Automated Software Testing as a Service.** George Candea, Stefan Bucur, Cristian Zamfir. *ACM Symposium on Cloud Computing (SOCC),* Indianapolis, IN, June 2010.

[ACM OSR] **Cloud9: A Software Testing Service.** Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, George Candea. *ACM Operating Systems Review, Vol. 43, No. 4*, December 2009. (Also in *Proceedings of the 3rd SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, October 2009)

Honors

Gold Prize in the Open Source Software World Challenge 2013
for the Cloud9 project

Google European Doctoral Fellowship in Software Dependability 2011

Valedictorian of the 2009 Class 2009
in “Politehnica” University of Bucharest

First Prize at the National IBM Romania Best Linux Application 2008
for the project *Dynamic Loading of ELF Modules for SYSLINUX*

Worldwide Finals of Windows Embedded Student Challenge 2006
for the project *PiCoPS: Pipe Contamination Prevention System*

Bronze Medal at the International Physics Olympiad, Pohang, South Korea 2004

Invited Talks and Conference Presentations

Prototyping Symbolic Execution Engines for Interpreted Languages 2014
Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Salt Lake City, UT

Parallel Symbolic Execution for Automated Real-world Software Testing 2012
2nd EcoCloud Annual Event, Lausanne, Switzerland

Parallel and Selective Symbolic Execution 2011
1st International SAT/SMT Solver Summer School, MIT, Cambridge, MA

Parallel Symbolic Execution for Automated Real-World Software Testing 2011
6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), Salzburg, Austria

Cloud9: A Software Testing Service 2009
3rd SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS), Big Sky, MT.

Teaching

EPFL

Teaching Assistant

Software Engineering	Fall 2010, 2012, 2013
Software Development Project	Fall 2010
Probabilities and Statistics	Spring 2012
Programming I	Spring 2014

Project Supervisor

"Guest-level API for the S2E Symbolic Analysis Platform"	Fall 2014
Martin Weber - Bachelor semester project	
"Extending Cloud9's POSIX model with signals and symbolic file systems"	Spring 2011
Calin Iorgulescu - BSc disertation	
"Extending Cloud9's POSIX model with IPC and advanced pthreads support"	Spring 2011
Tudor Cazangiu - BSc disertation	

"Politehnica" University of Bucharest

Teaching Assistant

Computer Programming	Fall 2006, 2008
Data Structures	Spring 2007, 2008
Introduction to Operating Systems	Fall 2006, 2007
Compilers Design	Fall 2008

Google Summer of Code 2009

Mentored Claudiu Mihail in the project
"Converting the CLI subsystem of the Syslinux bootloader from assembly to C"

Professional Service

Member of Program Committees

NBiS (Network-Based Information Systems)	2011

External Reviewer

SAS (International Static Analysis Symposium)	2014
SPE (Software: Practice and Experience)	2014
SOSP (ACM Symposium on Operating Systems Principles)	2011, 2013
HotOS (USENIX Workshop on Hot Topics in Operating Systems)	2013
CIDR (Conference on Innovative Data Systems Research)	2013
SOCC (ACM Symposium on Cloud Computing)	2012
ASPLOS (ACM Conf. on Architectural Support for Prog. Lang. and Operating Systems)	2010, 2011
EuroSys (ACM SIGOPS European Conference on Computer Systems)	2010, 2011
USENIX (USENIX Annual Technical Conference)	2011
SPIN (International Workshop on Model Checking of Software)	2011

Patents

Parallel Automated Testing Platform for Software Systems
Stefan Bucur, George Candea, Cristian Zamfir, *US Patent No. 8,863,096 (October 2014)*

Advantageous State Merging During Symbolic Analysis
Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, George Candea *(pending)*

Miscellaneous

Languages: English (fluent), Romanian (fluent), French (good)

Hobbies: rock climbing, photography, classical guitar, board games, table tennis.