# BUILDING ABSTRACTIONS FOR FAST, SECURE, RELIABLE COMPUTER SYSTEMS

BY

HAOHUI MAI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Samuel T.King, Chair, Director of Research
Professor P. Brighten Godfrey
Professor Matthew Caesar
Professor Madhusudan Parthasarathy
Professor Nickolai Zeldovich, MIT

# Abstract

Modern computer systems play important roles in our society and everyday lives. Their performance, security and reliability are of critical importance. Real-world computer systems, however, occasionally suffer from performance degradation, security exploits, and poor reliability, because of the lack of efficient automatic analyses.

This dissertation introduces a new methodology for building efficient automatic analyses for real-world computer systems through identifying and designing proper abstractions. It demonstrates the methodology within the context of three real-world computer systems: detecting network defects at the data plane level, exploiting data parallelism in web pages, and formally verifying security invariants in operating system kernels.

This dissertation presents the design, implementation, and evaluation of the above systems, and shows that choosing the proper set of abstractions is an essential step to constructing efficient automatic analyses for real-world computer systems. Moreover, these analyses can become valuable tools to improve the performance, security and reliability of computer systems.

# Acknowledgments

I want to thank my undergraduate supervisor Wenguang Chen, who introduced me to the academic research when I was an undergraduate in Tsinghua University. I also want to thank my colleagues in Microsoft Research Asia (especially my mentor Xuezheng Liu) during my days as an undergraduate intern there. They led me to the path of doing system research.

I am fortunate to work with and around many bright people at University of Illinois. I want to thank Professors Vikram Adve and Yuanyuan Zhou, for their guidance during my early years. I also want to thank my many co-workers over the years: Anthony Cozzie, Nathan Dautenhahn, Murph Finnicum, Matthew Hicks, Shuo Tang, and Hui Xue, for many hours of discussions raning from research to sports. I also want to thank Xi Wang and Zhilei Xu for many inspiring discussions on research. These discussions benefit both my research and me personally a lot.

I want to thank my research collaborators: Rachit Agarwal, Calin Cascaval, Ahmed Khurshid, Pablo Montesinos, Shankar Pasupathy, Edgar Pek, Weiwei Xiong, and Professors Matthew Caesar, Brighten Godfrey, Madhusudan Parthasarathy, Lin Tan, Ding Yuan, and Yuanyuan Zhou, who made doing interesting work and creative ideas possible.

The largest thanks go to my advisor Professor Sam King. Beyond teaching me how to do research, he served as a good example of staying persistent and focused on research, and being a joyful person in everyday lives.

Lastly, I thank my committee: Professors Brighten Godfrey, Matthew Caesar, Madhusudan Parthasarathy, Nickolai Zeldovich for their support and direction.

*To my family*

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Modern computer systems continue to reshape our society and our everyday lives. At the center of this revolution are computer networks and mobile devices, which put a wealth of information and ever-increasing opportunities for social interaction at the fingertips of users. The performance, security and reliability of these systems are of critical importance – operation practice in industry shows that inefficiency, vulnerability, and service outages can be costly [71].

Unfortunately, it is common to see real-world computer systems occasionally suffer from performance degradation [80, 83], security vulnerabilities [24, 66, 82], and poor reliability [81]. Preventing, detecting, and diagnosing on these problems are difficult, because *these systems lack efficient automatic analyses in general*. Real-world computer systems are highly complex artifacts, which contain tremendous amounts of code and convoluted interactions between hundreds of different components. For example, Android 4.0 has more than 13 million lines of code, which includes the Linux kernel, the Dalvik virtual machine, etc. Also, the UIUC campus network uses several hundreds network devices from multiple vendors, performing diverse codependent functions such as routing, switching, and access control across physical and virtual networks (VPNs and VLANs). In summary, the scales and complexity of real-world computer systems impose significant challenges for constructing automatic analyses for them.

This dissertation introduces a new methodology for *building efficient automatic analyses for real-world computer systems through identifying and designing proper abstractions*. We demonstrate the methodology within the context of three real-world computer systems: detecting network defects at the data plane level, exploiting data parallelism in web pages, and formally verifying security invariants in operating system kernels. Our experience shows that choosing proper abstractions is a general and powerful approach to construct automatic analyses for real systems, and to improve their performance, security and reliability.

The following sections outline these chapters.

## 1.1 Detecting network defects at the data plane level

Modern enterprise networks are complex. As in any complex computer system, enterprise networks are prone to a wide range of errors [26, 36, 39, 41, 79, 80, 100, 121, 127], such as misconfiguration, software bugs, or unexpected interactions across protocols. These errors can lead to oscillations, black holes, faulty advertisements, or route leaks that ultimately cause disconnectivity and security vulnerabilities.

Diagnosing issues in these complex heterogeneous network environments demands an efficient automatic tool. One question that naturally arises is that *which level of abstractions would allow an automatic tool to reason about computer networks efficiently?*

Previous work focuses on building automatic tools that analyzes the configuration files of network devices [41, 130]. While useful, these tools have two limitations stemming from their analyses of high-level configuration files. First, configuration analysis cannot find bugs in router software, which interprets and acts on those configuration files. Both commercial and open source router software regularly exhibit bugs that affect network availability or security [127] and have led to multiple high-profile outages and vulnerabilities [36, 135]. Second, configuration analysis must model complex configuration languages and dynamic protocol behavior in order to determine the ultimate effect of a configuration. As a result, these tools generally focus on checking correctness of a single protocol such as BGP [41, 42] or firewalls [1, 130]. Such diagnosis will be unable to reason about interactions that span multiple protocols, and may have difficulty dealing with the diversity in configuration languages from different vendors making up typical networks.

Chapter 2 describes a different approach that diagnoses problems as close as possible to the network's actual behavior through *formal analysis of data plane state*. This approach has two main advantages. First, by checking the results of routing software rather than its inputs, we can catch bugs that are invisible at the level of configuration files. Second, it becomes easier to perform unified analysis of a network across many protocols and implementations, because data plane analysis avoids modeling dynamic routing protocols and operates on comparatively simple input formats that are common across many protocols and implementations.

The chapter describes the design, implementation, and evaluation of our research prototype, Anteater, a tool that analyzes the data plane state of network devices. Anteater detects network defects through data plane analysis. It models the network and the analysis as SAT problems, then it leverages the power of an off-the-shelf SAT solver to analyze the networks. Finally, it reports to the operators if any network defects have been detected.

## 1.2 Exploiting data parallelism in web pages

The performance of mobile web browsing is of critical importance [87, 104, 126]. Although current mobile platforms continue to increase their processing power by introducing more CPU cores [16, 103], commodity mobile web browsers receive few performance gains due to their single-threaded, event-driven architectures.

Parallelization is an appealing solution to utilize current multi-core mobile hardware platforms, and to improve the performance of mobile web browsing. The question is that *what should be parallelized?*

Previous work focuses on building parallel browsers for multi-core mobile platforms, which includes parallel layout algorithms [6, 90], and applying task-level parallelism to the browser [20, 51]. These special cases, however, only speed up web apps that make heavy use of specific features (e.g., cascading style sheets (CSS)), or are limited to the tasks that the browser developers identify ahead of time. Unfortunately, years of sequential optimization, the sheer size of modern browsers (e.g., Firefox has over three million lines of code), and the fundamentally single-threaded event-driven programming model of modern browsers make it challenging to generalize this approach to refactor today's browsers into fully multi-threaded parallel applications.

Chapter 3 describes an approach that focuses on *parallelizing web pages*. By taking a holistic approach, we anticipate an architecture that can work on a wide range of existing commodity browsers with only a few minor changes to their implementation, rather than a major refactoring of existing browsers or a re-implementation of these mature and feature-rich applications.

The chapter describes the design, implementation, and evaluation of our research prototype, Peregrine, a system that extracts data parallelism from web pages, and speeds up web browsing for multi-core mobile devices like smart phones and tablets. Peregrine consists a server-side preprocessor and a client-side browser. The server-side component decomposes the web pages to loosely coupled sub pages, each of which is a "complete" web page that consists of HTML, JavaScript, CSS, etc. The client side browser renders these sub pages concurrently in separate processes to fully utilize the multi-core hardware.

## 1.3 Formally verifying security invariants in operating system kernels

Modern mobile devices such as smart phones and tablets give people a nearly constant connection to the Internet. Applications running on these devices provide users with a wide range of functionality, but vulnerabilities and exploits in their software stacks pose a real threat to the security and privacy of modern mobile systems [24, 66].

The security and privacy of an application relies on the security guarantees provided by the underlying systems. The question is that *what are the proper levels of abstractions of security guarantees?*

Much of the current work provides secure abstractions at the hypervisor level, which includes encrypting virtual memory and persistent storage [25, 54, 133], attestation on executions [88, 105], and virtualizing mobile hardware [3]. These systems, however, suffer from two limitations due to the unavailability of the application-level information. First, they have to make over-pessimistic assumptions due to the unavailability of application-level information, which leads to additional overheads. For example, Overshadow [25] has to encrypt all virtual memory and I/O requests although arguably only the private data of the applications needs to be protected. Second, the semantic gaps between the abstractions of hypervisors and the system calls make it difficult to provide assurance of properties that are implemented at the user-level (e.g., properties about UI state), or to prevent Iago attacks [23].

Chapter 4 describes a direct approach that enforces security at the system call level. It describes an operating system prototype called ExpressOS, which provides *application-level security invariants* for application security policies, and a verified implementation of the mechanisms that ExpressOS uses to enforce its security policies. Our design includes an OS architecture for coping with legacy hardware safely, a programming language and run-time system for building our operating system, and a set of proofs on our kernel implementation that provide high assurance that our system can uphold the security invariants we define.

The proofs focus on eight security invariants covering secure storage, memory isolation, address space integrity, user interface (UI) isolation, and secure IPC. By proving these invariants, ExpressOS enables sensitive applications to provably isolate their state (mostly ensuring integrity of state; confidentiality is also ensured to a certain degree, but side-channel attacks are not provably prevented), and run-time events from malicious applications running on the same device.

The chapter also describes the implementation and evaluation of ExpressOS. We show that ExpressOS is realistic enough to run real-world Android application like a web browser. Then we quantitatively analyze the security of the approach of ExpressOS. Finally we show that ExpressOS is a practical approach to improve the security of commodity mobile operating systems.

## 1.4 Contributions

This dissertation shows that *choosing the proper set of abstractions is an essential step to building fast, secure, and reliable computer systems*. These analyses can later become valuable tools to improve the performance, security and reliability of the systems. It makes a number of contributions.

**First data plane analysis.** Anteater is the first design and implementation of a data plane analysis system used to find real bugs in real networks. We demonstrate how to express data plane state and network invariants as SAT problems, and propose a novel algorithm for handling packet transformations.

We develop optimizations to our algorithms and implementation to enable Anteater to check invariants efficiently using a SAT solver, and demonstrate experimentally that Anteater is sufficiently scalable to be a practical tool. We applied Anteater on the UIUC campus network and revealed 23 real bugs in less than an hour.

**First data-parallel web browser.** Peregrine is the first system to decompose web pages into loosely coupled mini pages to enable parallel processing on the client. The Peregrine browser loads mini pages in parallel while maintaining correct JavaScript semantics and browser display state.

The page decomposition algorithm is the first browser-processing algorithm that is capable of eliminating JavaScript serialization for at least part of the DOM. The algorithm preserves JavaScript semantics while splitting the DOM into a JavaScript dependent page and several JavaScript independent mini pages.

We evaluate Peregrine and show empirically that Peregrine can improve the performance of mobile web browsing. For one experiment, Peregrine speeds up web browsing up to 3.95x, reducing the page load latency time up to 14.9 seconds. Out of the 170 popular web sites we evaluated, Peregrine speeds up 151 out of 170 (89%) sites, and reduces the latency for 39 (23%) sites by two seconds.

**First large-scale OS kernel that provides formally verified security invariants.** ExpressOS is the first OS architecture that provides verifiable, high-level abstractions for building mobile applications. Our experience with ExpressOS shows that verifying security invariants in a practical large-scale system is feasible with the help of new programming languages and minor source code annotations ($\sim 2.8\%$ annotation overhead).

Our experience and evaluation of ExpressOS shows that this type of OS can be practical and it improves the security of software systems running on mobile devices. In one test, we ran the same web browser on ExpressOS and on an Android-based system, and found that ExpressOS adds 16% overhead on average to the page load latency time for nine popular web sites. We also examined 383 recent vulnerabilities listed in CVE [28]. ExpressOS successfully prevents 364 out of 383 (95%) of them.

## 1.5 Dissertation plan

This dissertation proceeds as follows. Chapter 2 describes the design and implementation of Anteater, an automatic tool that analyzes the data plane state of the network. It also describes the evaluation of Anteater on the UIUC campus network. Chapter 3 details the work related to Peregrine, a web browser that exploit data parallelism by parallelizing web pages. Chapter 4 discusses the design and implementation of ExpressOS, an operating system that provides high assurance security invariants at the application level. Finally, we conclude with future directions in Chapter 5.

# Chapter 2

# Detecting network defects at the data plane level

## 2.1 Introduction

Chapter 1 have established the importance of automatic tools for network diagnosis, and the advantage of data plane analysis. The remaining question is that whether it is practical to diagnose real-world network problems?

Our experience shows that the answer is yes. This chapter describes the design, implementation, and evaluation of Anteater, a tool that analyzes the data plane state of network devices. Anteater collects the network topology and devices' forwarding information bases (FIBs), and represents them as boolean functions. The network operator specifies an invariant to be checked against the network, such as reachability, loop-free forwarding, or consistency of forwarding rules between routers. Anteater combines the invariant and the data plane state into instances of boolean satisfiability problem (SAT), and uses a SAT solver to perform analysis. If the network state violates an invariant, Anteater provides a specific counterexample — such as a packet header, FIB entries, and path — that triggers the potential bug.

We applied Anteater to a large university campus network, analyzing the FIBs of $178$ routers that support over $70,000$ end-user machines and servers, with FIB entries inserted by a combination of BGP, OSPF, and static ACLs and routes. Anteater revealed 23 confirmed bugs in the campus network, including forwarding loops and stale ACL rules. Nine of these faults are being fixed by campus network operators. For example, Anteater detected a forwarding loop between a pair of routers that was unintentionally introduced after a network upgrade and had been present in the network for over a month. These results demonstrate the utility of the approach of data plane analysis.

## 2.2 Overview of architecture

Anteater's primary goal is to detect and diagnose a broad, general class of network problems. The system detects problems by analyzing the contents of forwarding tables contained in routers, switches, firewalls, and other networking equipment (Figure 2.1). Operators use Anteater to check whether the network conforms to a set of *invariants* (i.e., correctness conditions regarding the network's forwarding behavior). Violations of these invariants usually indicate a bug in the network. Here are a few examples of invariants:

- Loop-free forwarding. There should not exist any packet that could be injected into the network that would cause a forwarding loop.
- Connectivity. All computers in the campus network are able to access both the intranet and the Internet, while respecting network policies such as access control lists.
- Consistency. The policies of two replicated routers should have the same forwarding behavior. More concretely, the possible set of packets that can reach the external network through them are the same.

Anteater checks invariants through several steps. First, Anteater collects the contents of FIBs from networking equipment through vtys (terminals), SNMP, or control sessions maintained to routers [40, 73]. These FIBs may be simple IP longest prefix match rules, or more complex actions like access control lists or modifications of the packet header [60, 63, 89]. Second, the operator creates new invariants or selects from a menu of standard invariants to be checked against the network. This is done via bindings in Ruby or in a declarative language that we designed to streamline the expression of invariants. Third, Anteater translates both the FIBs and invariants into instances of SAT, which are resolved by an off-the-shelf SAT solver. Finally, if the results from the SAT solver indicate that the supplied invariants are violated, Anteater will derive a counterexample to help diagnosis.

The next section describes the design and implementation in more detail, including writing invariants, translating the invariants and the network into instances of SAT, and solving them efficiently.

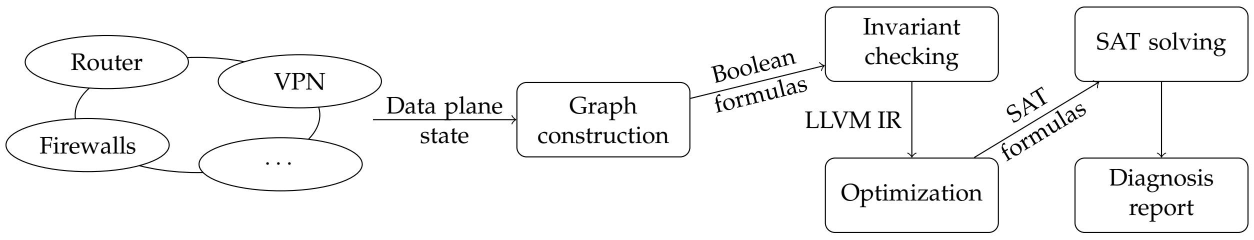


Figure 2.1: The work flow of Anteater. Ovals are network devices. Rectangles are stages in the work flow. Text on the edges shows the type of data flowing between stages.

## 2.3 Design

A SAT problem evaluates a set of boolean formulas to determine if there exists at least one variable assignment such that all formulas evaluate to true. If such an assignment exists, then the set of formulas are *satisfiable*; otherwise they are *unsatisfiable*.

SAT is an NP-complete problem. Specialized tools called SAT solvers, however, use heuristics to solve SAT efficiently in some cases [17]. Engineers use SAT solvers in a number of different problem domains, including model checking, hardware verification, and program analysis. Please see Section 2.6 for more details.

Network reachability can, in the general case, also be NP-complete (see Appendix). We cast network reachability and other network invariants as SAT problems. In this section we discuss our model for network policies, and our algorithms for detecting bugs using sets of boolean formulas and a SAT solver.

Anteater uses an existing theoretical algorithm for checking reachability [123], and we use this reachability algorithm to design our own algorithms for detecting forwarding loops, detecting packet loss (i.e., "black holes"), and checking forwarding consistency between routers. Also, we present a novel algorithm for handling arbitrary packet transformations.

### 2.3.1 Modeling network behavior

Table 2.1 shows our notation. A network $G$ is a 3-tuple $G = (V, E, \mathcal{P})$, where $V$ is the set of networking devices and possible destinations, $E$ is the set of directed edges representing connections between vertices. $\mathcal{P}$ is a function defined on $E$ to represent general policies.

| Symbol | Description |
|---|---|
| $G$ | Network graph $(V, E, \mathcal{P})$ |
| $V$ | Vertices (e.g., devices) in $G$ |
| $E$ | Directed edges in $G$ |
| $\mathcal{P}$ | Policy function for edges |

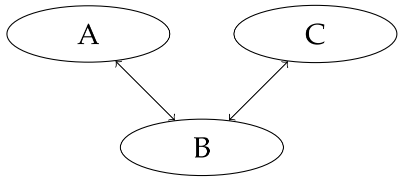
Table 2.1: Notation used in Section 2.3.

Since many of the formulas we discuss deal with IP prefix matching, we introduce the notation $var =_{width} prefix$ to simplify our discussion. This notation is a convenient way of writing a boolean formula saying that the first $width$ bits of the variable $var$ are the same as those of $prefix$. For example, $dst\_ip =_{24} 10.1.3.0$ is a boolean formula testing the equality between the first 24 bits of $dst\_ip$ and 10.1.3.0. The notion $var \neq_{width} prefix$ is the negation of $var =_{width} prefix$.

For each edge $(u, v)$, we define $\mathcal{P}(u, v)$ as the policy for packets traveling from $u$ to $v$, represented as a boolean formula over a symbolic packet. A *symbolic packet* is a set of variables representing the values of fields in packets, like the MAC address, IP address, and port number. A packet can flow over an edge if and only if it satisfies the corresponding boolean formulas. We use this function to represent general policies including forwarding, packet filtering, and transformations of the packet. $\mathcal{P}(u, v)$ is the conjunction (logical $and$) over all policies' constraints on symbolic packets from node $u$ to node $v$.

$\mathcal{P}(u, v)$ can be used to represent a filter. For example, in Figure 2.2 the filtering rule on edge $(B, C)$ blocks all packets destined to 10.1.3.128/25; thus, $\mathcal{P}(B, C)$ has $dst\_ip \neq_{25} 10.1.3.128$ as a part of it. Forwarding is represented as a constraint as well: $\mathcal{P}(u, v)$ will be constrained to include only those symbolic packets that router $u$ would forward to router $v$. The sub-formula $dst\_ip =_{24} 10.1.3.0$ in $\mathcal{P}(B, C)$ in Figure 2.2 is an example.

Packet transformations – for example, setting a quality of service bit, or tunneling the packet by adding a new header – might appear different since they intuitively modify the symbolic packet rather than just constraining it. Somewhat surprisingly, we can represent transformations as constraints too, through a technique that we present in Section 2.3.4.

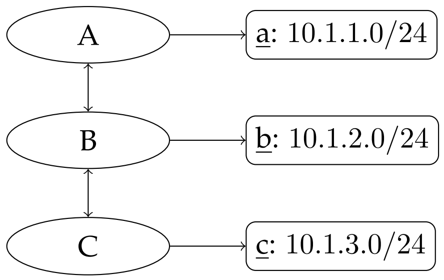


A:
10.1.1.0/24 → DIRECT
10.1.2.0/24 → B
10.1.3.0/24 → B

B:
10.1.1.0/24 → A
10.1.2.0/24 → DIRECT
10.1.3.0/24 → C

C:
10.1.1.0/24 → B
10.1.2.0/24 → B
10.1.3.0/24 → DIRECT

B → C:
10.1.3.128/25 → DROP



$$
\begin{aligned}
\mathcal{P}(A,a) &= \mathit{dst\_ip} =_{24} 10.1.1.0 \\
\mathcal{P}(A,B) &= \mathit{dst\_ip} =_{24} 10.1.2.0 \vee \mathit{dst\_ip} =_{24} 10.1.3.0 \\
\mathcal{P}(B,A) &= \mathit{dst\_ip} =_{24} 10.1.1.0 \\
\mathcal{P}(B,b) &= \mathit{dst\_ip} =_{24} 10.1.2.0 \\
\mathcal{P}(B,C) &= \mathit{dst\_ip} =_{24} 10.1.3.0 \wedge \mathit{dst\_ip} \neq_{25} 10.1.3.128 \\
\mathcal{P}(C,B) &= \mathit{dst\_ip} =_{24} 10.1.1.0 \vee \mathit{dst\_ip} =_{24} 10.1.2.0 \\
\mathcal{P}(C,c) &= \mathit{dst\_ip} =_{24} 10.1.3.0
\end{aligned}
$$

Figure 2.2: An example of a 3-node IP network. *Top:* Network topology, with FIBs in dashed boxes. *Bottom:* graph used to model network behavior. Ovals represent networking equipment; rounded rectangles represent special vertices such as destinations, labeled by lower case letters. The lower half of the bottom figure shows the value of $\mathcal{P}$ for each edge in the graph.

### 2.3.2 Checking reachability

In this subsection, we describe how Anteater checks the most basic invariant: reachability. The next subsection, then, uses this algorithm to check higher-level invariants.

Recall that vertices $V$ correspond to devices or destinations in the network. Given two vertices $s, t \in V$, we define the $s$-$t$ reachability problem as deciding whether there exists a packet that can be forwarded from $s$ to $t$. More formally, the problem is to decide if there exists a symbolic packet $p$ and an $s \rightsquigarrow t$ path such that $p$ satisfies all constraints $\mathcal{P}$ along the edges of the path. Figure 2.3 shows a dynamic programming algorithm to calculate a boolean formula $f$ representing reachability from $s$ to $t$. The boolean formula $f$ has a satisfying assignment if and only if there exists a packet that can be routed from $s$ to $t$ in at most $k$ hops. This part of Anteater is similar to an algorithm proposed by Xie et al. [123], expressed as constraints rather than sets of packets.

To guarantee that all reachability is discovered, one would pick in the worst case $k = n - 1$

```
**function** reach($s, t, k, G$)
$r[t][0] \leftarrow$ true
$r[v][0] \leftarrow$ false for all $v \in V(G) \setminus t$
**for** $i = 1$ to $k$ **do**
   **for all** $v \in V(G) \setminus t$ **do**
      $r[v][i] \leftarrow \bigvee_{(v,u)\in E(G)} (\mathcal{P}(v,u) \wedge r[u][i-1])$
   **end for**
**end for**
**return** $\bigvee_{1 \le i \le k} r[s][i]$
```

Figure 2.3: Algorithm to compute a boolean formula representing reachability from $s$ to $t$ in at most $k$ hops in network graph $G$.

where $n$ is the number of network devices modeled in $G$. A much smaller $k$ may suffice in practice because path lengths are expected to be smaller than $n-1$.

We give an example run of the algorithm for the network of Figure 2.2. Suppose we want to check reachability from $A$ to $C$. Here $k = 2$ suffices since there are only 3 devices. Anteater initializes $\mathcal{P}$ as shown in Figure 2.2 and the algorithm initializes $s \leftarrow A$, $t \leftarrow C$, $k \leftarrow 3$, $r[C][0] \leftarrow$ true, $r[A][0] \leftarrow$ false, and $r[B][0] \leftarrow$ false. After the first iteration of the outer loop we have:

$$\begin{aligned} r[A][1] &= \text{false} \\ r[B][1] &= \mathcal{P}(B, C) \\ &= (dst\_ip =_{24} 10.1.3.0 \wedge dst\_ip \neq_{25} 10.1.3.128) \end{aligned}$$

After the second iteration we have:

$$\begin{aligned} r[A][2] &= r[B][1] \wedge \mathcal{P}(A, B) \\ &= dst\_ip =_{24} 10.1.3.0 \ \wedge dst\_ip \neq_{25} 10.1.3.128 \wedge \\ &\quad (dst\_ip =_{24} 10.1.2.0 \vee dst\_ip =_{24} 10.1.3.0) \\ r[B][2] &= \text{false} \end{aligned}$$

```
**function** loop($v, G$)
$v' \leftarrow$ a new vertex in $V(G)$
**for all** $(u, v) \in E(G)$ **do**
    $E(G) \leftarrow E(G) \cup \{(u, v')\}$
    $\mathcal{P}(u, v') \leftarrow \mathcal{P}(u, v)$
**end for**
Test satisfiability of reach($v$, $v'$, $|V(G)|$, $G$)
```

Figure 2.4: Algorithm to detect forwarding loops involving vertex $v$ in network $G$.

```
**function** packet_loss($v, D, G$)
$n \leftarrow$ the number of network devices in $G$
$d \leftarrow$ a new vertex in $V(G)$
**for all** $u \in D$ **do**
    $(u, d) \leftarrow$ a new edge in $E(G)$
    $\mathcal{P}(u, d) \leftarrow$ true
**end for**
$c \leftarrow$ reach($v$, $d$, $n$, $G$)
Test satisfiability of $\neg c$
```

Figure 2.5: Algorithm to check whether packets starting at $v$ are dropped without reaching any of the destinations $D$ in network $G$.

The algorithm then returns the formula $r[A][1] \vee r[A][2]$.

### 2.3.3 Checking forwarding loops, packet loss, and consistency

The reachability algorithm can be used as a building block to check other invariants.

**Loops.** Figure 2.4 shows Anteater's algorithm for detecting forwarding loops involving vertex $v$. The basic idea of the algorithm is to modify the network graph by creating a dummy vertex $v'$ that can receive the same set of packets as $v$ (i.e., $v$ and $v'$ have the same set of incoming edges and edge policies). Thus, $v$-$v'$ reachability corresponds to a forwarding loop. The algorithm can be run for each vertex $v$. Anteater thus either verifies that the network is loop-free, or returns an example of a loop.

**Packet loss.** Another property of interest is whether "black holes" exist: i.e., whether packets may be lost without reaching any destination. Figure 2.5 shows Anteater's algorithm for checking whether packets from a vertex $v$ could be lost before reaching a given set of destinations $D$, which can be picked as (for example) the set of all local destination prefixes plus external routers. The idea is to add a "sink" vertex $d$ which is reachable from all of $D$, and then (in the algorithm's last line) test the *absence* of $v$-$d$ reachability. This will produce an example of a packet that is dropped or confirm that none exists.[1] Of course, in some cases packet loss is the correct behavior. For example, in the campus network we tested, some destinations are filtered due to security concerns. Our implementation allows operators to specify lists of IP addresses or other conditions that are intentionally not reachable; Anteater will then look for packets that are unintentionally black-holed. We omit this extension from Figure 2.5 for simplicity.

**Consistency.** Networks commonly have devices that are expected to have identical forwarding policy, so any differing behavior may indicate a bug. Suppose, for example, that the operator wishes to test if two vertices $v_1$ and $v_2$ will drop the same set of packets. This can be done by running `packet_loss` to construct two formulas $c_1 = \texttt{packet\_loss}(v_1, D, G)$ and $c_2 = \texttt{packet\_loss}(v_2, D, G)$, and testing satisfiability of ($c_1$ xor $c_2$). This offers the operator a convenient way to find potential bugs without specifically listing the set of packets that are intentionally dropped. Other notions of consistency (e.g., based on reachability to specific destinations) can be computed analogously.

### 2.3.4 Packet transformations

The discussion in earlier subsections assumed that packets traversing the network remain unchanged. Numerous protocols, however, employ mechanisms that *transform* packets while they are in flight. For example, MPLS swaps labels, border routers can mark packets to provide QoS services, and packets can be tunneled through virtual links which involves prepending a header. In this subsection, we present a technique that flexibly handles packet transformations.

**Basic technique.** Rather than working with a single symbolic packet, we use a *symbolic packet history*. Specifically, we replace each symbolic packet $s$ with an array $(s_0, \dots, s_k)$ where $s_i$ represents the state of the packet at the $i$th hop. Now, rather than transforming a packet, we can express a transformation as a constraint on its history: a packet transformation $f(\cdot)$ at hop $i$ in-

[1]This loss could be due either to black holes or loops. If black holes specifically are desired, then either the loops can be fixed first, or the algorithm can be rerun with instructions to filter the previous results. We omit the details.

duces the constraint $s_{i+1} = f(s_i)$. For example, an edge traversed by two MPLS label switched paths with incoming labels $\ell_1^{in}, \ell_2^{in}$ and corresponding outgoing labels $\ell_1^{out}, \ell_2^{out}$ would have the transformation constraint

$$\bigvee_{j\in\{1,2\}} \left(s_i.label = \ell_j^{in} \wedge s_{i+1}.label = \ell_j^{out}\right).$$

Another transformation could represent a network address translation (NAT) rule, setting an internal source IP address to an external one:

$$s_{i+1}.source\_ip = 12.34.56.78$$

A NAT rule could be non-deterministic, if a snapshot of the NAT's internal state is not available and it may choose from multiple external IP addresses in a certain prefix. This can be represented by a looser constraint:

$$s_{i+1}.source\_ip =_{24} 12.34.56.0$$

And of course, a link with no transformation simply induces the identity constraint:

$$s_{i+1} = s_i.$$

We let $\mathcal{T}_i(v, w)$ refer to the transformation constraints for packets arriving at $v$ after $i$ hops and continuing to $w$.

**Application to invariant algorithms.** Implementing this technique in our earlier reachability algorithm involves two principal changes. First, we must include the transformation constraints $\mathcal{T}$ in addition to the policy constraints $\mathcal{P}$. Second, the edge policy function $\mathcal{P}(u, v)$, rather than referring to variables in a single symbolic packet $s$, will be applied to various entries of the symbolic packet array $(s_i)$. So it is parameterized with the relevant entry index, which we write as $\mathcal{P}_i(u, v)$; and when computing reachability we must check the appropriate positions of the array.

Incorporating those changes, Line 5 of our reachability algorithm (Fig. 2.3) becomes

$$r[v][i] \leftarrow \bigvee_{(v,u)\in E(G)} \left(\mathcal{T}_{i-1}(v,u) \wedge \mathcal{P}_{i-1}(v,u) \wedge r[u][i-1]\right).$$

The loop detection algorithm, as it simply calls reachability as a subroutine, requires no further changes.

The packet loss and consistency algorithms have a complication: as written, they test satisfiability of the *negation* of a reachability formula. The negation can be satisfied either with a symbolic packet that would be lost in the network, or a symbolic packet history that couldn't have existed because it violates the transformation constraints. We need to differentiate between these, and find only true packet loss. To do this, we avoid negating the formula. Specifically, we modify the network by adding a node $\ell$ acting as a sink for lost packets. For each non-destination node $u$, we add an edge $u \to \ell$ annotated with the constraint that the packet is dropped by $u$ (i.e., the packet violates the policy constraints on all of $u$'s outgoing edges). We also add an edge $w \to \ell$ with no constraint, for each destination node $w \notin D$. We can now check for packet loss starting at $v$ by testing satisfiability of the formula $\texttt{reach}(v, \ell, n-1, G)$ where $n$ is the number of nodes and $G$ is the network modified as described here.

The consistency algorithm encounters a similar problem due to the xor operation, and has a similar solution.

**Notes.** We note two effects which are not true in the simpler transformation-free case. First, the above packet loss algorithm does not find packets which loop (since they never transit to $\ell$); but of course, they can be found separately through our loop-detection algorithm.

Second, computing up to $k = n-1$ hops does not guarantee that all reachability or loops will be discovered. In the transformation-free case, $k = n-1$ was sufficient because after $n-1$ hops the packet must either have been delivered or revisited a node, in which case it will loop indefinitely. But transformations allow the state of a packet to change, so revisiting a node doesn't imply that the packet will loop indefinitely. In theory, packets might travel an arbitrarily large number of hops before being delivered or dropped. However, we expect $k \leq n-1$ to be sufficient in practice.

**Application to other invariants.** Packet transformations enable us to express certain other invari-

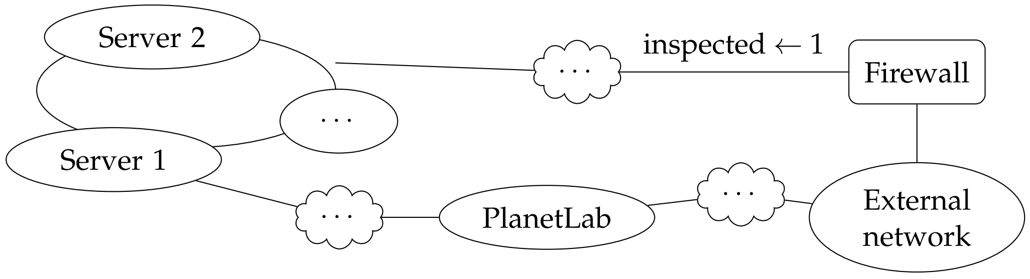


Figure 2.6: An example where packet transformations allow convenient checking of firewall policy. Solid lines are network links; text on the links represents a transformation constraint to express the invariant. Clouds represent omitted components in the network.

ants succinctly. Figure 2.6 shows a simplified version of a real-world example from our campus network. Most servers are connected to the external network via a firewall, but the PlanetLab servers connect to the external network directly. For security purposes, all traffic between campus servers and PlanetLab nodes is routed through the external network, except for administrative links between the PlanetLab nodes and a few trusted servers. One interesting invariant is to check whether all traffic from the external network to protected servers indeed goes through the firewall as intended.

This invariant can be expressed conveniently as follows. We introduce a new field $inspected$ in the symbolic packet, and for each edge $(f, v)$ going from the firewall $f$ towards the internal network of servers, we add a transformation constraint:

$$\mathcal{T}_i(f, v) = s_{i+1}.inspected \leftarrow 1.$$

Then for each internal server $S$, we check whether

$$(s_k.inspected = 0) \wedge R(ext, S)$$

where $ext$ is the node representing the external network, and $R(S, ext)$ is the boolean formula representing reachability from $ext$ to $S$ computed by the `reach` algorithm. If this formula is true, Anteater will give an example of a packet which circumvents the firewall.

## 2.4 Implementation

We implemented Anteater on Linux with about 3,500 lines of C++ and Ruby code, along with roughly 300 lines of auxiliary scripts to canonicalize data plane information from Foundry, Juniper and Cisco routers into a comma-separated value format.

Our Anteater implementation represents boolean functions and formulas in the intermediate representation format of LLVM [74]. LLVM is not essential to Anteater; our invariant algorithms could output SAT formulas directly. But LLVM provides a convenient way to represent SAT formulas as functions, inline these functions, and simplify the resulting formulas.

In particular, Anteater checks an invariant as follows. First, Anteater translates the policy constraints $\mathcal{P}$ and the transformation constraints $\mathcal{T}$ into LLVM functions, whose arguments are the symbolic packets they are constraining. Then Anteater runs the desired invariant algorithm (reachability, loop detection, etc.; Section 2.3), outputting the formula using calls to the $\mathcal{P}$ and $\mathcal{T}$ functions. The resulting formula is stored in the `@main` function. Next, LLVM links together the $\mathcal{P}$, $\mathcal{T}$, and `@main` functions and optimizes when necessary. The result is translated into SAT formulas, which are passed into a SAT solver. Finally, Anteater invokes the SAT solver and reports the results to the operator.

Recall the example presented in Section 2.3.2. We want to check reachability from $A$ to $C$ in Figure 2.2. Anteater translates the policy function $\mathcal{P}(B, C)$ into function `@p_bc()`, and puts the result of dynamic programming algorithm into `@main()`:

```
define i1 @p_bc(%s_i, %s_i+1) {
  %0 = load %s_i.dst_ip
  %1 = and %0, 0xffffff00
  %2 = icmp eq 0xa010300,
  %3 = and %0, 0xffffff80
  %4 = icmp eq 0xa010380, %3
  %5 = xor %4, true
  %6 = and %2, %5
  ret i1 %6
}
```

```
@pkt = external global

define void @main() {
  %0 = call @p_bc(@pkt, @pkt)
  %1 = call @p_ab(@pkt, @pkt)
  %2 = and %0, %1
  call void @assert(%2)
  ret void
}
```

Figure 2.7: LLVM IR used by Anteater when checking reachability of the example in Section 2.3.2.

The function `@p_bc` represents the function

$$\mathcal{P}(B, C) = dst\_ip =_{24} 10.1.3.0 \wedge dst\_ip \neq_{25} 10.1.3.128$$

The function takes two parameters `%`$s_i$ and `%`$s_{i+1}$ to support packet transformations as described in Section 2.3.4.

The `@main` function is shown at the right side of the snippet. `@p_ab` is the LLVM function representing $\mathcal{P}(A, B)$. `@pkt` is a global variable representing a symbolic packet. Since there is no transformation involved, the main function calls the policy functions `@p_bc` and `@p_ab` with the same symbolic packet. The call to `@assert` indicates the final boolean formula to be checked by the SAT solver. Next, LLVM performs standard compiler optimization, including inlining and simplifying expressions, whose results are shown on the left of Figure 2.8:

```
@pkt = external global

define void @main() {
  %0 = load @pkt.dst_ip
  %1 = and %0, 0xffffff00
  %2 = icmp eq %1, 0xa010300
  %3 = and %0, 0xffffff80
  %4 = icmp ne %3, 0xa010380
  %5 = and %2, %4
  %6 = and %0, 0xfffffe00
  %7 = icmp eq %6, 0xa010200
  %8 = and %5, %7
  call void @assert(i1 %8)
  ret void
}
```

```
:formula

 (let (?t1 (bvand p0 0xffffff00))
 (let (?t2 (= ?t1 0x0a010300))
 (let (?t3 (bvand p0 0xffffff80))
 (let (?t4 (not (= ?t3 0x0a010380)))
 (let (?t5 (and ?t2 ?t4))
 (let (?t6 (bvand p0 0xfffffe00))
 (let (?t7 (= ?t6 0x0a010200))
 (let (?t8 (and ?t5 ?t7))
 (?t8))))))))
```

Figure 2.8: Intermediate representations used by the reachability checker. The left and the right parts represent the invariant in LLVM IR and SMTLIB format respectively.

Then the result is directly translated into the input format of the SAT solver, which is shown in the right of Figure 2.8. In this example, it is a one-to-one translation except that `@pkt.dst_ip` is renamed to `p0`. After that, Anteater passes the formula into the SAT solver to determine its satisfiability. If the formula is satisfiable, the SAT solver will output an assignment to `pkt.0/p0`, which is a concrete example (the destination IP in this case) of the packet which satisfies the desired constraint.

Anteater parallelizes the checking to reduce the running time. generates `@main` functions for each instance of the invariant, and check them independently (e.g., for each starting vertex when checking loop-freeness).

Anteater implements language bindings for both Ruby and SLang, a declarative, Prolog-like domain-specific language that we designed for writing customized invariants, and implemented on top of Ruby-Prolog [102]. Operators can express invariants via either Ruby scripts or SLang queries; we found that both of them are able to express the three invariants efficiently. The details of SLang are beyond the scope of this chapter.

## 2.5 Evaluation

Our evaluation of Anteater has three parts. First (Section 2.5.1), we applied Anteater to a large university campus network. Our tests uncovered multiple faults, including forwarding loops, traffic-blocking ACL rules that were no longer needed, and redundant statically-configured FIB entries.

Second (Section 2.5.2), we evaluate how applicable Anteater is to detecting router software bugs by classifying the reported effects of a random sample of bugs from the Quagga Bugzilla database. We find that the majority of these bugs have the potential to produce effects detectable by Anteater.

Third (Section 2.5.3), we conduct a performance and scalability evaluation of Anteater. Anteater takes about half an hour to check for static properties in networks of up to $384$ nodes.

We ran all experiments on a Dell Precision WorkStation T5500 machine running 64-bit CentOS 5. The machine had two $2.4$ GHz quad-core Intel Xeon X5530 CPUs, and $48$ GB of DDR3 RAM. It connected to the campus network via a Gigabit Ethernet channel. Anteater ran on a NFS volume mounted on the machine. The implementation used LLVM 2.9 and JRuby 1.6.2. All SAT queries were resolved by Boolector 1.4.1 with PicoSAT 936 and PrecoSAT 570 [17]. All experiments were conducted under $16$-way parallelism.

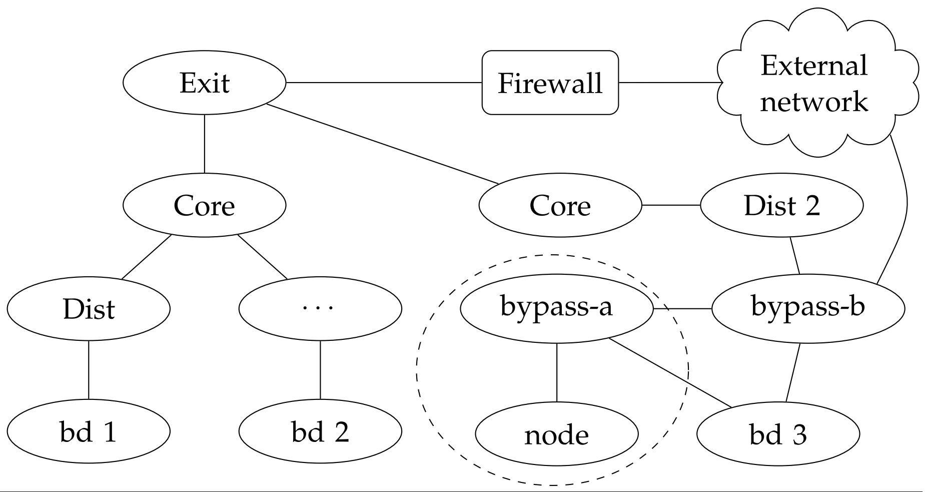
| Invariants | Loops | Packet loss | Consistency |
|---|---|---|---|
| *Alerts* | *9* | *17* | *2* |
| Being fixed | 9 | 0 | 0 |
| Stale configuration | 0 | 13 | 1 |
| False positives | 0 | 4 | 1 |
| Number of runs | 7 | 6 | 6 |

Table 2.2: Summary of evaluation results of Anteater on our campus network.

### 2.5.1 Bugs found in a deployed network

We applied Anteater to our campus network. We collected the IP forwarding tables and access control rules from 178 routers in the campus. The maximal length of loop-free paths in the network is 9. The mean FIB size was 1,627 entries per router, which were inserted by a combination of BGP, OSPF, and static routing. We also used a network-wide map of the campus topology as an additional input.

We implemented the invariants of Section 2.3, and report their evaluation results on our campus network. Table 2.2 reports the number of invariant violations we found with Anteater. The row *Alert* shows the number of distinct violations detected by an invariant, as a bug might violate multiple invariants at the same time. For example, a forwarding loop creating a black hole would be detected by both the invariant for detecting forwarding loops and the invariant for detecting packet loss. We classified these alerts into three categories. First, the row *Being fixed* means the alerts are confirmed as bugs and currently being fixed by our campus network operators. Second, the row *Stale configuration* means that these alerts result from explicit and intentional configuration rules, but rules that are outdated and no longer needed. Our campus network operators decided to not fix these stale configurations immediately, but plan to revisit them during the next major network upgrade. Third, *False positive* means that these alerts flag a configuration that correctly reflected the operator's intent and these alerts are not bugs. Finally, *Number of runs* reports the total number of runs required to issue all alerts; the SAT solver reports only one example violation per run. For each run, we filtered the violations found by previous runs and rechecked the invariants until no violations were reported.



bypass-a:
S 130.126.244.0/22 → node
O 130.126.244.0/23 → node
S 0.0.0.0/0 → bypass-b

node:
C 130.126.244.0/23 → DIRECT
S 0.0.0.0/0 → bypass-a

Figure 2.9: Top: Part of the topology of the campus network. Ovals and solid lines are routers and links respectively. The oval with dashed lines circles the location where a forwarding loop was detected. Bottom: Fragments of data plane information in the network. S stands for static, O stands for OSPF, and C stands for connected.

**Forwarding loops**

Anteater detected nine potential forwarding loops in the network. One of them is shown in Figure 2.9 highlighted by a dashed circle. The loop involved two routers: `node` and `bypass-a`. Router `bypass-a` had a static route for prefix 130.126.244.0/22 towards router `node`. At the same time, Router `node` had a default route towards router `bypass-a`.

According to longest prefix match rules, packets destined to 130.126.244.0/23 from router `bypass-a` could reach the destination. Packets destined to the prefix 130.126.244.0/22 but not in 130.126.244.0/23 would fall into the forwarding loop.

Incidentally, all nine loops happened between these two routers. According to the network operator, router `bd 3` used to connect with router `node` directly, and `node` used to connect with the external network. It was a single choke point to aggregate traffic so that the operator could deploy Intrusion Detection and Prevention (IDP) devices at one single point. The IDP device, however, was unable to keep up after the upgrade, so router `bypass-a` was introduced to offload the traffic. As a side effect, the forwarding loops were also introduced when the operator configured forwarding for that router incorrectly.

These loops are reachable from 64 of 178 routers in the network. All loops have been confirmed by the network operator and they are currently being fixed.

### Packet loss

Anteater issued 17 packet loss alerts, scattered at routers at different levels of hierarchy. One is due to the lack of default routes in the router; three are due to blocking traffic towards unused IP spaces; and the other 13 alerts are because the network blocks traffic towards certain end-hosts.

We recognized that four alerts are legitimate operational practice and classified them as false positives. Further investigation of the other 13 alerts shows that they are stale configuration entries: seven out of 13 are internal IP addresses that were used in the previous generation of the network. The other six blocked IP addresses are external, and they are related to security issues. For example, an external IP was blocked in April 2009 because the host made phishing attempts to the campus e-mail system. The block was placed to defend against the attack without increasing the load on the campus firewalls.

The operator confirmed that these 13 instances can be dated back as early as September 2008 and they are unnecessary, and probably will be removed during next major network upgrade.

### Consistency

Based on conversations with our campus network operators, we know that campus routers in the same level of hierarchy should have identical policies. Hence, we picked one representative router in the hierarchy and checked the consistency between this router and all others at the same level of hierarchy. Anteater issued two new alerts: (1) The two core routers had different policies on IP prefix 10.0.3.0/24; (2) Some building routers had different policies on the private IP address ranges 169.254.0.0/16 and 192.168.0.0/16.

Upon investigating the alert we found that one router exposed its web-based management interface through 10.0.3.0/24. The other alert was due to a legacy issue that could be dated back to the early 1990's: according to the design documents of the campus, 169.254.0.0/16 and

192.168.0.0/16 were intended to be only used within one building. Usually each department had only one building and these IP spaces were used in the whole department. As some departments spanned their offices across more than one building, network operators had to maintain compatibility by allowing this traffic to go one level higher in the hierarchy, and let the router at higher level connect them together by creating a virtual LAN for these buildings.

### 2.5.2 Applicability to router bugs

Like configuration errors, defects in router software might affect the network. These defects tend to be out of the scope of configuration analysis, but Anteater might be able to detect the subset of such defects which manifest themselves in the data plane.

To evaluate the effectiveness of Anteater's data plane analysis approach for catching router software bugs, we studied 78 bugs randomly sampled from the Bugzilla repository of Quagga [99]. Quagga is an open-source software router which is used in both research and production. We studied the same set of bugs presented in [127]. For each bug, we studied whether it could affect the data plane, as well as what invariants are required to detect it. We found 86% (67 out of 78) of the bugs might have visible effects on data plane, and potentially can be detected by Anteater.

*Detectable with `packet_loss` and `loop`.* 60 bugs could be detected by the packet loss detection algorithm, and 46 bugs could be detected by the loop detection algorithm. For example, when under heavy load, Quagga 0.96.5 fails to update the Linux kernel's routing tables after receiving BGP updates (Bug 122). This can result in either black holes or forwarding loops in the data plane, which could be detected by either `packet_loss` or `loop`.

*Detectable with other invariants.* 7 bugs can be detected by other network invariants. For example, in Quagga 0.99.5, a BGP session could remain active after it has been shut down in the control plane (Bug 416). Therefore, packets would continue to follow the path in the data plane, violating the operator's intent. This bug cannot be detected by either `packet_loss` or `loop`, but it is possible to detect it via a customized query: checking that there is no data flow across the given link. We reproduced this bug on a local Quagga testbed and successfully detected it with Anteater.

*No visible data plane effects.* 11 bugs lack visible effects on the data plane. For example, the terminal

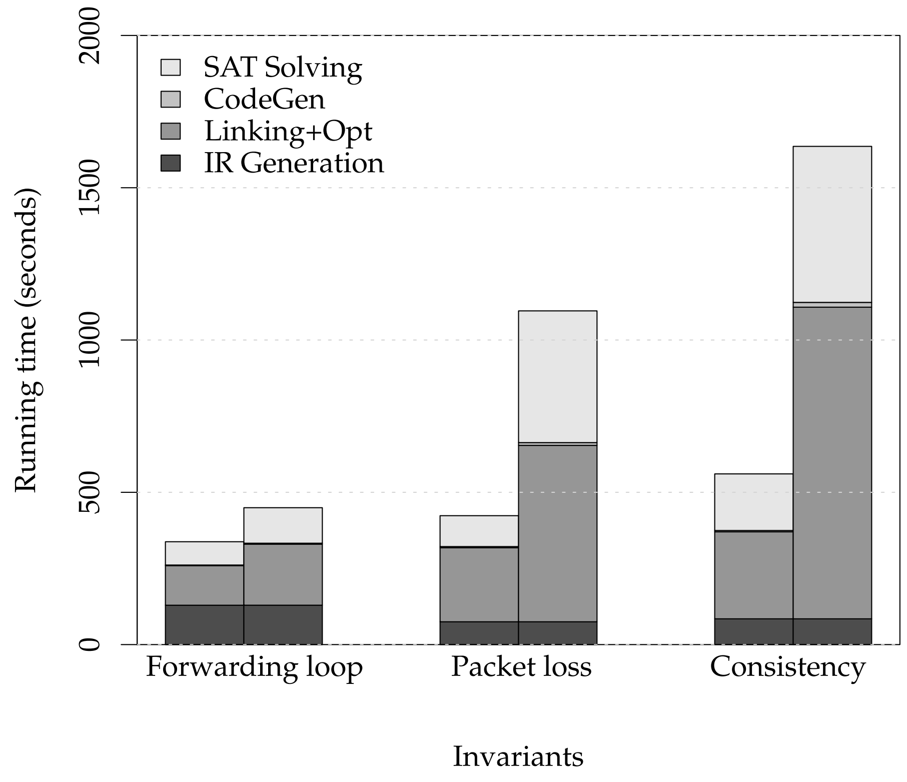


Figure 2.10: Performance of Anteater when checking three invariants. Time is measured by wall-clock seconds. The left and the right column represent the time of the first run and the total running time for each invariant.

hangs in Quagga 0.96.4 during the execution of `show ip bgp` when the data plane has a large number of entries (Bug 87). Anteater is unable to detect this type of bug.

### 2.5.3 Performance and scalability

#### Performance on the campus network

Figure 2.10 shows the total running time of Anteater when checking invariants on the campus network. We present both the time spent on the first run and the total time to issue all alerts.

Anteater's running time can be broken into three parts: (a) compiling and executing the invariant checkers to generate IR; (b) optimizing the IR with LLVM and generating SAT formulas; (c) running the SAT solver to resolve the SAT queries.

The characteristics of the total running time differ for the three invariants. The reason is that a bug has different impact on each invariant; thus the number of routers needed to be checked during the next run varies greatly. For example, if there exists a forwarding loop in the network for some subnet $S$, the loop-free forwarding invariant only reports routers which are involved in the forward loop. Routers that remain unreported are proved to loop-free with respect to the snapshot of data plane, provided that the corresponding SAT queries are unsatisfiable. Therefore, in the next run, Anteater only needs to check those routers which are reported to have a loop. The connectivity and consistency invariants, however, could potentially report that packets destined for the loopy subnet $S$ from all routers are lost, due to the loop. That means potentially all routers must be checked during the next run, resulting in longer run time.

**Scalability**

*Scalability on the campus network.* To evaluate Anteater's scalability, we scaled down the campus network while honoring its hierarchical structure by removing routers at the lowest layer of the hierarchy first, and continuing upwards until a desired number of nodes remain. Figure 2.11 presents the time spent on the first run when running the forwarding loop invariant on different subsets of the campus network.

Figure 2.12 breaks down the running time for IR generation, linking and optimization, and SAT solving. We omit the time of code generation since we found that it is negligible. In Figure 2.12, the running time of first three components are roughly proportional to the square of the number of routers. The running time for SAT solver also roughly fits a quadratic curve, implying that it is able to find heuristics to resolve our queries efficiently for this particular network.

*Scalability on synthesized autonomous system (AS) networks.* We synthesized FIBs for six AS networks (ASes $1221$, $1755$, $3257$, $3967$, $4755$, $6461$) based on topologies from the Rocketfuel project [110], and evaluated the performance of the forwarding loop invariant. We picked $k = 64$ in this experiment. To evaluate how sensitive the invariant is to the complexity of FIB entries, we defined $L$ as a parameter to control the number of "levels" of prefixes in the FIBs. When $L = 1$, all prefixes are non-overlapping $/16$s. When $L = 2$, half of the prefixes (chosen uniform-randomly) are non-overlapping $/16$s, and each of the remaining prefixes is a sub-prefix of one random prefix from the first half — thus exercising the longest-prefix match functionality. For example, with

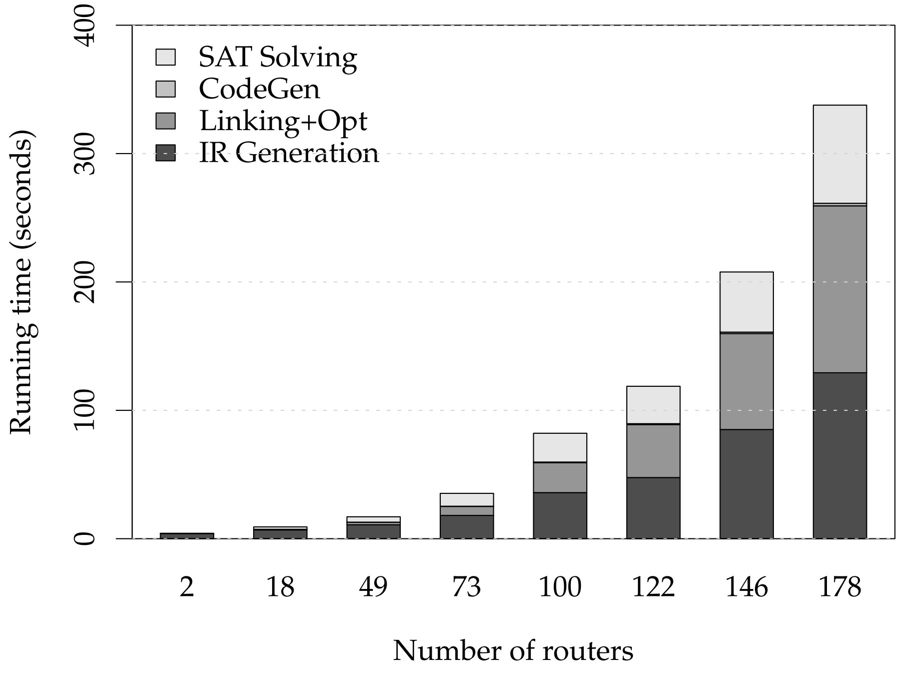


Figure 2.11: Scalability results of the loop-free forwarding invariant on different subsets of the campus network. The parameter $k$ was set to $n - 1$ for each instance.

---

$L = 2$ and two prefixes, we might have $p_1 = 10.1.0.0/16$ and $p_2 = 10.1.1.0/24$. Figure 2.13 shows Anteater's running time on these generated networks; the $L = 2$ case is slightly slower than $L = 1$.

It takes about half an hour for Anteater to check the largest network (AS 1221 with $384$ vertices). These results have a large degree of freedom: they depend on the complexity of network topology and FIB information, and the running time of SAT solvers depends on both heuristics and random number seeds. These results, though inconclusive, indicate that Anteater might be capable of handling larger production networks.

*Scalability on networks with packet transformations.* We evaluated the case of our campus network with NAT devices deployed. We manually injected NAT rules into the data in three steps. First, we picked a set of edge routers. For each router $R$ in the set, we created a phantom router $R'$ which only had a bidirectional link to $R$. Second, we attached a private subnet for each phantom router $R'$, and updated the FIBs of both $R$ and $R'$ accordingly for the private subnet. Finally, we

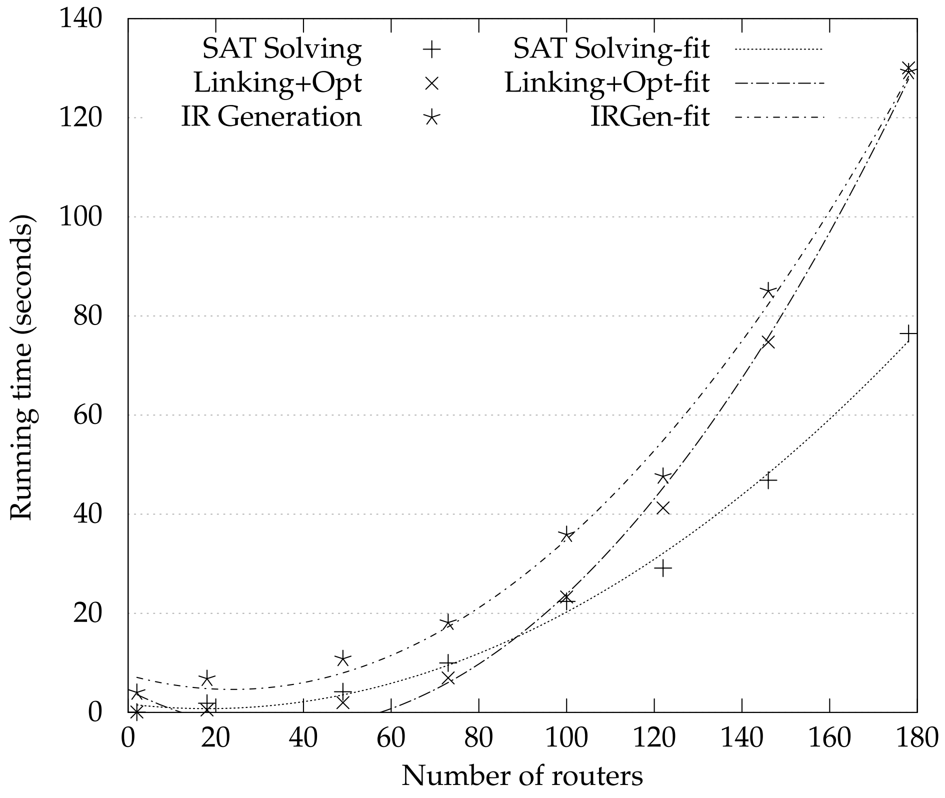


Figure 2.12: Scatter plots for individual components of the data of Figure 2.11. Solid lines are quadratic curves fitted for each category of data points.

---

added NAT rules as described in Section 2.3.4 on the links between $R'$ and $R$.

Figure 2.14 presents the running time of the first run of the loop-free forwarding invariant as a function of the number of routers involved in NAT. We picked the maximum hops $k$ to be $20$ since the maximum length of loop-free paths is $9$ in our campus network.

The portion of time spent in IR generation and code generation is consistent among the different number of NAT-enabled routers. The time spent on linking, optimization and SAT solving, however, increases slowly with the number of NAT-enabled routers.

## 2.6 Related work

*Static analysis of the data plane.* The research most closely related to Anteater performs static analysis of data plane protocols. Xie et al. [123] introduced algorithms to check reachability in

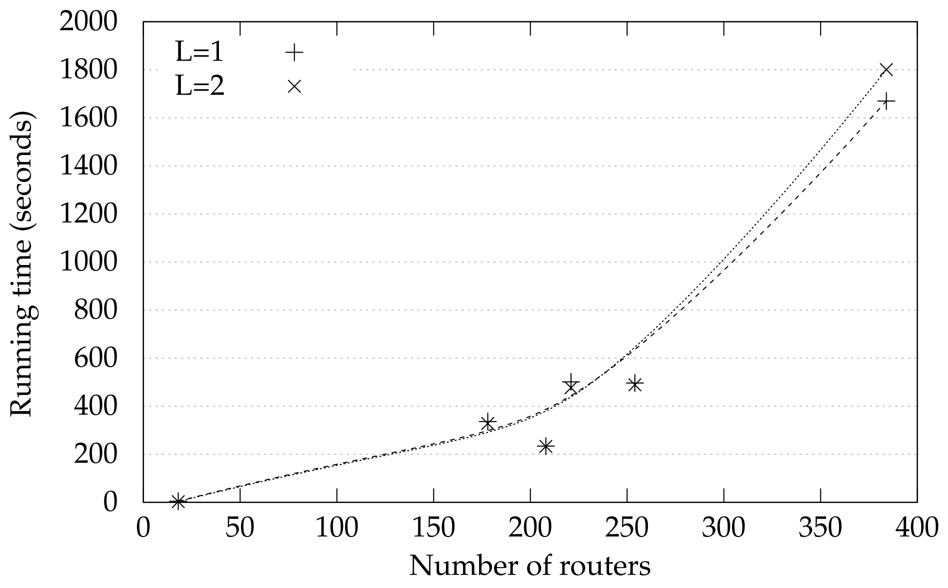


Figure 2.13: Performance of the loop-free forwarding checker on six AS networks from [110]. *L* is the parameter to control the complexity of FIBs. Dots show the running time of the checker for each network. Solid lines are fitted curves generated from the dots.

---

IP networks with support for ACL policies. Their design was a theoretical proposal without an implementation or evaluation. Anteater uses this algorithm, but we show how to make it practical by designing and implementing our own algorithms to use reachability to check meaningful network invariants, developing a system to make these algorithmically complex operations (see the Appendix) tractable, and using Anteater on a real network to find 23 real bugs. Xie et al. also propose an algorithm for handling packet transformations. However, their proposal did not handle fully general transformations, requiring knowledge of an inverse transform function and only handling non-loopy paths. Our novel algorithm handles arbitrary packet transformations (without needing the inverse transform). This distinction becomes important for practical protocols that can cause packets to revisit the same node more than once (e.g., MPLS Fast Reroute).

Roscoe et al. [101] proposed predicate routing to unify the notions of both routing and firewalling into boolean expressions, Bush and Griffin [18] gave a formal model of integrity (including connectivity and isolation) of virtual private routed networks, and Hamed et al. [52] designed algorithms and a system to identify policy conflicts in IPSec, demonstrating bug-finding efficacy in a user study. In contrast, Anteater is a general framework that can be used to check many

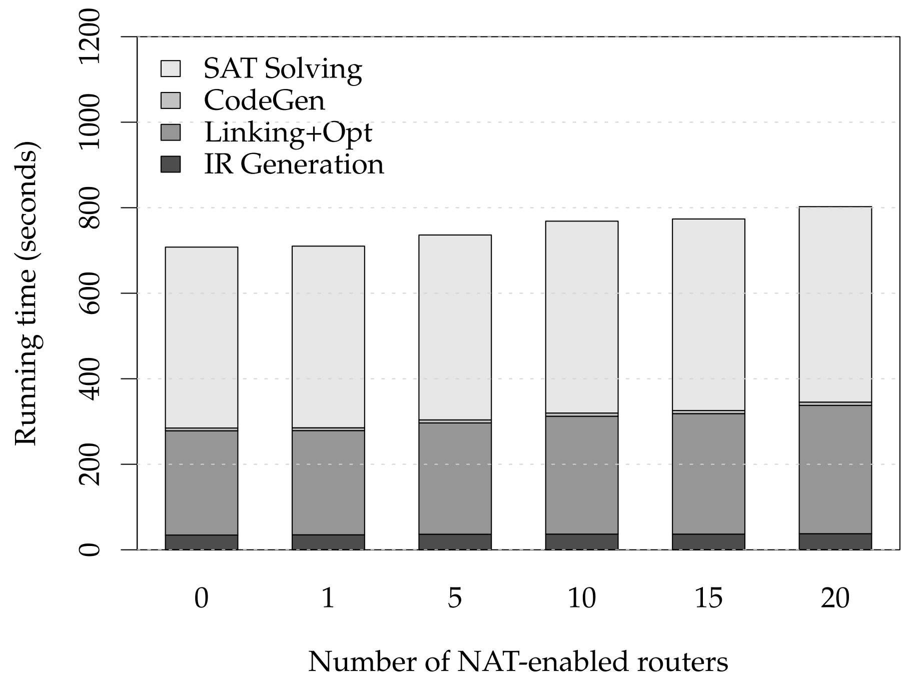


Figure 2.14: Running time of the loop-free forwarding invariant as a function of the number of routers that have NAT rules.

---

protocols, and we have demonstrated that it can find bugs in real deployed networks.

Influenced by Anteater, several projects proposed to model software-defined networks as boolean algebra. Kazemian et al. [65] proposed modeling the data plane as algebras over boolean vectors. Khurshid et al. [67] introduced tailored algorithms to analyze data plane state at real-time.

*Static analysis of control plane configuration.* Analyzing configurations of the control plane, including routers [9, 41] and firewalls [1, 7, 130], can serve as a sanity check prior to deployment. Configuration analysis, however, has two disadvantages. First, it must simulate the behavior of the control plane for the given configuration, making these tools protocol-specific; indeed, the task of parsing configurations is non-trivial and error-prone [84, 127]. Second, configuration analysis will miss non-configuration errors (e.g., errors in router software and inconsistencies between the control plane and data plane [47, 86, 127]; see our study of such errors in Section 2.5.2).

However, configuration analysis has the potential to detect bugs *before* a new configuration is deployed. Anteater can detect bugs only once they have affected the data plane — though, as we have shown, there are subtle bugs that fall into this category (e.g., router implementation bugs, copying wrong configurations to routers) that only a data plane analysis approach like Anteater can detect. Control plane analysis and Anteater are thus complementary.

*Intercepting control plane dynamics.* Monitoring the dynamics of the control plane can detect a broad class of failures [44, 56] with little overhead, but may miss bugs that only affect the data plane. As above, the approach is complementary to ours.

*Traffic monitoring.* Traffic monitoring is widely used to detect network anomalies as they occur [5, 93, 107, 116]. Anteater's approach is complementary: it can *provably* detect or rule out certain classes of bugs, and it can detect problems that are not being triggered by currently active flows or that do not cause a statistical anomaly in aggregate traffic flow.

*SAT solving in other settings.* Work on model checking, hardware verification and program analysis [12, 124, 129] often encounter problems that are NP-Complete. They are often reduced into SAT problems so that SAT solvers can solve them effectively in practice. This work inspired our approach of using SAT solving to model and analyze data-plane behavior.

## 2.7 Summary

We presented Anteater, a practical system for finding bugs in networks via data plane analysis. Anteater collects data plane information from network devices, models data plane behavior as instances of satisfiability problems, and uses formal analysis techniques to systematically analyze the network. To the best of our knowledge, Anteater is the first design and implementation of a data plane analysis system used to find real bugs in real networks.

We ran Anteater on our campus network and uncovered 23 bugs. Anteater helped our network operators improve the reliability of the campus network. Our study suggests that analyzing data plane information could be a feasible approach to assist debugging today's networks.

# Chapter 3

# Exploiting data parallelism in web pages

## 3.1 Introduction

Chapter 1 discussed the importance of web browsing performance. Recent reports from industry show that performance is critical; Google and Microsoft reported that a 200ms increase in page load latency times resulted in "strong negative impacts", and that delays of under 500ms seconds "impact business metrics" [104].

There are multiple sources of overhead that impinge on the browsing performance. One source of overhead for web-based applications (web apps) is the network [119]. Engineers have attempted to mitigate this source of overhead with increased network bandwidth, prefetching, caching, content delivery networks, and by ordering network requests carefully.

A second and increasing source of overhead for web apps is the client CPU [35, 62]. Web browsers combine a parser (HTML), a layout engine, and a language environment (JavaScript), where the CPU sits squarely on the critical path [6, 45, 90]. Even though the serial performance of mobile CPUs continues to increase, the constraints on mobile device form factors and battery power imposes fundamental limitations on further improvement.

The chapter describes an approach that utilizes the current multi-core hardware platforms in mobile web browsing by *parallelizing the web pages*. We present Peregrine, a system that attempts to speed up web apps for multi-core mobile platforms, like smart phones and tablets. We propose a fundamentally different approach to browser performance: Peregrine consists of two components, a server side pre-processor and a client that renders pages concurrently. The server side of Peregrine decomposes web pages on the server into loosely coupled sub pages, or *mini pages*. The client side processes the mini pages in parallel. Each mini page is a "complete" web page that consists of HTML, JavaScript, CSS, and so on, so Peregrine can download, parse, and render this content in parallel while still using single-threaded and mature browser processes on the client.

Several factors contribute the performance improvement. First, the exploitation of the multiple cores in the client allows mini pages to be rendered concurrently. Second, the server preprocessing of the pages, enabling several optimization that allows the client to avoid processing delays due to network resource availability [57]. In particular, when creating mini pages, the Peregrine server separates JavaScript into one mini page (i.e., the main mini page). This essentially frees the processing on the rest of the mini pages from waiting on JavaScript (JavaScript forces the serialization of the processing because it can change the DOM dynamically). In addition, the Peregrine server inlines other resources, such as CSS styles and simplifies some of the HTML to reduce the amount of processing and waiting on the client.

Peregrine mini pages run as separate processes, but share global data structures and display state. In browsers, the document object model (DOM) data structure and API presents JavaScript code with a programmatic interface to the browser's content. Peregrine spreads the DOM across multiple mini pages, but maintains the global and consistent DOM abstraction for all JavaScript code. Also, Peregrine combines displays from separate mini pages into a single display window. By managing these distributed resources, Peregrine preserves compatibility with current web standards for web developers, and maintains consistent visual outputs for users.

We evaluated the performance of Peregrine on 170 popular web sites on a quad-core ARM platform. For one experiment, Peregrine speeds up web browsing up to 3.95x, reducing the page load latency time up to 14.9 seconds. Among 170 popular web sites, it speeds up 151 out of 170 (89%) sites, and reduces the latency for 39 (23%) sites by two seconds.

## 3.2 Motivation

Previous work already established that the client CPU is on the critical path for web apps [35, 62] and that mobile browsers are CPU bound [90]. In this section we measure the performance of browser, to estimate potential performance improvement coming from next-generation, multi-core mobile platforms.

We picked 170 web pages from the 250 most popular web sites according to Alexa [2], and mirrored them to local network. We ran a QtWebkit-based browser loaded each web page for 20

| Component | % of CPU | 4 cores | 16 cores |
|---|---|---|---|
| *V8* | 16% | 1.13 | 1.17 |
| *X & Kernel* | 17% | 1.14 | 1.19 |
| *Painting* | 10% | 1.08 | 1.10 |
| *libc+Qt* | 25% | 1.23 | 1.31 |
| *CSS* | 4% | 1.03 | 1.04 |
| *Layout/Rendering* | 22% | 1.20 | 1.27 |
| *Other* | 6% | 1.05 | 1.06 |

Table 3.1: Breakdown of CPU time spent on web browsing. The last two columns predict the ideal speed ups with Amdahl's law, assuming that either 4 or 16 cores are available.

times, and instrumented its execution with OProfile [96] to derive the time that spent on different components in the browser. We conducted the experiments on a quad-core, 400 MHz ARM Cortex-A9 development board. For the details of the configuration of the experiment, please refer to Section 3.6.

Table 3.1 categorizes the CPU time spent on web browsing to the six components: (1) the V8 JavaScript engine [50] (*V8*), (2) The Linux kernel and X server (*X & Kernel*), (3) Qt Painting and rendering (*Painting*), (4) libc and other components in Qt (*libc+qt*), (5) CSS Selection (*CSS*), (5) WebKit layout and rendering (*Layout/Rendering*), (6) everything else (*Other*). Figure 3.1 also shows the ideal speed ups based on Amdahl's law when 4 or 16 cores are available, assuming each component can be parallelized completely.

Table 3.1 shows two findings:

- There's no single component to be the bottleneck of the browser.
- Even when many cores are available, speed up ratio for task-level parallelism is moderate (up to 1.31x for 16 cores).

These results are not surprising. Browsers are heavily optimized artifacts after years of production uses. Also, when a single component in the browser does become a bottleneck, browser developers improve its performance, leaving a set of components that all contribute roughly evenly to the overhead of the system, as we observed in our experiments.

The ideal speed up ratios in Table 3.1 suggest that one should look for *data parallelism* when parallelizing web browsers. It's Amdahl's Law that limits the maximum performance gains from

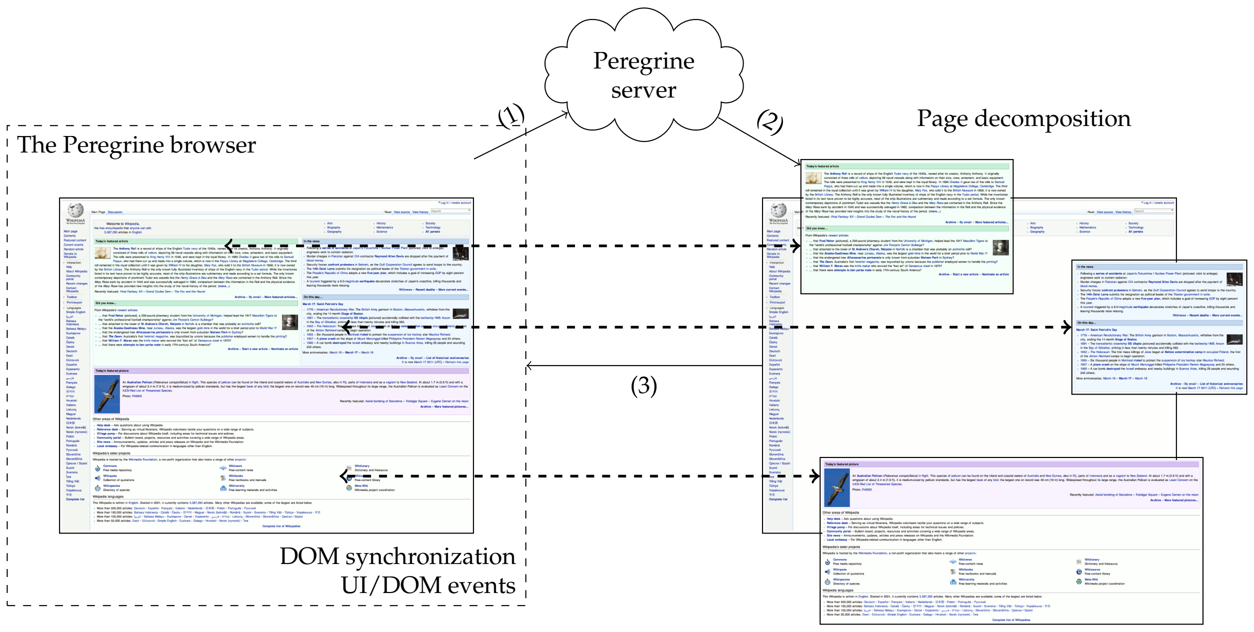


Figure 3.1: Work flows of Peregrine when accessing `wikipedia.org`. Solid lines represent the work flow, dotted lines show the mappings between mini pages. This figure shows how Peregrine browser downloads and renders each mini page while maintaining the proper visual and programmatic semantics. All mini pages run in their own process and the Peregrine browser processes these mini pages in parallel.

---

task-level parallelism. Although researchers reported impressive potential speed ups, say, 80x when parallelizing a single component [45, 90], experiment showed that their effects on overall performace was limited (1.6x for one web site) [6]. Data parallelism has potential larger performance gains as Amdahl's Law applies to data parallelism differently.

## 3.3 Design

This section describes our design for the Peregrine system that strives to improve the performance of web apps by enabling browsers to process web pages in parallel. Four key principles guide our design:

1. *Exploit data parallelism.* The experiment in Section 3.2 suggests that Amdahl's Law limits the potential benefits of task-level parallelism. Exploiting data parallelism doesn't suffer for this limitation thus has higher potential performance gains.

2. *Remove serialization bottlenecks.* Web page processing consists of several components, including JavaScript processing. JavaScript has serialization semantics because it requires the precise DOM context where the script is located and its API permits changing the DOM arbitrarily. In our design, we exploit the client-server architecture to identify Javascript serialization behavior and isolate it into a single mini page, thus allowing other mini pages to run concurrently.

3. *Maintain current web app semantics.* A practical system should be able to work with legacy web apps automatically and any changes should be completely invisible from the user's perspective.

4. *Easily adoptable by other browser implementation.* Browsers are complex artifacts. It's challenging to refactor today's browsers. Peregrine reuses existing mechanisms and implementation whenever possible to make it easily adoptable.

Figure 3.1 shows the work flow when a user accesses `wikipedia.org` with the Peregrine browser. First, the browser issues the request to a Peregrine server. Second, the server decomposes the web page into mini pages. Currently the server uses visual cues of the web page and Javascript semantics to guide the decomposition. Third, the browser downloads, parses, and renders each of these mini pages in separate processes running in parallel. The browser is responsible for properly synchronizing global data structures, such as the DOM tree, and propagating UI and JavaScript events to maintains correct web semantics. In this figure the server decomposes the `wikipedia.org` page into four mini pages and the browser runs four mini page processes in parallel to render the page.

This fundamentally different technique for parallelizing web browsers has four key advantages. First, Peregrine is a data-parallel system [53] where each of mini page process runs the same code but is presented with different data. This technique has more opportunity to scale due to the unique characteristics of the workload of web browsing. Second, decomposition leads to smaller working sets for each mini page, which enables Peregrine to use the "divide and conqueror" strategy to reduce the total amount of work from some tasks, such as layout and rendering. Third, pre-processing the pages on servers opens up the opportunity to shift computation from the client, and minimizing networking delays. For example, the Peregrine server could inline and optimize the CSS stylesheets to reduce both networking and computation overheads. Finally, careful decomposition could potentially remove serialization bottlenecks. Although JavaScript has serialization semantics, Peregrine could isolate JavaScript into a single mini page to allow other mini pages to run concurrently.

Although Peregrine has theoretical advantages, there are a number of key challenges we must overcome to make this system practical. In this section we discuss our techniques for maintaining current JavaScript semantics, maintaining DOM consistency across mini pages, techniques for decomposing web pages into mini pages, and opportunities for pre computing results on the server to reduce the load on the client.

### 3.3.1 Maintaining JavaScript semantics

Our first challenge is maintaining current JavaScript semantics to ensure that Peregrine can process legacy web apps correctly. JavaScript code is the main executable portion of a web page and JavaScript code can access browser states through the DOM.

One potential opportunity for performance improvements in Peregrine is to run JavaScript code in separate mini pages to parallelize the execution of JavaScript. However, JavaScript is a sequential language with global variables and namespaces, so parallelizing JavaScript correctly would require maintaining consistent global JavaScript states across different mini pages and enforcing ordering constraints on dependent snippets of JavaScript code to preserve correct sequential semantics.

Our goal is to support legacy web apps and current web development practices without adding complexity to the Peregrine browser.

**Solution: Run all JavaScript in a single mini page.** Our solution to the first challenge is for our decomposition algorithm to place all JavaScript in a single *main mini page*. Although running all JavaScript in the main mini page prevents us from running JavaScript code in parallel, it maintains all JavaScript variables and namespaces in the main mini page and executes JavaScript code sequentially, making it straightforward to maintain current JavaScript semantics.

In addition to our automatic algorithm, Peregrine does give web developers APIs for decomposing web pages manually, which one could use to parallelize JavaScript. If a developer can identify independent JavaScript code, he or she can place it in its own mini page and the Peregrine browser will execute this JavaScript code in parallel. However, the Peregrine browser will execute this JavaScript code without maintaining JavaScript namespaces or states across mini pages, potentially limiting the types of JavaScript code a web developer might run in parallel.

However, even if we do not support executing Javascript code in parallel, we gain significant benefits when Javascript accesses a subset of the DOM. In our approach, Javascript serialization semantics applies only to the mini-page executing Javascript, freeing the rest of the DOM to be processed concurrently. As we shall see in the experimental section, several websites benefit significantly from this isolation.

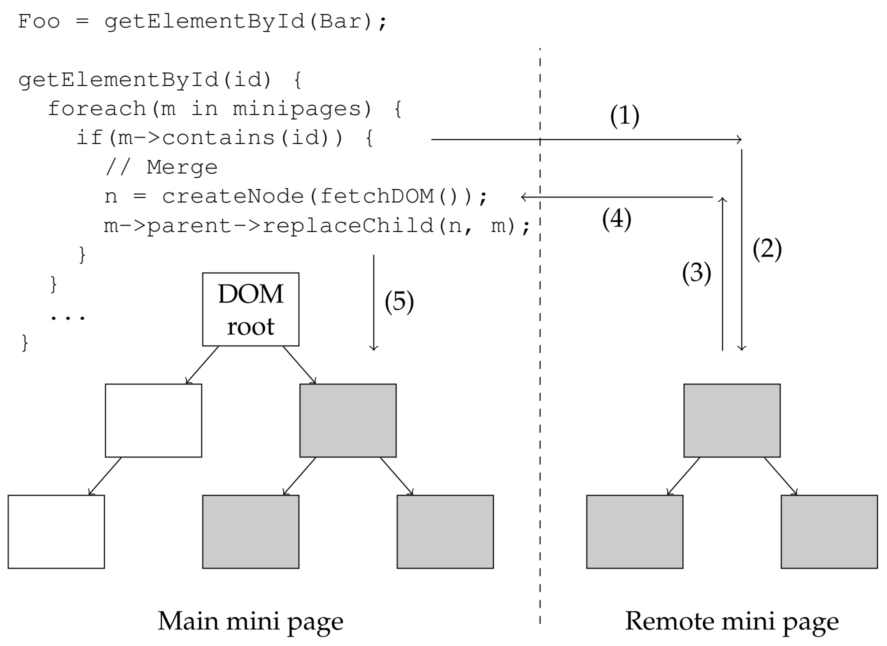
### 3.3.2 DOM consistency

Our next challenge is maintaining a consistent DOM data structure when the data is distributed across multiple mini pages. The DOM data structure represents the state of the browser and it provides APIs for allowing JavaScript to access browser content. The key problem is that we are taking a data structure, the DOM, and distributing it across multiple processes. Conceptually, distributing data structures is straightforward, but pragmatically the DOM data structure is deeply embedded within the browser's implementation.

Our goal is to develop a technique for ensuring DOM consistency that minimizes the changes to the browser.

**Solution: Merging mini pages.** Our solution to this challenge is to merge mini pages when JavaScript accesses remote DOM states (Figure 3.2). When JavaScript accesses remote DOM states the remote mini page serializes its entire DOM and sends the contents back to the requesting JavaScript code. The JavaScript code then inserts this DOM into the DOM of its own mini page, creating a local and consistent view of the DOM. Also, it terminates the remote mini page, leaving the local copy as the sole copy of the newly inserted DOM states. Although this merging operating can be expensive, we implement it using existing DOM API calls and these merging events are rare for most of the web pages we test.

In a traditional sequential web page, the browser builds up the DOM data structure as it parses



```
Foo = getElementById(Bar);

getElementById(id) {
  foreach(m in minipages) {
    if(m->contains(id)) {
      // Merge
      n = createNode(fetchDOM());
      m->parent->replaceChild(n, m);
    }
  }
  ...
}
```

Figure 3.2: Merging mini pages. When JavaScript code accesses a DOM node residing on a remote mini page, Peregrine will (1) issue a request to the remote mini page that will (2) read, (3) serialize, and (4) return the results back to the requesting mini page. Then (5) the mini page inserts the remote DOM into its own DOM tree before terminating the remote mini page.

the HTML of the page. If the parser encounters JavaScript code, it executes this code with the current state of the DOM. After the JavaScript code finishes, the parser continues building up the DOM data structure.

Each mini page builds up its own DOM structure in parallel, so when JavaScript code executes, the Peregrine browser has to ensure that JavaScript accesses the correct DOM state. To ensure that JavaScript accesses the correct DOM state, Peregrine inspects the program counter of the JavaScript code at the base of the current call stack. If the JavaScript code is earlier in the page than the DOM of a mini page, the Peregrine browser skips the mini page when searching for DOM elements because from the JavaScript code's perspective that later portion of the DOM does not yet exist. If the JavaScript code is later in the page than the DOM of a mini page, the Peregrine browser blocks the JavaScript request until the mini page finishes forming its DOM. The decomposition algorithm slices pages to ensure that JavaScript never needs partially-formed DOM states from remote mini pages.

### 3.3.3 Decomposing web pages into mini pages

Our next challenge is determining how to decompose a web page for fast performance on the client. The three key factors that our decomposition algorithm strives to optimize are: maintaining the visual semantics, load balancing for mini pages, and reducing merging. A correct decomposition needs to respect the visual semantics of the original page. Perfectly load balanced mini pages will take the same amount of time to run on the client, thus minimizing our parallel execution segment. Merging can be expensive and it reduces parallelism, so Peregrine strives to avoid merging by identifying independent mini pages.

Our goal is to develop an algorithm that enables us to load balance loosely coupled mini pages.

**Visual cues and feedback from browsers** Our first solution to this challenge is to use visual cues for hints about where to slice a web page. Our intuition is that web developers will divide up web pages into separate visual containers that are mostly independent, and separating them out still preserves the visual semantics. To exploit this design pattern the algorithm decomposes the page based on the minimum bounding box of visual elements on the page. These bounding boxes must *not* overlap and must respect the layout of the original page.

Peregrine also uses visual cues for load balancing mini pages. Currently Peregrine uses the size of the screen area that a mini page occupies to approximate the processing time, thus the algorithm tries to create mini pages that share roughly equal amount of screen.

Our second solution to this challenge is to correct for decomposition that results in slow performance by enabling the Peregrine browser to provide feedback to the server. Specifically, if the Peregrine browser detects merging for a mini page it will notify the server about this merging on subsequent requests to the same page. The Peregrine decomposition algorithm will prevent the merged mini page from forming a separate mini page in future invocations of the algorithm.

For more details on our Peregrine decomposition algorithm see Section 3.5.

### 3.3.4 Pre-computing results on the server

In our final challenge we look at ways that the server-side software can help reduce the amount of computation on the client. There are two key opportunities to reduce computations. First, we can rethink fundamental browser algorithms. Because Peregrine spreads the DOM across multiple mini pages, there is an opportunity to simplify algorithms that enumerate through all DOM elements. Second, the server can provide information for the Peregrine browser to help it avoid costly operations induced by our architecture.

Our goal is to use server computation to help speed up performance of mini pages on the client.

**Solution: CSS, mini page requests, and getElementById.** Our first opportunity for reducing computation is with the CSS selector matching algorithm. CSS rules are a mechanism that enables web developers to separate the display attributes of a web page from the web page itself. For example, a web developer can use the CSS rule `h1 em { color: red }` to specify that any emphasized text (em) within a heading (h1) should be red. To implement CSS rules browsers enumerate through all of the DOM nodes to find selector matches to apply the appropriate rules.

In Peregrine we improve the performance of the CSS selector matching algorithm. Because mini pages have fewer DOM nodes than unmodified web pages, CSS selector matching will have a smaller set of nodes to process and this processing happens in parallel. Furthermore, not all CSS rules apply to all mini pages. For example, if a mini page did not have any heading elements, then it could skip a rule for setting the color of emphasized text within a heading. As a result, Peregrine passes to each mini page only the subset of CSS rules relevant to the individual mini page. The net effects of our changes are that the semantics of CSS are preserved, but this combination of fewer DOM nodes, fewer CSS rules, and parallel processing provides opportunities for performance improvements on the client.

Layout and rendering use non-linear, flow-based algorithms which can be sped up by the decomposition as well. Because the algorithms are non-linear, the decomposition essentially help to "divide and conqueror" the work, reducing the total amount of work has to be done.

In addition, Peregrine passes information to the client to help overcome some of the latency added by operations inherent in the use of mini pages. First, Peregrine includes URLs for each of the mini pages in the header of the main HTML document that the server returns. This placement allows the Peregrine browser to issue mini pages requests without parsing and downloading the entire main HTML document. Second, Peregrine uses a Bloom filter [13] to improve the speed of calls to the DOM `getElementById` function. JavaScript code can use the `getElementById` function to get a pointer to a specific DOM element and the `getElementById` function must check each mini page for the requested element. To avoid this remote procedure call, the Peregrine browser uses a Bloom filter to check quickly if the requested ID is contained within a mini page, avoiding cross mini-page communication.

## 3.4 Peregrine server and browser

In this section we describe our server architecture and our Peregrine browser implementation.

### 3.4.1 Server architecture

Overall we envision two different deployment strategies for the Peregrine server components: a proxy based solution and a web developer guided solution. The web proxy approach interposes processing on the communication between web servers and the Peregrine browser. The proxy fetches raw web contents from web sites, decomposes the resulting web page on the fly, and presents the results to the browser. For web sites with mostly static content, like `wikipedia.org` and `nytimes.com`, the web proxy can cache the translated web site to reduce latency and load for subsequent requests of the same page. However, dynamic content, such as from `facebook.com` or `gmail.com`, has to be reprocessed on each new request.

The primary advantage of proxy-based architecture is that Peregrine can process arbitrary web sites without any assistance from the developer. In fact, this basic strategy mirrors closely the server-side architecture for other mobile browsers, like Opera mini and Skyfire [95, 109], that render content on a proxy before passing data to the client. The primary disadvantage of this proxy-based architecture is that the decomposition algorithm adds latency directly to the web page.

In the second deployment strategy, Peregrine requires web developers to decompose the page. They either run the page through the Peregrine decomposition algorithm, or match the boundaries of sub pages when the page is constructed from server-side sub pages [22, 59, 61]. The server processes the web page as any other web pages following the integration of the decomposition into the server-side code. The primary advantage of this approach is that the resulting modifications add essentially no overhead to the server – the web server processes this modified page like any other web page. The primary disadvantage is that the developer has to modify the web page manually. Modern pages are very dynamic, mixing HTML with server-side lanugages, such as PHP. As a result, the full structure of the page is unknown until run time, making it difficult for the Peregrine decomposition algorithm to reason about the page layout offline.

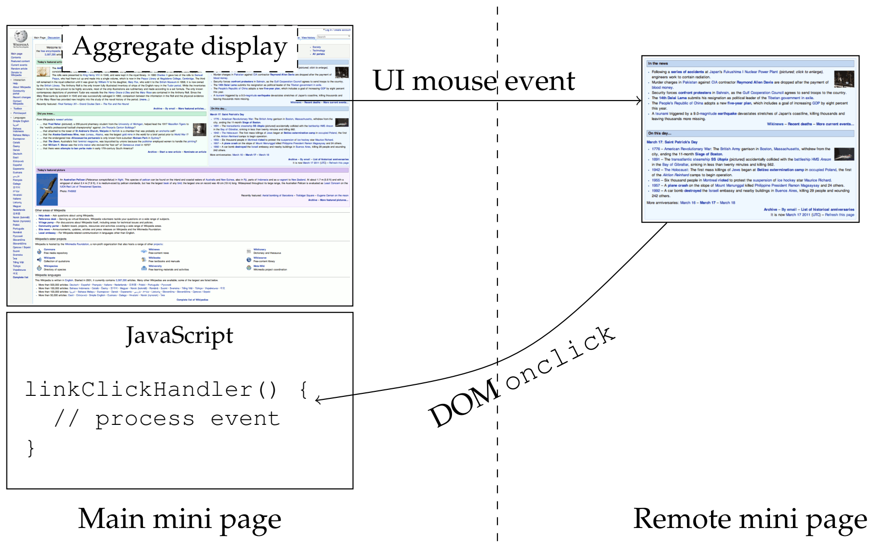
### 3.4.2 The Peregrine browser

Our overall goal with our browser implementation is to make it easy to deploy to other browsers. Although we do have to make some changes to the browser itself, we strive to minimize these changes and to work within the architecture of current browser implementations. The current implementation is built on top of WebKit and the V8 JavaScript engine, which is almost identical to the Android browser except that it uses a different UI toolkit (Qt) to implement the platform specific portions of the browser.

To simplify the integration of mini pages we implement mini pages as browser plugins. In browsers, plugins are applications used to render non-HTML content in a web page, like Flash. In Peregrine, we use a *main mini page* that hosts the display for the other mini pages and embeds the mini pages as plugins using `<embed>` tags. These mini page plugins, however, *do* render HTML content (they are browsers after all), but they are integrated into main mini page through existing mechanisms. We did add IPC channels to enable the mini pages to communicate, but because we use plugins and existing HTML renderers for our mini pages this basic technique should be applicable to other browsers with only minor modifications.

Our first implementation used the XEmbed protocol [38] to assemble mini pages into a single display. However, we ran into scalability problems with our X server that resulted from having up to four mini pages access the X server at the same time. Our current implementation renders mini page content to a bitmap and pass this bitmap to the main mini page using shared memory. The main mini page then displays this bitmap. This display technique is similar to how Chrome separates rendered HTML content from the rest of the Chrome UI [48], which runs in a separate process.

The main mini page recieves UI events from the window manager. If the destination of the event is a mini page, the main mini page will forward the UI event to the appropriate mini page.



```
linkClickHandler() {
  // process event
}
```

Figure 3.3: Event routing. This figure shows how Peregrine handles a mouse click on a link.

The mini page will then process the event. Some UI events cause DOM events (e.g., `onclick`), and programmers can add JavaScript event handlers that run in response to these events. If a mini page detects a DOM event that has a registered listener, the mini page will route this event to the main mini page where the event handler JavaScript code resides. Figure 3.3 shows an example of our event routing mechanisms. If a user clicks on a link in a mini page, the main mini page will send the UI mouse event to the mini page. After the mini page processes the UI mouse event and determines that the click was on a link, the mini page then forwards the `onclick` DOM event for that link back to the main mini page, which runs the appropriate event handlers.

We modified for Peregrine browser to support merging mini pages, event delivery, and UI rendering. We modified the DOM event-delivery mechanism to ensure that DOM events originating in mini pages are routed to the main mini page. In addition we implemented support for displaying bitmaps created by mini pages and routing UI events to mini pages. To handle mini-page merging, we modified the Node methods `firstChild`, `lastChild`, `previousSibling`, and `nextSibling`; the Element methods `firstElementChild`, `lastElementChild`, `previous-ElementSibling`, and `nextElementSibling`; and the DOM API calls `getElementById` and `getElementsByTagName`. Finally, we hook the bindings between the browser and the V8 JavaScript engine.

| Symbol | Description |
|---|---|
| $s = (e, p, q)$ | A segment which contains all nodes between $p-$th and $q-$th children of the node $e$. |
| $p_i$ | The $i-$th mini page, which is a segment. |
| $p'$ | The main mini page. |
| $W(e)$ | Weight of the node $e$ |
| $R(e)$ | The sum of weight for all descendants of $e$ plus the weight of $e$. |
| $r(s)$ | The rank of a segment $s$. It is the sum of $W$ for all nodes in $s$. |
| $bb(s)$ | Bounding box of all visible elements in $s$, i.e., the minimum rectangles fully covers all visible elements in $s$. |
| $sticky(s)$ | true if there exists a sticky node which is a descendant of one of the nodes in $s$. |
| $S$ | Sets of mini pages. |
| $C(S)$ | The cost function of the decomposition. |

Table 3.2: Notation used in Section 3.5.

## 3.5 Page decomposition algorithm

This section describes the decomposition algorithm Peregrine uses to slice mini pages out of a web page. Table 3.2 lists the symbols we use when describing our algorithm.

The goal of the decomposition algorithm is to divides the DOM tree into $k$ mini pages $S = (p', p_1, p_2, \cdots, p_{k-1})$. The algorithm slices $k-1$ mini pages $(p_1, \cdots, p_{k-1})$ out of the DOM tree and puts the remaining DOM elements into the main mini page $p'$.

The algorithm considers four constraints when decomposing pages:

- Consistent state of the DOM. The decomposition preserves the order of nodes (i.e., siblings and parents) in the DOM trees, so that the DOM of the mini pages can be directly plugged back to the main page.

- Respect the layout of both the main mini page and mini pages. Peregrine uses browser plug-ins to implement mini pages, which requires (1) the shape of mini pages to be rectangular, and (2) there should be no overlap between different mini pages and the main page. Otherwise the layout of the main page will potentially be different.

- Place sticky nodes into the main page. A sticky node is a DOM node that should be placed in the main page unconditionally. For example, all `<script>` nodes are sticky nodes, so that none of the nodes in mini pages contain `<script>` tags.

- Balance the load between the $k$ parts and minimize synchronization (i.e., merging).

We require a mini page to be a segment, therefore the first constraint is implicitly satisfied. The following formulas models the second and the third constraints:

$$\begin{cases} bb(p_i) \cap bb(p_j) = \Phi & \forall 0 < i < j < k \\ sticky(p_i) = false & \forall 0 < i < k \end{cases}$$

And we model the forth constraint as the object function $C(S)$:

$$C(S) = \min\{\max_{s \in S}\{r(s)\}\}$$

Figure 3.4 sketches a worklist algorithm to compute the decomposition $S$. Intuitively, it keeps dividing the page with respect to the above constraints, and extracts the segments with appropriate ranks (parameterized against $\alpha$ and $\beta$) as mini pages. The line *overlapped(s)* tests whether the non-overlapping constraint is satisfied if $s$ is added into the result $S$, and *adjust(s)* further decomposes $s$ in order to satisfy the non-overlapping constraint. Subroutine *divide($s, r$)* decomposes the segment $s$ into smaller segments whose ranks are at most $r$. Subroutine *extract(s)* extracts the segment $s$ into a mini page.

For both *adjust(s)* and *divide($s, r$)*, they decompose the grandchildren of $s$ if there is only one node in the segment. We omit the details for these helper functions for simplicity.

As a heuristic, Peregrine sets the rank of a node $e$ to be the normalized size of the screen that it occupied, and computes the weight of a node $w(e)$ by subtracting the rank of all its children.

We implement *adjust()* and *overlapped()* with the HitTest framework in the browser. The HitTest framework enables Peregrine to query the set of elements inside a rectangle. Peregrine calculates the bounding box and queries whether there exists any elements in the main page or in mini pages to make sure the segment meets the non-overlapping constraint.

```
decompose($root$, $k$)
$q \leftarrow$ Maximum priority queue of segments with respect to their ranks.
Add a shadow node $root'$ which contains only one child $root$
Enqueue($q$, $(root', 0, 1)$)
$t \leftarrow r(root)/k$
while $k > 1$ and $q$ is not empty do
  $s \leftarrow$ pop_front($q$)
  if overlapped($s$) then
    $\mathcal{S} \leftarrow$ adjust($s$)
    Enqueue all segments in $\mathcal{S}$ into $q$
    continue
  end if
  if $r(s) < \beta t$ then
    break
  else if $r(s) > \alpha t$ or $sticky(s)$ then
    $\mathcal{S} \leftarrow$divide($s$, $\alpha t$)
    Enqueue all segments in $\mathcal{S}$ into $q$
  else
    $p_{k-1} \leftarrow$ extract($s$)
    $k \leftarrow k - 1$
  end if
end while
Put everything left into $p_0$
```

Figure 3.4: Heuristic algorithm to decompose web pages.

## 3.6 Evaluation

This section presents a detailed evaluation of Peregrine's server and browser components. We first describe our experimental setup. Then we examine Peregrine's performance and compare it with that of an unmodified WebKit-based browser. Finally, we discuss the sources of Peregrine's overheads.

### 3.6.1 Experimental setup

In our experiments, Peregrine browser runs on a CoreTile Express A9x4 ARM development board. The board has a quad-core Cortex-A9 CPU running at $400$Mhz and $768$MB of DDR2 RAM. Peregrine server runs on a server-grade machine with 12GB of DDR3 RAM and two Intel Xeon X5550 CPUs running at $2.67$GHz. They are connected via a FastEthernet switch. We install Ubuntu 11.04 ARMv7 on the board and Ubuntu 11.04 x86-64 on the server. We compile Peregrine against Qt

4.7.2 and WebKit r74353 on both systems.

We choose 170 of the 250 most popular web sites according to Alexa [2] as a benchmark. We select sites with different characteristics. For example, some sites use JavaScript heavily — `www.google.com` —, while others have large DOM trees but use little JavaScript — `en.wikipedia.org/wiki/Nokia` —. To further increase the diversity of the benchmark, we choose only one representative from a equivalence class. For example, `google.com`, `google.co.in` and `google.ca` are all in the Top Alexa 250, but only `google.com` is part of our benchmark suite. In addition to the landing pages listed in Alexa, the benchmark also includes the pages that provide the main functionality of the web site. For example, we include result pages from a Google search, a news story from the NYT, etc.

We use a QtWebKit-based browser as our baseline. It is a minimal web browser linked against unmodified WebKit r74353. It is the same browser we mentioned in Section 3.2, and we will refer to it as QtBrowser in this section.

We evaluate Peregrine browser and the QtBrowser by measuring the time they take to load web pages. Previous work uses visual cues, such as the readiness, or the time to interact with certain pivot elements to measure the performance [61, 91]. We choose page loading time as a metric because it is objective and it does not depend on the element chosen as pivot.

Unless explicitly stated otherwise, we mirror the web pages from the Internet and serve them on the machine running Peregrine server with Squid 3.1.11 and nginx 1.0.4. Peregrine browser uses all four cores on the board. We repeat each measurement 10 times and report their average.

### 3.6.2 Overall performance of Peregrine

Figure 3.5 describes the distribution of the differences of page load latency time between Peregrine browser and QtBrowser. We use two different Peregrine server configurations: *cached* and *uncached*.

When the server caches the results of page decomposition, Peregrine reduces the page load latency by 1.75s on average — or a 1.54x overall speed up across the benchmark. Peregrine speeds up 151 out of 170 (89%) sites. Moreover, 39 (23%) sites improve by two seconds or more. In the best case, Peregrine reduces the page load latency by 14.9s. In the worst case, it increases the page load

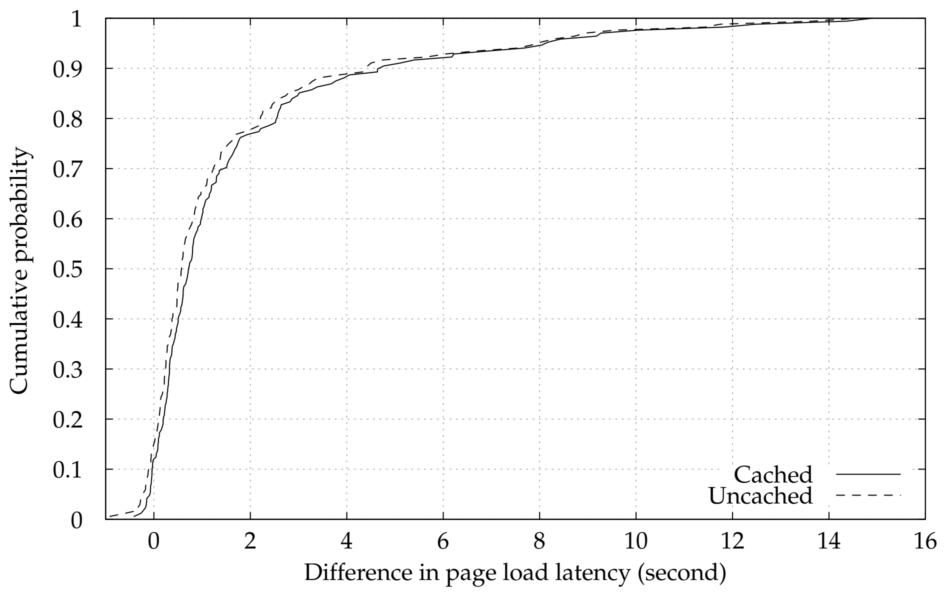


Figure 3.5: Cumulative distribution function (CDF) for the differences of page load latency time between Peregrine browser and QtBrowser. Positive values mean Peregrine browser is faster than QtBrowser. *Cached* means that Peregrine server cached the result of page decomposition and served it directly. *Uncached* means Peregrine server decomposed the page on the fly.

latency by 269ms. Notice that Peregrine server does not need to split a page if it is not profitable. That is not the case in this evaluation, and in our experiments we pay the price of non-profitable splits.

In the uncached scenario, Peregrine server has to decompose the page for each request, adding 184ms on average to the page load latency. The decomposition algorithm is efficient, and it only takes 13ms on average to decompose a web page from our benchmark.

In a real-world deployment, only the first client would suffer the extra 184ms. We believe it is a small price to pay for the performace improvement that the subsequent clients would experience.

### 3.6.3 Experiences with Peregrine

To characterize Peregrine, we study the performance of three representative cases in detail. We instrument the execution of both Peregrine browser and QtBrowser with OProfile to collect run-time statistics.

Each case study contains two graphs: a *timeline graph* and a *workload graph*. The timeline graph plots the total page loading time for QtBrowser and Peregrine. The top-most bar represents the total page loading time of QtBrowser (Full). The gray bars below represent the page loading time of the main mini page (a.k.a, MP) and Mini Pages (MP1$\sim$ MP3). Thus, the total page loading time for Peregrine browser is determined by the longest gray bar. For comparison, we load each mini page with QtBrowser and report execution time with the corresponding white bars.

The workload graph classifies the workload of the two browsers into six disjoint categories: (1) the V8 JavaScript engine (V8), (2) The Linux kernel (Kernel), (3) Qt Painting and rendering (Painting), (4) CSS Selection (CSS), (5) WebKit except for CSS Selection (WK-CSS), and (6) libc and other components in Qt (libc+Qt). These six categories consume most of the CPU time. The execution time of each of these six components is normalized with respect to the total time spent by QtBrowser to load the original page. Notice that, the workload graph stacks the execution of all Peregrine processes into one bar.

**Case study I: Wiki-Nokia** . Figure 3.6 shows the performance characterization of the Wikipedia entry for Nokia (`http://en.wikipedia.org/wiki/Nokia`). Peregrine reduces the page loading time by 11.7s. The timeline graph shows that Peregrine server split the page into 3 mini pages and Peregrine browser was able to render them in parallel. This is because the Nokia entry in Wikipedia is long, and there is enough independent work for each mini page.

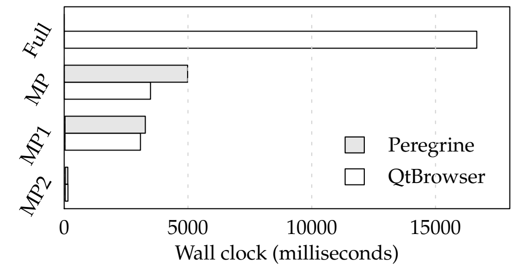
However, simply splitting the page does not explain why Peregrine is so much faster than QtBrowser— if that were the case, the total time for the white rectangles in all mini pages would roughly be equal to the top-most bar. The workload graph shows that there is almost a 3x reduction for both CSS and WK-CSS. For CSS, the decomposition brings in two benefits: (1) Peregrine server speeds up CSS for MP1 and MP2 through inlining CSS rules. (2) CSS selection runs on fewer elements in the main page (30% of the original page), reducing the total amount of work.

For WK-CSS, analysis reveals that the execution time reduction can be mainly attributed to layout and rendering. The decomposition enables the main page to treat both MP1 and MP2 as "black-boxes" during layout and rendering. The main page is no longer responsible to render elements inside mini pages as mini pages running in other processes are rendering them. For layout, the main page obviously has fewer elements to lay out, as it only needs to lay out the elements staying in the main pages and the black boxes containing the mini pages. Relayout is

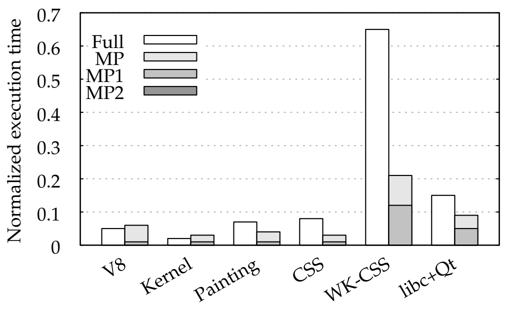
**Case I: Wiki-Nokia**

QtBrowser / Peregrine: 16.7s / 4.99s

*Timeline graph*



*Workload Graph*



*Summary*

Why faster?

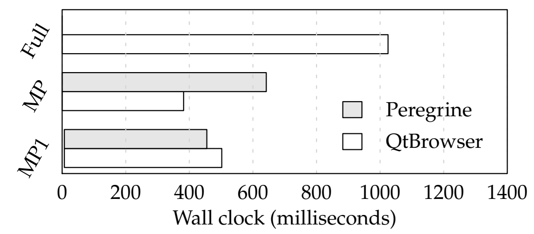
+ Simplified CSS, layout and rendering
+ Layout and rendering executed in parallel

Figure 3.6: Performance analysis for `http://en.wikipedia.org/wiki/Nokia`.

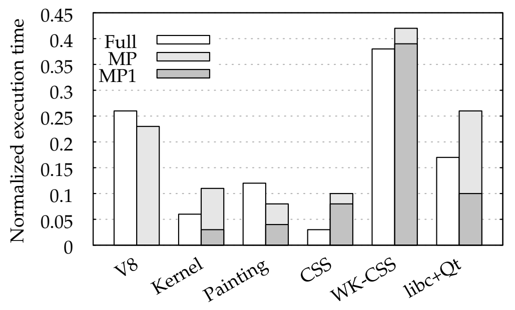
**Case II: http://sfbay.craigslist.org**

QtBrowser / Peregrine 1.03s / 0.64s

*Timeline graph*



*Workload Graph*



*Summary*

Why faster?

+ Layout, rendering and JavaScript execute in different processes that run in parallel.

Figure 3.7: Performance analysis for `http://sfbay.craigslist.org`.

also (which could happen when WebKit determines the size of an image) simplified due to the same reason.

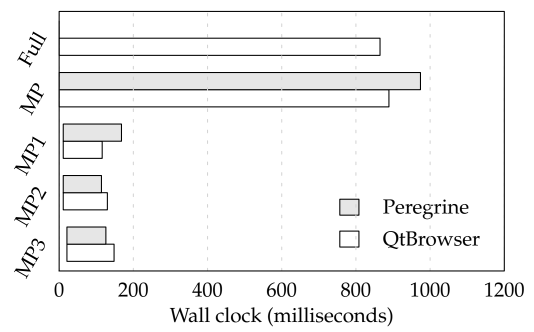
**Case study II: `http://sfbay.craigslist.org`** . This is the landing page of `craigslist.org`, which is built mostly upon a number of tables and hyperlinks. Peregrine speeds up the loading of the page by 1.61x (Figure 3.7). The workload graph shows that all the mini pages combined do more work than QtBrowser, but because they do it in parallel, Peregrine is still faster than QtBrowser. Notice that the bars for the MP1 do not start at time 0. This is because the main mini page does some work before it spawns it.

More specifically, Peregrine decomposes the page into two mini pages: (1) the main mini page

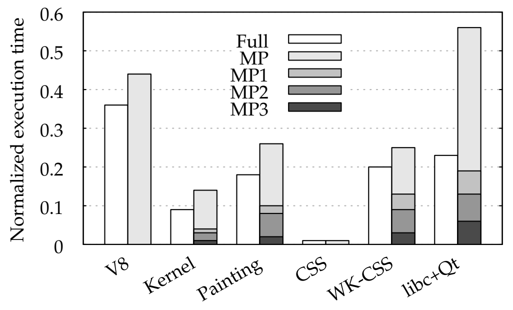
**Case III: bj.58.com**

QtBrowser / Peregrine: 0.86s / 0.97s

*Timeline graph*



*Workload Graph*



*Summary*

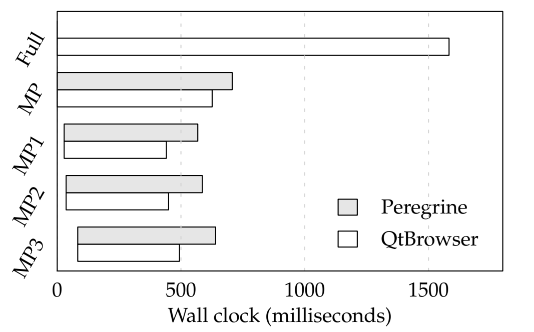
Why slower?
- Limited the granularity of parallelism.
- Excessive fine-grained parallelization imposes significant overheads.

Figure 3.8: Performance analysis for `http://bj.58.com`.

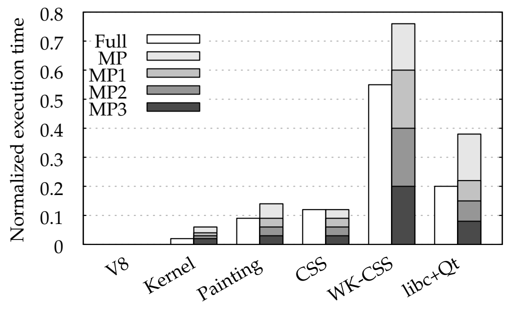
**Case IV: php.net (Perfectly balanced)**

QtBrowser / Peregrine: 1.58s / 0.70s

*Timeline graph*



*Workload Graph*



*Summary*

Why faster?
+ Parallel execution
+ Simplified layout and rendering
+ Smaller working set

Figure 3.9: Performance analysis for an artificial perfectly load balanced web page.

taking over all JavaScript and with a few elements, and (2) MP1, which takes the majority of elements in the original page. Therefore, most of the CSS selection, layout, and rendering happens inside MP1. Analyzing the source code of the page shows that this decomposition matches the intended modularity of the page: the JavaScript implements the user interface for the rightmost column of the page. There're a few sections in the column, where the inactive section is hidden before showing the active one that the user have clicked on.

The user interface is implemented via jQuery [114], which takes a non-negligible amount of CPU time to parse and execute. By decomposing the page into two pages, Peregrine is able to execute JavaScript and to render the page on the screen concurrently, leading to performance improvement.

In this case, the performance improvement depends on the correct decomposition made by Peregrine server. The current implementation of the server divides the page in this way without any hints, although a redesign of the page's UI might require hints telling the server to stick the rightmost column into the main page.

**Case study III: `http://bj.58.com` .** The Peregrine server successfully decomposes the page into four pages but the page load latency for this page increases by 101ms (Figure 3.8). The timeline and workload graphs imply that the load of mini pages is imbalanced, and that the additional overheads from parallelization hurts performance.

Analysis shows that the page contains many elements that can be pulled out into mini pages, but the guarantees of visual correctness and DOM consistency from Peregrine keep these elements to be extracted as consecutive thunks. Therefore, mini pages contain too little work to overcome the performance penalties for parallelization. There is a `<div>` showing the message "loading", which is hidden by JavaScript right before the `onLoad` event is fired. Unfortunately, the `<div>` resides in the giant `<div>` which contains most of the contents of the page. The Peregrine server sticks the `<div>` in the main mini page to avoid merging, thus Peregrine is forced to divide the sub DOM tree of giant `<div>` into much smaller thunks to preserve DOM consistency.

We believe the problem could be easily fixable by developers: moving the `<div>` right under `<body>`, and fixing its position via additional CSS rules would solve the performance problem and retain the same functionality of the page. To test this hypothesis, we implemented the proposed changes in our mirrored copy. After our changes, the page loaded in 743ms, 15% faster than QtBrowser.

However, the workload graph shows that there are cases where Peregrine might cause non-negligible overhead. We discuss this in the next subsection.

### Overheads for parallelization

To further understand the overheads of Peregrine, we derived a perfectly load balanced test case from the contents of `http://www.php.net`. We (i) got rid of all JavaScript in the page, (ii) cloned its contents four times, (iii) put it all together in a single page and (iv) hard-coded the decomposition in Peregrine server to decompose the big pages into identical mini pages.

Figure 3.9 describes the overall performance for this test case. Surprisingly, Peregrine browser is only 2.25x faster. Although the main Page and mini pages have exactly one fourth of the contents of the original page, the execution time of each ranges from $35\%$ to $45\%$ of the time spent by QtBrowser.

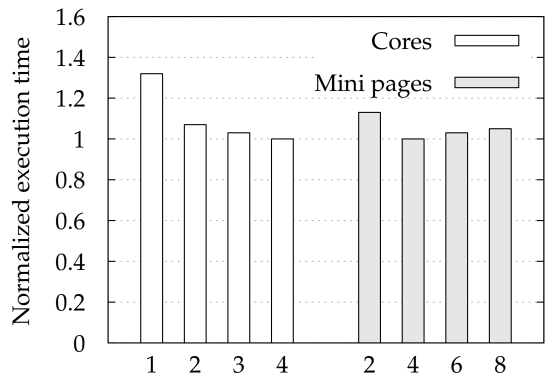
Analysis shows that there are two types of overheads. First, there is there's *less caching* from WebKit and other libraries. For example, WebKit only decodes the same image for a single time even when there are multiple uses of it in the page. Also, the FreeType library caches the glyphs of characters. As an example, if there are two words "foo" in a page, the FreeType library only calculates the glyphs for once. The current implementation of Peregrine uses multiple processes to render the page, and these types of caching are process-specific, thus they are recomputed for each rendering process. Most of the overheads in the category Painting and WK-CSS fall in this category.

The second type of overheads come from our Peregrine's implementation. According to the workload graph, there are some overheads coming from the OS kernel for book keeping and context switching. There are also overheads caused by mini-page integration: Peregrine browsers propagates all events painting and input events from MP $1/2/3$ into the process running the main mini page via inter-process communication (IPC). Extra care has to be taken so that IPC doesn't block the event loop of the process. Peregrine has dedicated threads for IPC, and events are posted asynchronously into the main event loop. The additional overheads in libc+Qt for the main mini page fall in this category.

We believe that some of these overheads could be amended by more mature implementation, such as implementing better caching and more efficient IPC mechanism.

**Load imbalance**

As we have shown before, it is sometimes difficult to properly balance work among mini pages due to correctness guarantees. A common strategy is to over-decompose, i.e., to have more mini pages than the number of available cores. Peregrine still uses one rendering processes for each



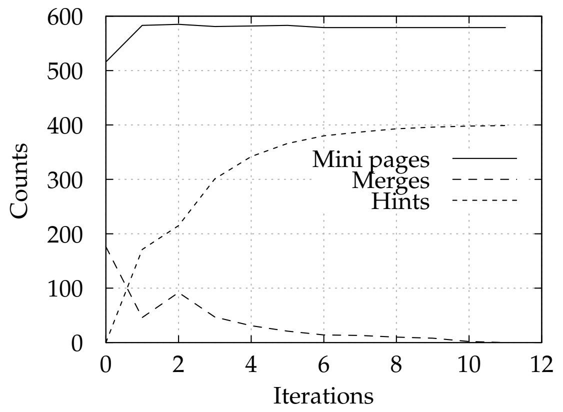


Figure 3.10: Performance impact for different number of cores and mini pages.

Figure 3.11: Effectiveness of feedback loop to avoid excessive merging.

mini page, but the OS scheduler is is in charge of balancing their execution.

Figure 3.10 describes the performance of Peregrine when different number of cores and mini pages are available. The execution time is normalized to the run that both the maximum number of mini pages and available number of cores are both four. The white bars show the performance of Peregrine browser when splitting a page into four mini pages. The shaded bars show the performance of Peregrine browser when all four cores are available, but maximum number of mini pages created by Peregrine server varies. Although in some cases picking different number of mini pages could improve the performance, the graph shows that it is wise to pick the maximum number of mini pages in Peregrine server to be the maximum number of available cores.

**Merging**

To evaluate how well a feedback mechanism can avoid merging, we conducted the following experiment. Peregrine server begins execution with empty hints, and the Peregrine browser loads the page and records the element that causes merging. The information is fed into the server at the end of each run so that it avoids pulling the element out of the main page in the future. We iterate the experiment until the feedback loop is stabilized.

Figure 3.11 shows that the feedback loop is effective: starting with 176 merging operations for 170 web pages, the feedback loop successfully eliminates all merging after just 11 iterations, ending up with 399 hints fed into the server in total. It takes 2.55 runs on average to eliminate all merging for all sites in the benchmark.

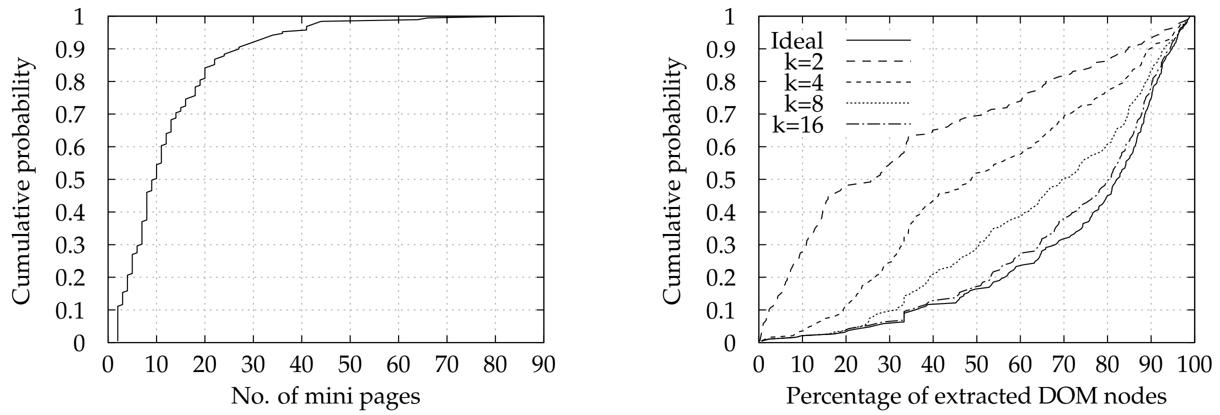


Figure 3.12: Cumulative distribution function for the maximum number of extractable minipages and the percentages of extractable DOM elements.

### 3.6.4 Availability of data parallelism

To evaluate how much potential data parallelism is available, we modified the algorithm described in Section 3.5 to understand two key parameters: how many mini pages are needed to extract all parallelism, and how much parallelism can be extracted with a fixed number of mini pages. We estimate the availability of parallelism with the number of extractable DOM elements, that is, elements could be put into mini pages without triggering merging given the hints available. The algorithm uses the same hints that are used in the previous experiment.

Figure 3.12 presents the CDF of the number of mini pages required to extract all parallelism, and the percentages of extractable DOM elements with different numbers of mini pages. These figures show that there might be abundant data parallelism that can be extracted in practice: First, many DOM elements could be placed into mini pages. When Peregrine uses unlimited numbers to exploit parallelism (Ideal), more than $50\%$ of elements are extractable for about $85\%$ of the web sites, and more than $80\%$ of DOM elements are extractable for half of the sites. Second, Peregrine browser only needs a relatively small number of mini pages to extract all available parallelism. $16$ mini pages can drain up all parallelism for more than $70\%$ of web sites, and they can extract most of available parallelism for all sites.

We believe that Peregrine has great potential for speeding up web browsing as more cores become available on mobile devices in near future.

## 3.7 Discussion

*Compatibility with today's web.* We did not observe any visual differences in Peregrine browser for all web sites in our benchmark. We believe the design of Peregrine can maintain the semantics of both DOM and JavaScript semantic, but we anticipate the current implementation of Peregrine would fail in some cases. For example, the intermediate results of layout could be different, although the page will look exactly the same after loading is finished. This is because elements in mini pages participate layout differently. Peregrine browser lays out a mini page as a black box, where the actual layout of all elements in the mini page is delegated to another process. Other browsers, however, iterate the layout of these elements when more information are available. For example, these elements could be pushed to the lower part of the screen because the size of an image before them are available. As a result, users could potentially find they could interact with the bottom of the web page but the top half of the page hasn't correctly laid out yet. It might be confusing even the page shortly becomes correct. Developers could quickly fix this problem by specifying more information in the original page, which is a widely adopted optimization to avoid excessive relayout during page loading today.

Some compatibility issues could be attributed to immaturity of the implementation. First, the current implementation only handles CSS inheritance partially, which might lead to incorrect layout and rendering. Second, WebKit fails to report the portion of the screen occupied by an element in some cases, confusing the page decomposition algorithm. Third, the current implementation fails to relayout a mini page in some subtle cases. We believe all these problems can be addressed with more mature implementation, although in the worst case either the Peregrine browser could merge all mini pages together, or the Peregrine server could bypass the decomposition to work around compatibility issues.

*Server scalability.* Peregrine server runs WebKit to parse web pages and compute mini pages. The work load is reduced because the server avoids rendering and JavaScript execution. Experimental results show that Peregrine servers spends an average of 184ms for mini page decomposition, indicating that it is feasible for modern web servers. Moreover, some commercial web proxies, such as Opera mini and Skyfire, do full web page rendering, suggesting that pushing this computation to the server can be practical.

*Impacts of network latency.* Huang *et al.* [57] suggests that mobile browsers are also bounded by the long the round-trip time of the today's 3G network. To emulate the network effects in practice, we used dummynet [19] to inject 20 and 150 milliseconds round-trip latency between the board and the server in our experimental environment. We recorded 1.35x and 1.24x speed ups on average across the benchmark.

The Peregrine browser requests web pages through Peregrine server. The extra indirection opens up an opportunity to deploy new networking protocols like SPDY [49] transparently. By replacing the protocols between Peregrine browser and Peregrine server, Peregrine can benefit from new protocols and maintain the compatibility of today's web.

## 3.8 Related work

The most closely related work to Peregrine is a recent proposal by Meyerovich and Bodík [90], where they propose parallel layout algorithms for web browsers. Fortuna *et al.* suggests that reasonable amount of JavaScript workload could potentially be parallelizable [45]. As we discussed in Section 3.2, the diversity of web browsing's workload and Amdahl's Law limit the potential benefits from exploiting task-level parallelism. Meyerovich and Bodík evaluated their algorithm outside a browser and reported as much as 80x speed ups. However, when Badea *et al.* tested a parallel layout algorithm in Firefox using a four-core system, they saw a more modest maximum speedup of 1.6x when measuring page load latency times for the CSS-heavy `zimbra.com` web site [6].

Browsers can push computation to well-provisioned servers to reduce the load on resource-constrained clients. This offloading technique has been applied to browsers running on mobile phones. For example, Opera mini renders web pages in a remote proxy server and then presents a simplified representation to the client, and Flash proxy [92] renders Flash in a server and displays the rendered content in the client using remote display techniques. Although these optimizations improve the performance of the web app, the Opera mini approach deprecates web app functionality, and neither technique exploits the multi-core client systems that will run future mobile platforms.

Several recent techniques from industry focus on speeding up web apps. There are proposals for optimizing JavaScript [50], pipeplining network transfers and computations between web servers and the browser [61, 94], and rethinking network protocols between servers and browsers [49]. These efforts are all complementary to Peregrine and should help improve the performance of each of our individual mini pages. Additionally, pipelining network transfers and faster network protocols should increase the need for speeding up clients by further removing the network from the critical path.

Yahoo! published a list of 35 “best practices” for web developers to help speed up their web apps [125]. These techniques include adding cache-control headers to content to allow browsers to cache requests, and using Gzip to decrease the amount of data the server needs to send the client. These techniques are complementary to Peregrine, our techniques improve the performance of many web sites, including web sites that web developers have optimized.

## 3.9 Summary

This chapter presents Peregrine, a system that improves browser performance by changing the interaction between web servers and browsers. The Peregrine server decomposes legacy web pages into mostly independent pages, and the Peregrine browser renders these pages in parallel with multiple processes. The Peregrine browser integrates these pages to maintain the same visual presentation and functionality of the original web page.

Experimental results on 170 web sites showed that Peregrine is able to parallelize a wide range of workload in web browsing by exploiting data parallelism. Peregrine reduced the latency of loading web pages by 14.9 seconds on a quad-core ARM system. As mobile devices continue to adapt to multi-core architectures, we believe that Peregrine’s approach could provide even greater performance improvement for future mobile systems.

# Chapter 4

# Formally verifying security invariants in operating system kernels

## 4.1 Introduction

Chapter 1 has discussed the strong demands of security and privacy of modern mobile operating systems. To address this problem, current research focuses on reducing the size of TCB using a hypervisor (discussed in Chapter 1), or on formally verifying the operating system kernels. Some notable work includes formalizing UNIX implementations [43, 118] and on verifying microkernel abstractions [58, 69]. Although these projects approach or achieve full functional correctness, they require a large verification effort (the seL4 paper claims that it took 20 man-years to build and prove [69]) and detailed knowledge of low-level theorem-proving expertise.

This chapter describes a novel approach that enables formally verifying security invariants in operating system kernels *without proving full-blown functional correctness*. It describes ExpressOS, an operating system kernel that provides verified security invairants to mobile applications. Our experience show that by combining the power of different abstractions from programming languages, operating system kernels and formal methods, it is possible to provide high assurance security invariants with partial verification. For example, we have verified that the private storage areas for different applications are isolated in ExpressOS. We *do not* derive this security invariant by showing the every aspect of all involved components, like the file systems and the device drivers, is correct. Instead, ExpressOS isolates the above components as untrusted system services. The proofs show that ExpressOS encrypts all private data of applications before sending it out to the system services, and ExpressOS places security checks correctly so that only the application itself can access its private data.

We achieve the proofs of these invariants by annotating source code with formal specifications written using mathematical abstractions of the properties, using code contracts [77] and BOOGIE-

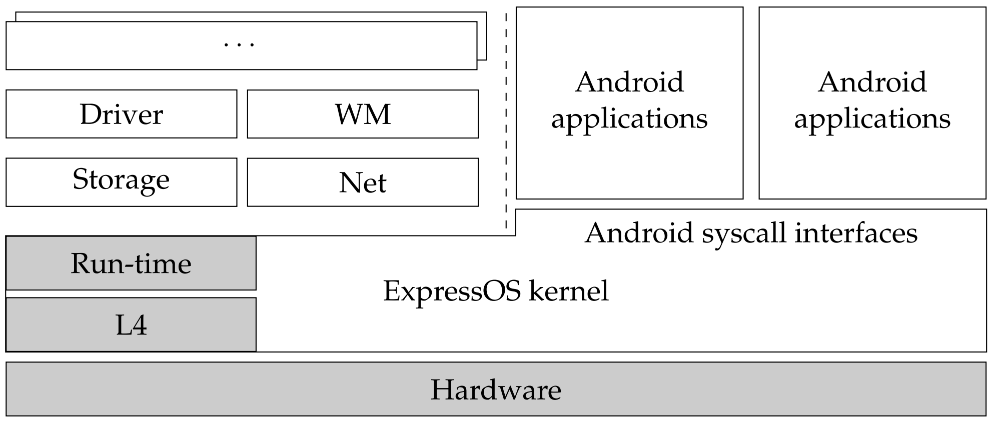


Figure 4.1: Overall architecture of ExpressOS. Android applications run directly on top of the ExpressOS kernel. Boxes that are left to the vertical dotted line represent the system services in ExpressOS. Shaded regions show the trusted computing base (TCB) of the system.

based tools [27, 76], writing ghost-code annotations that track and update these mathematical abstractions according to the code's progress, and by discharging verification using either abstract interpretation or automatic theorem provers (mainly SMT solvers [32]). A thorough verification of the invariants above requires only a modest annotation effort ($\sim 2.8\%$ annotation overhead).

We evaluated the security and the performance of ExpressOS. We have built a prototype system that runs on x86 hardware and exports a subset of the Android/Linux system call interfaces. To evaluate security, we have qualitatively analyzed the security of ExpressOS, then we have examined $383$ recent vulnerabilities listed in CVE [28]. The ExpressOS architecture successfully prevents $364$ of them. To evaluate performance, we have used an ASUS Eee PC and run a web browser on top of ExpressOS. Our experiments show that the performance of ExpressOS is comparable to Android: ExpressOS shows $16\%$ overhead for the web browsing benchmark.

## 4.2 ExpressOS overview

The primary goal of ExpressOS is to be a practical, high assurance operating system. As such, ExpressOS should support diverse hardware and legacy applications. Also, security invariants of ExpressOS should be formally verified.

This section provides an overview to the architecture, the verification approach, and system components of ExpressOS. Section 4.5 discusses the security implications of our architecture and the detailed design of the ExpressOS kernel.

### 4.2.1 Architecture for verification

Figure 4.1 describes the architecture of ExpressOS. The architecture includes the ExpressOS kernel, system services, and abstractions for applications.

ExpressOS uses four main techniques to simplify verification effort. First, ExpressOS pushes functionality into microkernel services, just like traditional microkernels, reducing the amount of code that needs to be verified. Second, it deploys end-to-end mechanisms in the kernel to defend against compromised services. For example, the ExpressOS kernel encrypts all data before sending it to the file system service. Third, ExpressOS relies on programming language type-safety to isolate the control and data flows within the kernel. Fourth, ExpressOS makes minor changes to the IPC system-call interface to expose explicitly IPC security policy data to the kernel. By using these techniques, ExpressOS isolates large components of the kernel while still being able to prove security properties about the abstractions they operate on.

Current techniques to isolate components and manage the complexity of a kernel include software-fault isolation [21, 112, 117], isolating application state through virtualization [3, 29, 120], and microkernel architectures [43, 58, 69, 118]. These techniques alone, however, are insufficient for verifying the implementation of a system's security policies – the correctness of the security policies still relies on the correctness of individual components. For example, an isolated but compromised file system might store private data with world-readable permissions, compromising the confidentiality of the data.

### 4.2.2 Design principles

In order to meet the overall goal of ExpressOS, three design principles guide our design:

- *Provide high-level, compatible, and verifiable abstractions*. ExpressOS should provide high-level, compatible abstractions (e.g., files) rather than low-level abstractions (e.g., disk blocks), so that existing applications can run on top of ExpressOS, and developers can express their security policies through familiar abstractions. More importantly, the security of these abstractions should be formally verifiable to support secure applications.

- *Reuse existing components*. ExpressOS should be able to reuse existing components to reduce the engineering effort to support production environments. Some of the components might have vulnerabilities. ExpressOS isolates these vulnerabilities to ensure they will not affect the security of the full system.

- *Minimize verification effort*. Fully verifying a practical system requires significant effort, thus the verification of ExpressOS focuses *only on security invariants*. Also, this principle enables combining lightweight techniques like code contracts with heavywieght techniques like Dafny annotations [76] to further reduce verification effort.

### 4.2.3 Verification approach

Our modular design limits the scope of verification down to the ExpressOS kernel. The ExpressOS kernel is implemented in C# and Dafny [76], both of which are type-safe languages. Dafny is a research programming language from Microsoft Research that researchers use to teach people about formal methods. Like SPIN [10] and Singularity [58], the language ensures that the ExpressOS kernel is free of memory errors, and provides fine-grain isolation between components within the kernel. A static compiler compiles both C# and Dafny programs to object code for native execution.

We verify security invariants in ExpressOS with both code contracts and Dafny annotations. Code contracts are verified using abstract interpretation techniques, which have low annotation overhead but are unable to reason about complicated properties, like properties about linked lists. In contrast, Dafny annotations are verified using logical constraint solvers, which are capable of handling complicated properties, but require a heavy annotation burden and deep expertise in formal methods to use. Based on their charactertics, we verified simpler security invariants using code contracts, and more complex ones (like manipulation of linked lists) in Dafny, where we use ghost variables (i.e., variables that aid verification) to connect the proofs of both techniques. The combination enables high productivity for verification using code contracts, yet still retaining the the full expressive power of Dafny to verify complicated properties needed to prove security invariants.

We verify security invariants that involve asynchronous execution contexts by reducing them into the object invariants of relevant data structures (Please see Section 4.4 for more about object invariants). Verification in asynchronous execution contexts is challenging due to the separation of the control and data flows. Verifying only the security invariants, however, has a simple solution. In particular, we express security invariants and the object invariants of relevant data structures in terms of ghost variables, and the object invariants imply the original security invariants. The main advantage is that these object invariants can be reasoned about locally; therefore it is easier to verify them rather than reason about asynchronous execution contexts.

### 4.2.4 The ExpressOS kernel

The ExpressOS kernel is responsible for managing the underlying hardware resources and providing abstractions to applications running above. The ExpressOS kernel uses L4 to access the underlying hardware. L4 provides abstractions for various hardware-related activities, such as context switching, address space manipulation, and inter-process communication (IPC). L4 resides in the TCB of ExpressOS, although a variant of L4, seL4 [69], has been fully formally verified.

### 4.2.5 System services

ExpressOS separates subsystems as system services running on top of the ExpressOS kernel. These services include persistent storage, device drivers, networking protocols, and a window manager for displaying application GUIs and handling input. ExpressOS reuses the existing implementation from L4Android [72] to implement these services.

All these services are untrusted components in ExpressOS. They are isolated from the ExpressOS kernel. The isolation combined with other techniques (discussed in Section 4.5) ensures that the verified security invariants remain valid even if a system service is compromised.

### 4.2.6 Application abstractions

The ExpressOS kernel exports a subset of the Android system call interfaces directly to applications. These abstractions include processes, files, sockets, and IPC. ExpressOS supports unmodified Android applications like a Web Browser to run on top of it directly.

The ExpressOS kernel provides an alternative set of IPC interfaces that is more amenable to formal verification, but still enables applications to perform Android-like IPC operations. First, it exposes all IPC interfaces as system calls rather than using `ioctl` calls. Second, it requires all IPC channels to be annotated with permissions, so that the ExpressOS kernel can perform access control on IPC operations. This design choice enables us to verify IPC operations in ExpressOS.

## 4.3 Threat model

An untrusted Android application runs directly on ExpressOS. Such an application contains arbitrary code and data, along with a manifest listing all its required permissions. The user grants all permissions listed on the manifest to the application once he or she agrees to install it into the system. ExpressOS must be able to confirm that all activities of the application conform to its permissions, and all security invariants defined by ExpressOS (discussed in Section 4.5) must hold during the lifetime of the system.

The goal of ExpressOS is to allow a sensitive application to run on top of ExpressOS, whose persistent storage, run-time state, and other data should remain intact. Malicious applications cannot compromise any security invariants of the sensitive application.

The TCB of ExpressOS includes the hardware, the L4 microkernel, the compilers, and the language run-time. All system services in ExpressOS, however, are untrusted. ExpressOS should be able to maintain its security invariants for all applications even if the system services execute arbitrary code.

This chapter focuses only on confidentiality and integrity; availability is out of the scope of the chapter.

## 4.4 Formal methods background

In this section, we describe a few basic verification related concepts for readers who are not familiar with formal techniques. Experienced readers could skip this section.

The idea of verification is to use a verifier to reason about the implementation of a program: whether its behavior matches programmer's intentions, which are formalized and referred to as the *specification*.

The Floyd-Hoare style approach to verification achieves modular verification by annotating each function/method with pre- and post-conditions, and annotating the rest of the code with appropriate assertions that capture the specification. In this chapter we focus on proving *safety properties*, which models properties that fail in finite time (availability is not a safety specification, typically). In particular, a *pre-condition* (*post-condition*) specifies certain assertions that have to hold before (after) executing a function. *Object invariants* refer to assertions that have to hold both before and after executing any methods in the object. Figure 4.4 shows an example of pre- and post-conditions used in ExpressOS.

Complex specifications, such as the security specifications we prove in our code, cannot always be stated in terms of assertions on the program state that is in scope at the point of assertion. This leads us to use *ghost code* in the verification. Ghost code is code added to the original program to aid formal reasoning during verification but is not needed at run-time. Ghost code changes only the *ghost states* associated with *ghost variables*, which are annotated with the keyword `ghost`. The ghost state can never change the semantics of the original code (for instance, a conditional on a ghost variable followed by an update of a real program variable is disallowed, syntactically). Furthermore, ghost code must always be provably terminating.

Ghost variables can range over mathematical types that are not supported in the programming language, including abstract sets, lists, etc., and can be used for abstracting the data contained in the program using mathematical objects, and thus encode specifications naturally (for example, a ghost sequence variable can track the sequence of keys stored in a linked list and we can then assert that this sequence is sorted). Apart from the ghost code for mathematical abstractions of the specification, the code is also annotated with additional information to prove a property of a correct program (typical examples include updating ghost states to reflect how the code meets the specification, inductive invariants on loops to facilitate automated reasoning for recursion, etc.). One good example of ghost variables in use is the `CurrentState` variable shown in Figure 4.4, which help maintain information during verification about the current state of the `Page` object.

The verification system (ours are based mostly on the suite of BOOGIE-based verification tools, including Dafny) proceeds to verify the code by taking straight-line fragments flanked by pre-

and post-annotations, and derives verification conditions, which capture the semantics of the program, and are purely logical statements whose validity implies correctness of the code segments. These logical statements are, thanks to ghost code, often expressible in quantifier-free first-order theories combining arithmetic, arrays, recursive data-structures, sets, sequences, etc., and can be discharged using automatic constraint solvers, especially the emerging powerful class of SMT solvers.

For some of the simpler specifications, a completely automatic analysis based on *abstract interpretation* is feasible. We use the code contracts framework to achieve this in ExpressOS.

## 4.5 Proving security invariants

This section describes that how we apply the techniques described in Section 4.2 to verify security invariants on the implementation of storage, memory management, user interface, and IPC in ExpressOS.

### 4.5.1 Secure storage

The storage system of ExpressOS provides guarantees of access control, confidentiality, and integrity for applications, yet still offers the same sets of storage APIs as Linux. The storage system is implemented as an additional layer on top of the basic storage APIs (e.g., `open()`, `read()`, and `write()`), which are provided by the untrusted storage service.

The first security invariant for secure storage that the ExpressOS kernel enforces is:

**SI 4.1.** *An application can access a file only if it has appropriate permissions. The permissions cannot be tampered with by the storage service.*

An application can implement its security policy directly on top of SI 4.1. For example, an application might restrict the permission of a file so that only the application itself has access to it.

Since other compromised components such as the storage service and device drivers can affect SI 4.1, the first step of verifying SI 4.1 is to isolate the effects of these services. The ExpressOS kernel uses the HMAC algorithm [8] to achieve this goal. The ExpressOS kernel prepends several pages to all files managed by the secure storage system. These pages store metadata such as

```
static SecureFSInode Create(Thread current, ByteBufferRef metadata, ...)
{
  ...
  var ret = InitializeAndVerifyMetadata(...);
  ...

  var metadata_verified = ret == 0;
  ...

  var access_permission_checked
    = SecurityManager.CanAccessFile(current, metadata);
  ...

  // Verify both Property 1 and Property 2
  Contract.Assert(metadata_verified && access_permission_checked);

  return ...;
}
```

Figure 4.2: Relevant code snippets in C# for Property 4.1 and Property 4.2.

permissions, the size of the file, as well as an HMAC signature of these pages. The ExpressOS kernel checks the HMAC signature to ensure the integrity of the metadata when loading the file into the memory.

Now SI 4.1 can be reduced to the following two lower level properties:

**Property 4.1** (Integrity-Metadata)**.** *The signature of a file's metadata is always checked before an application can perform any operations on it.*

**Property 4.2** (Access Control)**.** *An application can only access a file when it has appropriate permissions.*

Figure 4.2 shows relevant code of Property 4.1 and Property 4.2. The `Create()` function calls `InitializeAndVerifyMetadata()` to verify the HMAC signature of the metadata. Then it calls `SecurityManager.CanAccessFile()` to determine whether the current process has appropriate permissions to open the file. Finally, the annotation of `Contract.Assert()` instructs the verifier to prove that both the integrity of the metadata, and the access permission of the file have been checked.

The reduction is a trade-off between formal verification and practicality. We argue that pragmatically the conjunction of Property 4.1 and Property 4.2 implies SI 4.1. The reduction captures the important fact that ExpressOS misses no security checks in its access control logic, which is the main point of verifying ExpressOS. It does assume the implementation of relevant libraries like AES / SHA-1, and the one of `InitializeAndVerifyMetadata()` and `Security-Manager.CanAccessFile()` is correct. These components can be verified independently, and a verified implementation can be plugged into the system to further strengthen the proof.[1]

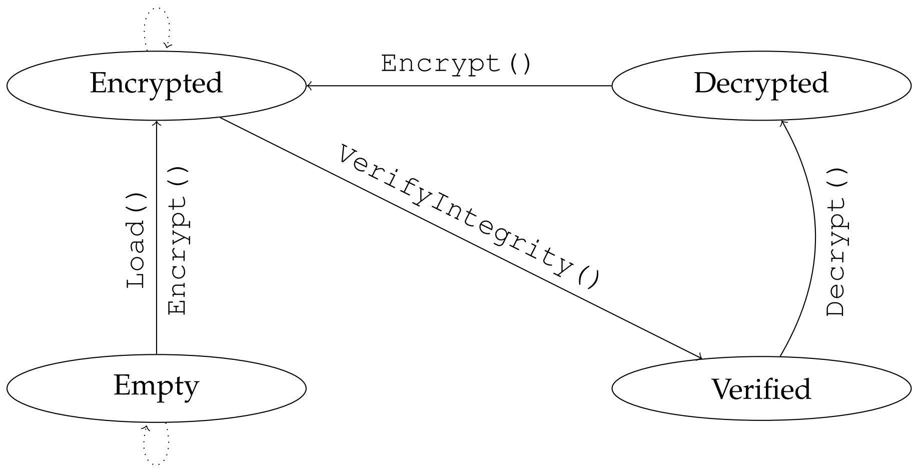


Figure 4.3: State transition diagram for the `Page` object. Ellipses represent the states. Solid and dotted arrows represent the successful and failed transitions.

The ExpressOS kernel also enforces integrity and confidentiality of its secure storage system:

**SI 4.2.** *Only the application itself can access its private data. Neither other applications nor system services can access or tamper with the data.*

Similar to SI 4.1, the ExpressOS kernel uses encryption to defend against compromised system services. From a high level, it partitions a file into multiple pages, and then it encrypts each page with AES for confidentiality. To ensure integrity, it signs each encrypted page with the HMAC algorithm, and packs the results to the metadata of the file. The overall implementation is similar to Cryptfs [131].

ExpressOS assigns each application a different private key during installation; therefore SI 4.2 can be reduced to the following two properties:

**Property 4.3** (Confidentiality)**.** *Every page is encrypted before it is sent to the storage service.*

[1]The `InitializeAndVerifyMetadata()` function parses binary data read from the disk and verifies its integrity, which is easier implemented in C. To demonstrate the feasibility of this approach, we have implemented `Initialize-AndVerifyMetadata()` in C, and verified its correctness with VCC [27].

```
class CachePage {
  enum State { Empty, Verified, Decrypted, Encrypted }
  [Ghost] State CurrentState;

  void Encrypt(...) {
    Contract.Requires(CurrentState == State.Empty || CurrentState == State.Decrypted);
    Contract.Ensures(CurrentState == State.Encrypted);
    ...
  }

  void Decrypt(...) {
    Contract.Requires(CurrentState == State.Verified);
    Contract.Ensures(CurrentState == State.Decrypted);
    ...
  }

  // Verify the integrity of the page, returns true if the page is authentic.
  bool VerifyIntegrity(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    Contract.Ensures(!Contract.Result<bool>() || CurrentState == State.Verified);
    ...
  }

  // Load the content of the page from the storage service.
  bool Load(...) {
    Contract.Requires(CurrentState == State.Empty);
    Contract.Ensures(!Contract.Result<bool>() || CurrentState == State.Encrypted);
    ...
  }

  void Flush(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    ...
  }
}
```

Figure 4.4: Relevant code snippets in C# for Property 4.3 and Property 4.4. `Contract.Requires()` and `Contract.Ensures()` specify the pre- and post-conditions of the functions.

**Property 4.4** (Integrity). *Each page loaded from the storage service has the appropriate integrity signature.*

The idea behind verifying Property 4.3 and Property 4.4 is to use the ghost variable `CurrentState` to record the current state of the page. Figure 4.3 shows the state transition diagram for a `Page` object. A page can be in the state of `Empty`, `Verified`, `Decrypted`, and `Encrypted`, meaning that (1) the page is empty, (2) its integrity has been verified, (3) its contents have been decrypted, and (4) its contents have been encrypted. To verify these properties, we specify the valid state transitions as the pre- and post-conditions of the relevant functions. For example, the specifications of `Decrypt()` state that the page should has its integrity verified before entering the function, and its contents are decrypted afterwards.

Figure 4.4 shows the relevant code snippets. Notice that Property 4.3 can be specified as a precondition of the function `Flush()`: the function can only be called if the page is in the `Encrypted` state.

For compatibility reasons, ExpressOS does allow an application to create unencrypted, public readable files. ExpressOS, however, does not provide additional security invariants for these files.

### 4.5.2 Memory isolation

The ExpressOS kernel enforces proper access control and isolation for all memory of the applications:

**SI 4.3.** *If a memory page of an application is backed by a file, the pager can map it in if and only if the application has proper access to the file.*

**SI 4.4.** *An application cannot access the memory of other applications, unless they explicitly share the memory.*

The challenge of verifying SI 4.3 is that there is insufficient information available for verification at the point of assertions (i.e., in the pager). This is because the security checks are executed in different contexts, where both the control and data flows are separated in these two contexts.

ExpressOS addresses this challenge by connecting the information indirectly through the *object invariants* of relevant data structures. It strengthens these object invariants to contain information about the security checks, so that the object invariants can derive the desired security invariants.

Figure 4.5 and Figure 4.6 show the relevant implementation of the pagers. From a high level, the ExpressOS kernel organizes the virtual memory of a process with a series of `MemoryRegion` objects. A `MemoryRegion` represents a continuous region of the virtual memory, which has information on its access permissions and location. In addition, if the region is mapped to a file, it contains a reference to the mapped file (i.e., the `File` field in the `MemoryRegion` class). Since we have verified that the ExpressOS kernel properly checks access to files in Section 4.5.1, SI 4.3 can be reduced to the following property:

**Property 4.5.** *When the page fault handler serves a file-backed page for a process, the file has to be opened by the same process.*

```
static uint HandlePageFault(Process proc, uint faultType,
                            Pointer faultAddress, Pointer faultIP) {
  Contract.Requires(proc.ObjectInvariant());
  ...
  AddressSpace space = proc.Space;
  var region = space.Regions.Find(faultAddress);

  if (region == null || (faultType & region.Access) == 0)
    return 0;

  ...
  var shared_memory_region = IsSharedRegion(region);
  var ghost_page_from_fresh_memory = false;

  if (!shared_memory_region) {
    page = Globals.PageAllocator.AllocPage();
    ghost_page_from_fresh_memory = true;
    ...

    if (region.File != null) {
      // Assertion of Property 5
      Contract.Assert(region.File.GhostOwner == proc);
      var r = region.File.Read(...);
      ...
    }
    ...
  } else { ... }

  Contract.Assert(shared_memory_region ^ ghost_page_from_fresh_memory);
  ...
}
```

Figure 4.5: Code snippets for the page fault handler.

The key of verifying Property 4.5 is to use a ghost variable to record the *ownership* of the relevant objects, and to specify object invariants based on the ownership. For example, the first assertion in Figure 4.5 specifies Property 4.5, which gets verified through a series of object invariants.

First, the `AddressSpace` class represents the virtual address space of a process by a sequence of `MemoryRegion` objects. Intuitively the `AddressSpace` object "owns" all `MemoryRegion` objects in the sequence, which is specified in the object invariant of the `AddressSpace` object:

$$\begin{aligned}\forall \mathtt{x},\ \ \mathtt{x} \in \mathtt{Contents} \rightarrow\ \ & \mathtt{x} \neq \mathtt{null} \wedge \mathtt{x.ObjectInvariant()} \\ & \wedge \mathtt{x.GhostOwner}\ \mathtt{==}\ \mathtt{GhostOwner}\end{aligned}$$

The `Find()` method looks up and returns the corresponding `MemoryRegion` object for the virtual address, therefore it only returns `MemoryRegion` objects that are owned by the `Address-`

```
class AddressSpace {
  var GhostOwner: Process;
  var Head: MemoryRegion;
  // The sequence of MemoryRegions owned by this AddressSpace
  ghost var Contents: seq<MemoryRegion>; ...

  function ObjectInvariant() : bool ... {
    ∀ x,x ∈ Contents → x ≠ null ∧ x.ObjectInvariant() ∧ x.GhostOwner == GhostOwner
    ...
  }

  method Find(address: Pointer) returns (ret: MemoryRegion)
  maintains ObjectInvariant();
  ensures ret ≠ null → ret.ObjectInvariant() ∧ ret.GhostOwner == GhostOwner;

  method Insert(r: MemoryRegion)
  maintains ObjectInvariant();
  requires r ≠ null ∧  r.ObjectInvariant() ∧ r.GhostOwner == GhostOwner;
}

class MemoryRegion {
  var GhostOwner: Process;
  var File: File;  ...
  function ObjectInvariant() : bool ... {
    File ≠ null → File.GhostOwner == GhostOwner
    ...
  }
}

class Process {
  var space: AddressSpace;
  function ObjectInvariant() : bool ... {
    space ≠ null ∧ space.GhostOwner == this
    ...
  }
}

class File { ... var GhostOwner: Process; }
```

Figure 4.6: Reduced code snippets in Dafny for the `MemoryRegion` and the `AddressSpace` class. A `maintains` clause specifies the condition as both pre- and post-conditions of the function.

`Space` object. Combining it with the object invariant above can lead to the post-condition of `Find()`:

```
ret ≠ null → ret.ObjectInvariant() ∧ ret.GhostOwner == GhostOwner
```

At the first assertion in Figure 4.5, this is simplified down to:

```
region.File.GhostOwner  ==  region.GhostOwner  ==  space.GhostOwner
```

The object invariant of the `proc` object ensures that

```
proc.space.GhostOwner == proc
```

Which leads to the assertion of Property 4.5.

Property 4.5 is strictly weaker than the property that ensures full functional correctness. For example, Property 4.5 does not enforce that the file used in page fault handler has to be the exact same file that was requested by the user. The property, however, shows that the file must be opened by the same process. It maintains isolation since the access to the file has been properly checked. Moreover, it can be verified through object invariants.

We combine code contracts and Dafny to verify this property. Dafny verifies the `Memory-Region` and `AddressSpace` class, because Dafny is able to reason about the linked lists in their implementation. The verification results from Dafny are expressed as properties of ghost variables (i.e., the `GhostOwner` fields). These properties are exported to code contracts as ground facts with the `Contract.Assume()` statements. Using `Contarct.Assume()` is a simple way let code contracts know about proofs about Dafny code. Additionally, we annotate the `GhostOwner` fields as read-only fields in C# to ensure soundness.

SI 4.4 can be expressed in a slightly different way to ease the verification.

**Property 4.6** (Freshness)**.** *The page fault handler maps in a fresh memory page when the page fault happens in non-shared memory.*

The idea is to ensure that the page fault handler always allocates a fresh memory page, i.e., a memory page that is not overlapped with any allocated pages. ExpressOS adopted the verified memory allocator from seL4 [115] for this purpose. ExpressOS dedicates a fixed region to this allocator in order to implement this property. The reduction allows specifying this property as the second assertion in Figure 4.5.

### 4.5.3 Address space integrity

The ExpressOS kernel enforces the integrity of applications' address spaces:

**SI 4.5.** *The kernel never corrupts the address space for an application.*

More concretely, we reduce SI 4.5 down to the following properties:

**Property 4.7.** `brk()` *always returns a virtual memory address on the heap, and it sets up the memory mappings correctly.*

**Property 4.8.** *`mmap()` always returns free virtual memory address when they are used to allocate new memory.*

**Property 4.9.** *Any reads and writes to any user-level buffers never goes out of bounds.*

These properties prevents the attacker hijacking the control flow of an application through Iago attacks [23].

We verify the above properties with three different strategies. Property 4.8 is a direct by-product when we verify the implementation details of SI 4.4. Verifying Property 4.7 is straightforward since the sole functionality of `brk()` is to manipulate the heap pointer in the process data structure. For Property 4.9, we verify each read and write to the applications' memory.

The ELF parser, however, parses the application executable and defines the start of the heap of the application. We plan to investigate verifying its implementation, or introducing code attestation [54, 105] into the kernel as future work.

### 4.5.4 UI isolation

The ExpressOS kernel enforces the following UI invariant:

**SI 4.6.** *There is at most one currently active (i.e., foreground) application in ExpressOS. An application can write to the screen buffer only if it is currently active.*

To write to the screen, an application requests shared memory from the window manager, and writes screen contents onto the shared memory region. In ExpressOS, this can only be done through an explicit API so that the ExpressOS kernel knows exactly which memory region is the screen buffer.

The ExpressOS kernel enforces SI 4.6 by explicitly enabling and disabling the write access of screen buffers when changing the currently active application.

```
class UIManager {
  Process ActiveProcess;

  [ContractInvariantMethod]
  void ObjectInvariantMethod() {
    Contract.Invariant(ActiveProcess == null || ActiveProcess.ScreenEnabled);
  }

  void OnActiveProcessChanged(Process next) {
    Contract.Requires(next != null);
    Contract.Ensures(ActiveProcess == next);
    ...
  }

  void DisableScreen() {
    Contract.Ensures(ActiveProcess == null);
    Contract.Ensures(!Contract.OldValue(ActiveProcess).ScreenEnabled);
    ...
  }

  void EnableScreen(Process proc) {
    ...
    Contract.Ensures(ActiveProcess == proc);
    Contract.Ensures(ActiveProcess.ScreenEnabled);
    ...
  }
}
```

Figure 4.7: Relevant code snippets for SI 4.6. `[ContractInvariantMethod]` annotates the method that specifies the object invariants.

The object invariant of `UIManager` in Figure 4.7 specifies SI 4.6. The boolean ghost variable `ScreenEnabled` (which is not shown in the paper) denotes whether the application has write access to the screen. The initial value of `ScreenEnabled` is set to false for each application. Since it is the only API to manipulate the screen, the object invariant implies that only the current application has write access to the screen buffer.

### 4.5.5 Secure IPC

To simplify verification, ExpressOS provides an alternative, secure IPC (SIPC) interface over the Android's IPC interface. First, SIPC exposes all IPC functionality explicitly through system calls to eliminate the implementation and verification efforts on complex logic in `ioctl()` of Android's IPC. Second, the ExpressOS kernel enforces proper access controls for SIPC, compared to relying on the receiver of Android's IPC for proper access control. This design moves the access control logic of SIPC into the ExpressOS kernel.

SIPC provides basic functionality to the applications, including creating SIPC channels, connecting to SIPC channels, and sending and receiving messages over the channel. Applications can still perform Android-like IPC operations using the SIPC interface.

The ExpressOS kernel enforces the following security invariants for SIPC:

**SI 4.7.** *An application can only connect to SIPC channels when it has appropriate permissions.*

**SI 4.8.** *An SIPC message will be sent only to its desired target.*

SI 4.7 and SI 4.8 can be verified with similar approaches described above.

SI 4.7 is an access control invariant, thus the strategy of proving SI 4.7 is similar to the one of SI 4.1. We use a ghost variable to indicate whether the process has properly checked the permissions when opening a new SIPC channel.

We follow the proving strategy of Property 4.5 to verify SI 4.8. The idea is to create a `SIPC-Message` object for each IPC message, and to introduce a ghost variable `target` to record the target process of the message, which provides sufficient information to verify SI 4.8.

## 4.6 Implementing ExpressOS

The implementation of ExpressOS consists of two parts: the ExpressOS kernel and ExpressOS services. The ExpressOS kernel is a single-thread, event-driven kernel built on top of L4::Fiasco. We have implemented the kernel in C# and Dafny, which is compiled to native X86 code using a static compiler.

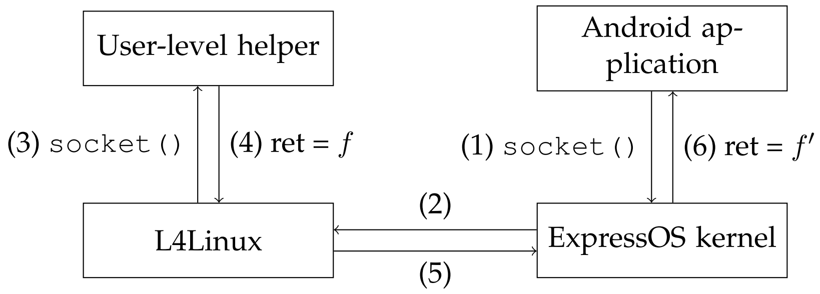


Figure 4.8: Work flow of handling the `socket()` system call in ExpressOS. The arrows represent the sequences of the work flow.

The implementation of ExpressOS kernel includes processes, threads, synchronization, memory management (e.g. `mmap()`), secure storage, and secure IPC. The kernel also implements a subset of Linux system calls to support Android applications like the Android Web browser. The kernel contains about 15K lines of code. The source code is available at `https://github.com/ExpressOS`.

The ExpressOS kernel delegates the implementation of system calls to ExpressOS services whenever it does not affect the soundness of the verification. These services include file systems, networks, device drivers, as well as Android's user-level services like window manager and service manager. ExpressOS reuses L4Android to implement these services. L4Android is a port of Android to a Linux kernel that runs on top of L4::Fiasco (i.e., L4Linux).

The rest of the section describes (i) how to dispatch a system call to ExpressOS services, (ii) how to bridge Android's binder IPCs between ExpressOS and Android's system services, and (iii) how to support shared memory between an application and ExpressOS services.

*Dispatching a system call to ExpressOS services.* The ExpressOS kernel forwards system calls with IPC calls to L4Android. Figure 4.8 shows the workflow of handling the `socket()` system call in the ExpressOS kernel. When (1) the application issues a `socket()` system call to the ExpressOS kernel, (2) the kernel wraps it as an IPC call to the L4Linux kernel. The L4Linux kernel executes the system call (which might involve a user-level helper like the step (3) & (4)), and (5) returns the result back to the ExpressOS kernel. The ExpressOS kernel (6) interprets the result and returns it to the application.

It is important for the ExpressOS kernel to maintain proper mappings between the file descriptors (fd) of the user-level helper and those of the application. In Figure 4.8, it maps between the fd $f$ and $f'$ so that subsequent calls like `send()` and `recv()` can be handled correctly. This workflow mirrors the implementation of the Coda file system inside Linux [68].

*Bridging Android's binder IPC.* Android applications communicate to Android system services (e.g., the window manager) through the Android's binder IPC interface. The ExpressOS kernel extends the mechanism in Figure 4.8 to bridge the binder IPC. The user-level helper in Figure 4.8 acts as a proxy between the Android application and Android system services. Both the user-level helper and the ExpressOS kernel transparently rewrite the IPC messages to support advanced features like exchanging file descriptors.

*Supporting shared memory.* To support shared memory between Android applications and Android services, the ExpressOS kernel maps all physical memory of L4Linux into its virtual address space. It maps the corresponding pages to the address space of the application when sharing occurs. We have modified L4Linux to expose its page allocation tables so that the ExpressOS kernel is able to compute the address. Both the ExpressOS kernel and L4Linux use the L4 Runtime Environment (L4Re) to facilitate this process.

## 4.7 Evaluation

This section describes our evaluation of ExpressOS. First, we examine 383 relevant real-world vulnerabilities to analyze the security of the system. Then we present the performance measurements of ExpressOS. Finally, we explore the generality of the techniques by applying them to a web application.

### 4.7.1 Vulnerability study

To understand to what extent ExpressOS is able to withstand real-world attacks, we studied 742 vulnerabilities (from Jun, 2011 to Jun, 2012) listed in CVE. 383 out of 742 are valid vulnerabilities, and they affect different components used in Android. We manually examined each of them, and classified it into one of the four categories based on its location:

*In the core of the kernel.* The vulnerability exists in the core of the Linux kernel, which means that the same functionality is implemented in the ExpressOS kernel. The proofs of ExpressOS ensure that such a vulnerability cannot affect any security invariants discussed in Section 4.5. If the vulnerability is irrelevant to the security invariants, the language run-time ensures it cannot subvert the control flow and data flow integrity to circumvent the proofs.

*In the libraries used by applications.* The vulnerability exists in the libraries used by the applications, like the Adobe Flash Player and libpng. In the worst case, the attacker can gain full control of applications by exploiting the vulnerability. ExpressOS ensures that the compromised application must adhere to its permissions, which prevents it from accessing other applications' private data, effectively protecting sensitive applications from compromised applications.

*In system services.* The vulnerability exists in the system services of ExpressOS, including the file system, the networking stack, device drivers, and Android user-level services. ExpressOS combines three techniques to contain the vulnerability.

First, ExpressOS uses end-to-end security mechanisms and the protections provided by the ExpressOS kernel to protect file system and network data. For example, both the confidentiality and integrity of any private data remain intact when the storage service is compromised, because SI 4.2 ensures that the attacker cannot access or tamper with private data. Similarly, the attacker cannot eavesdrop or tamper with any TLS/SSL/HTTPS connections, even if he or she compromises the networking service.

Second, ExpressOS isolates applications and system services, and restricts updates to the screen (i.e., SI 4.6) to contain compromises of the window manager service. An attacker with a compromised window manager takes full control of the physical screen. Successful attacks, however, still require information about UI widgets in the targeted application, and the ability to provide timely visual feedback to the user. For example, the attacker might steal the user's input by overlaying a malicious application running in background on top of the targeted application. The isolation mechanism prevents the window manager from accessing the memory of the targeted application to retrieve the exact locations of UI widgets, and SI 4.6 prevents the malicious application running in background updating the screen to timely react to the user's input.

Third, the L4 layer isolates the system services and the ExpressOS kernel. An attacker can potentially compromise the L4Android kernel with a vulnerability. However, the ExpressOS kernel remains intact because the L4 layer manages allocation of physical memory and the IOMMU, ensuring that the ExpressOS kernel's memory is isolated from all system services.

ExpressOS currently does not prevent compromises where a service acts as a privileged deputy that allows the attacker to use its permissions to attack the system. For example, "the Bluetooth service in Android 2.3 before 2.3.6 allows remote attackers within Bluetooth range to obtain contact data via an AT phone book transfer." (CVE-2011-4276), and "the HTC IQRD service for Android ... does not restrict localhost access to TCP port 2479, which allows remote attackers to send SMS messages." (CVE-2012-2217).

*In sensitive applications.* If an application is exploited, there is not much that ExpressOS can do for that application. Although we proved our implementation of several security policies in ExpressOS, if an application configures its policy incorrectly or its application logic leads to a security compromise, there is little the ExpressOS kernel can do to protect it.

| Location | Example | Num. | Prevented |
|---|---|---|---|
| The core of the kernel | Logic errors in the futex implementation allow local users to gain privileges. | 9 | 9 (100%) |
| Libraries of applications | Buffer overflow in libpng 1.5.x allows attackers to execute arbitrary code. | 102 | 102 (100%) |
| System services | Missing checks in the vold daemon allows local users to execute arbitrary code. | 240 | 226 (93%) |
| Sensitive applications | The BoA application stores a security question's answer in clear text which allows attackers to obtain the sensitive information. | 32 | 27 (84%) |
| Total | | 383 | 364 (95%) |

Table 4.1: Categorization on 383 relevant vulnerabilities listed in CVE. It shows the number of vulnerabilities that ExpressOS prevents.

Table 4.1 summarizes our analysis of 383 vulnerabilities. ExpressOS is able to prevent 364 (95%) of them.

Large portions of vulnerabilities are due to memory errors in application libraries or in the system services. The verified security invariants and the protection from the ExpressOS kernel protect the data of sensitive applications from being compromised.

Out of the 383 vulnerabilities, there are two vulnerabilities related to covert channels. For example, the Linux kernel before 3.1 allows local users to obtain sensitive I/O statistics to discover the length of another user's password (CVE-2011-2494). These types of vulnerabilities are beyond the scope of our verification efforts and something ExpressOS is unable to prevent.

### 4.7.2 Performance

We evaluate the performance of ExpressOS by measuring the execution time of a variety set of benchmarks. We compared the performance of benchmarks running on ExpressOS, unmodified L4Android, and Android-x86. All experiments run on an ASUS Eee PC 1005HA with an Intel Atom N270 CPU running at 1.60 GHz, 1GB of DDR2 memory, and a Seagate ST9160314AS 5,400

RPM, 160G hard drive. Our Eee PC connects to the campus network through its built-in Atheros AR8132 Fast Ethernet NIC.

Both ExpressOS and L4Android run the L4Linux 3.0.0 kernel. The Android-x86 runs on top of Linux 2.6.39. All three systems run the same Android 2.3.7 (Gingerbread) binaries in user spaces.

We evaluate the performance of the Android web browser on real network, and microbenchmarks evaluating different aspects of system performance, including IPC, the file system, the graphics subsystem, and the networking stack. All numbers reported in this section are the mean of five runs.

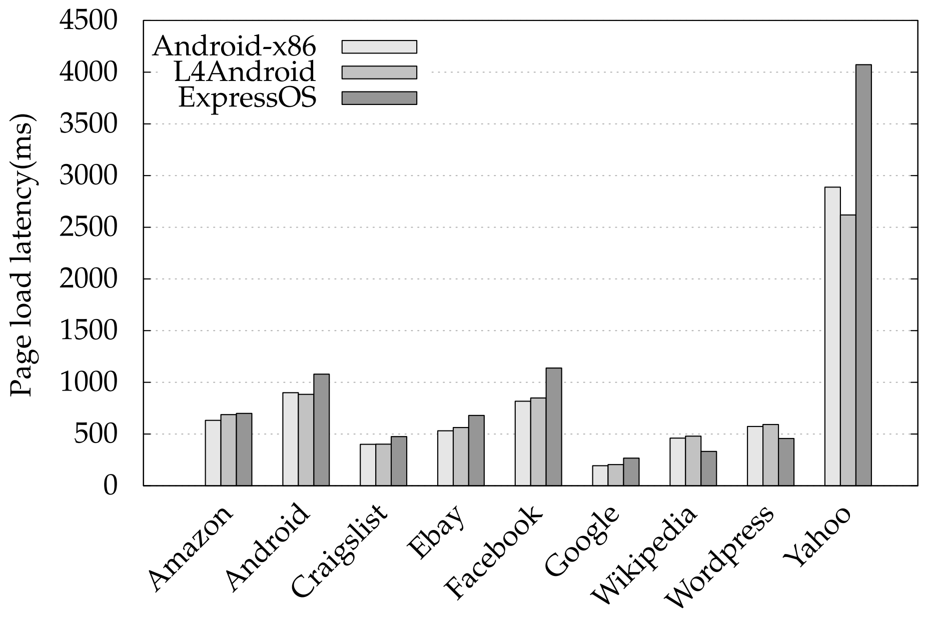


Figure 4.9: Page load latency in milliseconds for nine web sites for the same browser under Android-x86, L4Android, ExpressOS over a real network connection.

*Page load latency in web browsing.* We measure the page load latency for nine popular web sites to characterize the overall performance of ExpressOS compared to L4Android and Android-x86. The page load latency for a web site is the latency from initial URL request to the time when the browser fires the DOM `onload` event. The app clears all caches between each run.

Figure 4.9 shows the page load latency for nine web sites of all three systems. L4Android has 2% overhead on average, suggesting that the microkernel layer added by ExpressOS adds little overhead in real-world web browsing.

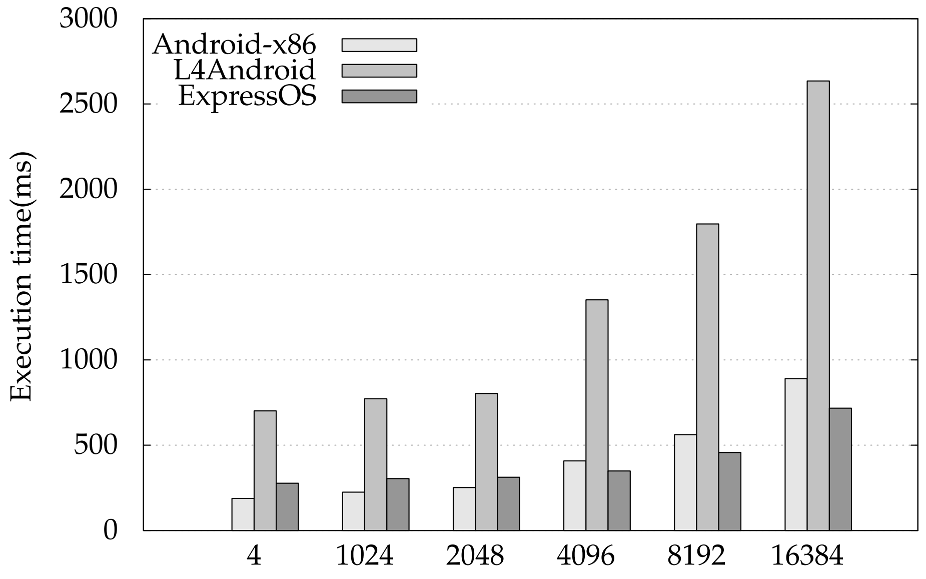


Figure 4.10: Performance of the ping-pong IPC benchmark of Android-x86, L4Android, and ExpressOS. X axis shows the size of IPC messages, Y axis shows the total execution time of running $10,000$ rounds of ping-pong IPC.

ExpressOS shows $14\%$ and $16\%$ overhead on average compared to L4Android and Android-x86.

*IPC performance.* We uses a simple ping-pong IPC benchmark to compare the performance of the SIPC mechanism in ExpressOS against the Android's Binder IPC mechanism. There are two entities (the server and the client) in this benchmark. For each round, the client sends a fixed-size IPC message to the server. The server receives the message and sends an IPC message back to the client which has the same content. Then the client receives the reply and continues to the next round.

We measured the total execution time for $10,000$ rounds of this benchmark on Android-x86, L4Android and ExpressOS. We measured the performance with different message sizes, including four bytes, 1KB, $4$KB, $8$KB, and $16$KB. Figure 4.10 describes the results of this benchmark. These numbers show that it is possible to implement a verifiable IPC mechanism in the ExpressOS architecture without sacrificing efficiency.

*Other microbenchmarks* . We further evaluate the performance of ExpressOS with three microbenchmarks, including (1) a SQLite benchmark (SQLite) which creates a new database, then inserts

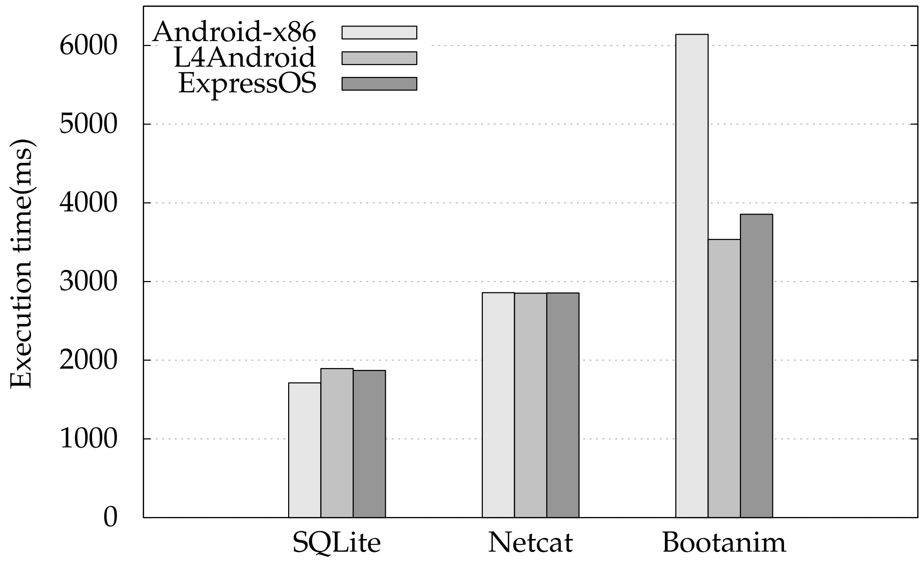


Figure 4.11: Performance results of the SQLite, Netcat and the Bootanim microbenchmark. X axis shows the type of the benchmark. Y axis shows their total execution time in milliseconds.

$25,000$ records in one transaction, and writes all data back to the disk. (2) A network benchmark (Netcat), which receives a $32$M file from local network. (3) A graphics benchmark (Bootanim), showing a PNG image, and adding light effects with OpenGL, which is derived from the boot animation program from Android.

These microbenchmarks help to categorize different aspects of the ExpressOS's performance. First, the SQLite benchmark manipulates the heap heavily, thus it is used to evaluate the performance of memory subsystem. Second, the Netcat benchmark helps quantify the effects of microkernel servers, because ExpressOS delegates all networking operations to L4Android. Finally, the Bootanim benchmark helps quantify the cost of using user-level helpers. Manipulating the screen heavily relies on Android's Binder IPC and shared semaphores, both of which are forwarded back and forth between the ExpressOS kernel and the user-level helpers in L4Android.

Figure 4.11 describes the results of all three microbenchmarks above. For the SQLite benchmark, both ExpressOS and L4Android are about $10\%$ slower than Android-x86. For the Netcat benchmark, Android-x86, L4Android, and ExpressOS perform almost the same. ExpressOS is about $10\%$ slower than L4Android in the Bootanim benchmark, but surprisingly, Android-x86 performs significantly worse than both L4Android and ExpressOS. We suspect that it might be due to some subtle differences between the kernel of L4Android and Android-x86, since all three systems are using the same user-level binaries.

### 4.7.3 Case study: SafeCRP

We implement a conference management system called SafeCRP that is similar to the HotCRP [70] system. SafeCRP is a standard three-tier web service implemented in C#, combining the HTTP server, the application logic, and the persistent storage layer into a single .NET executable. The architecture is similar to the one of a Jetty application [37].

We apply the techniques of ExpressOS to improve its security. We attempt to verify that SafeCRP faithfully implement the following two security properties:

**SI 4.9.** *SafeCRP can only send a user's password to his or her e-mail, or reveal the password to the PC chairs.*

**SI 4.10.** *SafeCRP never reveals the lists of authors when the submission is anonymous.*

Yip et al. have implemented these two properties in HotCRP using data flow assertions [128]. In SafeCRP, we verified both SI 4.9 and SI 4.10 using code contracts.

Figure 4.12 shows the relevant code snippets for SI 4.10. The variable `IsAnonymous` and `revealedAuthors` represent that whether the submission is anonymous and whether the list of authors has been revealed. Therefore, the assertion in Figure 4.12 is sufficient to verify the security of `HandleGetPaperInfo()`.

The verification described in Figure 4.12 implicitly assumes the integrity of `IsAnonymous`. A bug in SafeCRP might change the value of `IsAnonymous` incidentally, or the persistent storage layer returns incorrect values at the first place, both of which might lead to a violation to SI 4.10. To verify this assumption, we verify that all modifications on `IsAnonymous` match our intentions. We also verify that SafeCRP checks the integrity of the persistent stroage correctly, which is similar to Property 4.1.

It is debatable that how much security that the verification offers in this case study in practice. Indeed SafeCRP has the operating system, the language run-time, etc. in its TCB, where a vulnerability could potentially void all the proofs. We believe, however, that the techniques represent

```
void HandleGetPaperInfo(...) {
    Paper p = Store.Papers.Get(...);
    if (p == null) {
        HandleUnauthorized(...);
        return;
    }

    ...
    bool revealedAuthors = false;
    if (!p.IsAnonymous) {
        revealedAuthors = true;
        out.Append("Authors: <p>" + p.Authors + "</p>");
    }
    else {
        out.Append("Authors: <p>Anonymized</p>");
    }

    Contract.Assert(!p.IsAnonymous || !revealedAuthors);
}
```

Figure 4.12: Relevant code snippets to implement SI 4.10 in SafeCRP.

a practical trade-off between verification effort and security. We also believe that it is straightforward for other systems written in static type-safe languages to adopt these techniques to improve their security.

## 4.8 Discussion

This section discusses our experience with ExpressOS and its limitations. First, we discuss how close the security invariants are to end-to-end security guarantees. We then describe the assumptions and the design trade-offs in regards to introducing a new component into ExpressOS. Finally, we discuss the limitations of ExpressOS.

### 4.8.1 Towards end-to-end security invariants

The ExpressOS architecture allows the verifier to prove the security invariants within the local boundaries of the modules. One question that naturally arises is how strong these invariants are, that is, to what extent these locally proven security invariants can provide global, end-to-end security guarantees across the whole ExpressOS kernel.

Table 4.2 summarizes whether different security invariants provide end-to-end security for applications. In ExpressOS, all security invariants except for SI 4.6 provide end-to-end security

| Security invariants | Summary | End-to-end security |
|---|---|---|
| SI 4.1 | All file accesses of an application conform to its permissions, where the integrity of the permission data is protected. | Yes |
| SI 4.2 | ExpressOS isolates applications' private data and protects the integrity of the data. | Yes |
| SI 4.3 | When the pager of ExpressOS maps in a file-backed memory page for an application, the application must have proper access to the file. | Yes |
| SI 4.4 | An application cannot access the memory of other applications, unless they explicitly share the memory. | Yes |
| SI 4.5 | An application's address space is always well-defined. | Yes |
| SI 4.6 | There is at most one currently active application. An application can write to the screen buffer only if it is currently active. | No |
| SI 4.7 | An application can connect to SIPC channels only if it has appropriate permissions. | Yes |
| SI 4.8 | SIPC messages can only be dispatched to its desired target. | Yes |

Table 4.2: Summary of security invariants, and whether the invariants provide end-to-end security for applications.

guarantees.

We enforce two programming principles to simplify the verification on end-to-end security guarantees. The first principle is to enforce strong modularity. We mark all of the components' internal variables as private or read-only, thus an unverified component can only interact with the verified ones through their public interfaces. We then verify that the components' security invariants are always upheld with respect to their public interfaces, thus it is safe to conclude that an unverified component cannot subvert any security invariants. The second principle is to separate verifying security invariants from object alias analysis. Particularly, we parameterize the functions so that the verifier has enough information to prove the security invariant within the local scopes.

For example, we construct our proofs of SI 4.3 using the `GhostOwner` variable (Figure 4.5), which records the process that created the object. We annotate the variable as a read-only variable, whose value cannot be changed by any function (including the unverified ones) once the object has been initialized. We then verify that the constructor initializes the variable correctly. Therefore, the verifier only needs to inspect the `AddressSpace` class when verifying SI 4.3.

Another example in Figure 4.5 shows how we pass the process object (the `proc` variable) into the `HandlePageFault` function. What the verifier is essentially proving is that Property 4.5, which is the main part of SI 4.3, holds for any page faults of any processes. The key advantage of this design is that it does not require a global alias analysis on the `proc` variable. Indeed, under this design we do not verify that the caller always passes the correct process object into the function. The design, however, maintains the desired security property, since the property holds for all page faults of all processes.

Ideally SI 4.6 should enforce isolation between the screen buffers of different applications. Although the above programming principles still apply, the current implementation of SI 4.6 only enforces that only the currently active application can write to its screen buffer. This is because ExpressOS lacks of control of the window manager, which runs on top of the untrusted Linux kernel. In this case SI 4.6 does not provide end-to-end security guarantees, and in general any invariants that rely on the untrusted code to isolate the data cannot provide verifiable, end-to-end security guarantees. This is a limitation of the current implementation; the next subsection discusses potential approaches to mitigate it.

### 4.8.2 Trust and protection of unverified components

The principle of ExpressOS is to achieve verified security invariants without comprehensive verification of the entire software stack. Our experience shows that applying the principle to a new component in ExpressOS requires answering two important, but related questions: (1) to which extent the component needs to be trusted and (2) how to securely isolate data for different applications.

| Components | Degree of trust | Protection |
|---|---|---|
| File systems, block drivers, and networking stacks | No | End-to-end mechanisms |
| Random number generator | All | Isolation |
| Legacy device drivers | All | New architecture |

Table 4.3: Degree of trust and protection mechanisms used in ExpressOS for different components in the Linux kernel.

Table 4.3 summarizes the degree of trust and the protection mechanisms used in ExpressOS for different components in the Linux kernel. We categorize these components based on the protection mechanisms used to contain them in ExpressOS.

First, a component can remain untrusted if end-to-end mechanisms are available. The verification task is to prove that ExpressOS uses these mechanisms correctly. For example, SI 4.2 enforces both the confidentiality and integrity of persistent storage against compromised file systems or block drivers. ExpressOS protects the data with encryption. Each application has its own encryption key so that ExpressOS can isolate the data of different applications. In the case of defending a compromised networking stack, applications can use TLS / SSL to secure their network traffic.

Second, a component is susceptible to attack if ExpressOS cannot guarantee the integrity of its outputs. For example, ExpressOS relies on the random number generator in the Linux kernel. An attacker can break the cryptography used in ExpressOS when he or she compromises the random number generator. To reduce the attack surface of this component, it is important to isolate it from the untrusted Linux kernel (e.g., implementing it as a microkernel service).

Finally, the current implementation of ExpressOS has not addressed the problem of legacy device drivers. Therefore ExpressOS trusts legacy devices drivers and enforces no additional security invariants. However, we believe that alternative driver architectures, such as Exokernel [64], IBOS [113], and CuriOS [31] might be able to address the problem. Under these architectures, it is feasible to associate the state of the drivers (e.g., DMA buffers) directly to the application. The drivers no longer need to be trusted in this case and verifying isolation invariants is possible.

### 4.8.3 Verification experience

Overall, we found the verification effort in terms of annotations practical for the properties that were proven correct. While we developed the system and wrote code, we came up with the relevant security properties at the level of the module we were writing and formalized it using appropriate annotations. Combining code contracts and Dafny reduced the code-to-annotation ratio down to about $2.8\%$ (implementing the specification defined by the annotations). There were some instances where the code we wrote was incorrect, and using the verification tools to prove the property led us to find the error. Equally importantly, formalizing the specification at the level of the code crystallized the vague properties we had in mind, and helped us write better code as well.

One lesson that we had in ExpressOS was to refine the shapes and aliasing information of the objects through redesigning data structures, and implementing ownership using ghost variables. Simpler shapes eased the verification. For example, we have reimplemented the `MemoryRegion` object as a singly-linked list instead of a doubly-linked one, since verifying the manipulations of the linked list in the latter case requires specifying reachability predicates, which are difficult to reason about within SMT-based frameworks. The verification of heap structure properties in Dafny was achieved sometimes using further ghost annotations in the style of natural proofs [78, 98].

In simpler cases we used ownership to constrain the effect of aliasing. For example, the ghost field `GhostOwner` in the `AddressSpace` object specified which `Process` had created the object. The information was used in proving that each process creates its own `AddressSpace` object, effectively forbidding aliasing between `AddressSpace` objects of different processes.

One potential drawback we found during verification was that the specification using ghost code is sometimes too intimately interleaved with the implementation. Consequently, the specification gets strewn all across the code, and it is our responsibility that this actually is correct. Though the mathematical abstractions do help to some extent to distance the specification from the code, ghost updates to these abstractions are still intimately related. To illustrate this, consider a programmer implementing the code `Encrypt()` in Figure 4.4, and consider the scenario where the actual encryption fails for some reason, and yet the programmer puts the page into the `Encrypted` state. The verifier will go through even though the implementation is incorrect with respect to what the developer wanted; the onus of writing the correct specification is on the developer.

While we did not encounter any case where we noticed we made errors in inadvertently formulating too weak a specification, we did spend time double-checking that our *specifications* were indeed correct. We think an alternate mechanism for writing specifications that are a bit more independent from the code, resilient to code changes, and yet facilitates automated proving made the developer's work more robust and productive.

### 4.8.4 Limitations

There are several limitations in ExpressOS.

First, it might not always be possible to capture the application semantics at the system call level. For example, Android applications use the `ioctl` function for their IPC communications; they can also share memory by calling the `mmap` function on the same file. In these cases, it is difficult to extract or to verify security invariants. ExpressOS introduces new system calls (e.g., SIPC described in 4.5.5) to mitigate this problem, but the application itself also needs to be aware of the types of security invariants in ExpressOS and how to leverage them correctly.

Second, invariants related to availability or attestations could be difficult to verify in ExpressOS. This is because ExpressOS does not model the above behavior of the untrusted components. For example, we have verified the confidentiality and integrity of the persistent storage (SI 4.2), assuming only the storage layer is online. Wiping a file on the persistent storage, however, could be difficult to verify in ExpressOS. Notice that this is a design trade-off between the types of properties that ExpressOS can verify versus the assumptions that have to be made on the untrusted components. ExpressOS minimizes the assumptions made on the untrusted components and focuses on verifying isolation invariants.

Finally, verification limits the types of data structures available to ExpressOS. ExpressOS enforces stronger versions of object invariants of the container data structures in order to simplify the verification on other parts of the kernel. For example, ExpressOS ensures that the file descriptor table can only contain descriptors from a particular process so that any subsequent `get` operations must return file descriptors of the process. This requires specialized proofs on the details of the containers, which could be nontrivial for complex data structures like red-black trees. We anticipate that tighter integration between the type systems and the verification systems, as well as recent advances in formal verification [34, 98] can largely mitigate this limitation.

## 4.9 Related work

**Assurance from hypervisors.** Several proposals introduce a hypervisor into the software stack to remove the OS kernel from the TCB. For example, SecVisor [105] protects the code integrity of OS kernels. TrustVisor [88] provides a high assurance execution environment for small, isolated code. CloudVisor [133], Overshadow [25], and InkTag [54] secure the application data by encrypting the virtual memory and the persistent storage. These proposals effectively reduce the size of TCB because hypervisors are smaller than an OS kernel by orders of magnitudes. Research on microhypervisors [4, 111] further reduces the size of hypervisors.

Similar to Overshadow and InkTag, ExpressOS secures the persistent storage using encryption mechanisms. The main difference, however, is *where* the mechanisms are implemented. By implementing the mechanisms directly in the kernel, ExpressOS understands the application semantics, eliminating the semantic gap between the secure hypervisors and the applications, which is susceptible to attack [23]. Furthermore, formal verification removes ExpressOS from the TCB.

**Assurance from programming language techniques.** Much research proposes to eliminate memory errors by implementing kernels in type-safe languages [10, 58], isolating device drivers [112], and extending the languages or the compilers [30, 134]. Though useful, memory safety is only the first step to provide high assurance to the applications.

Several research proposals augment the type systems to automatically verify well-defined properties [33, 85]. Although they are effective to prove a subset of the properties described in this chapter, our experience shows that lightweight techniques are insufficient to derive full proofs of all security properties across the whole kernel. Therefore ExpressOS combines a heavyweight, expressive verifier (Dafny) and a lightweight verifier (code contract) to achieve the complete proofs.

**Assurance from alternative abstractions.** An application can express its own security policy through alternative abstractions such as information-flow control [132], capability [69, 106, 122], and logical attestations [108]. These abstractions simplify the specification and the enforcement of the security policy. For the browser application, Tahoma [29] encapsulates each browser instance into a virtual machine. IBOS [113] takes the concept further to align browser abstractions directly to the hardware abstractions.

ExpressOS is complementary to these approaches. These approaches can adopt the verification techniques of ExpressOS to verify their implementation.

**Assurance from formal verification.** Attempts to eliminate defects of operating systems with full formal verification date back to the late 1970s. Dealing with all details of real-world operating systems has been a challenge for heavyweight full formal verification methods. Verifying OSes that provide UNIX abstractions has been cumbersome (e.g., UCLA Secure Unix [118], PSOS [43], and KSOS [97] provide partially verified UNIX abstractions). Hypervisors and microkernels have lower-level abstractions that are more amenable for verification [11, 27, 55, 69, 106], but they provide lower-level abstractions such as IPC, interrupts and context switches, which are not immediately meaningful to applications.

The key difference between ExpressOS and the above work is that the verification of ExpressOS only focuses on security invariants rather than achieving full functional correctness. ExpressOS ensures that defects in unverified parts of the system cannot subvert the security invariants. As a result, ExpressOS provides high-level abstractions (e.g., files) with verified security invariants, and verifying these security invariants requires only $\sim 2.8\%$ annotation overhead.

## 4.10 Summary

This chapter has presented ExpressOS, a new OS architecture that provides formally verified security invariants to mobile applications.

The verified security invariants covers various high-level abstractions, including secure storage, memory isolation, UI isolation, and secure IPC. By proving these invariants, ExpressOS provides a secure foundation for sensitive applications to isolate their state and run-time events from malicious applications running on the same device.

The verification effort on ExpressOS focuses on the most important properties from a system builder's perspective rather than full functional correctness. Also, ExpressOS combines several verification techniques to further reduce the verification effort. The approach is relatively lightweight and has about $2.8\%$ annotation overhead.

Our evaluation shows that ExpressOS is effective in preventing existing vulnerabilities from different attack surfaces. Besides its strong security guarantees, ExpressOS is a practical system with performance comparable to native Android.

Our experience suggests that the verification technique we pursue is mature enough to be broadly used by systems developers in order to obtain lightweight proofs of safety and security by focusing on a small but crucial subset of properties.

# Chapter 5

# Conclusion

This dissertation developed techniques of identifying and designing abstractions of real-world computer systems to construct efficient automatic analysis. We discussed the techniques within the context of three different real-world systems: detecting network defects through analyzing the data plane, parallelizing web browsing by exploiting the data parallelism in web pages, and verifying security invariants in an operating system kernel.

In summary, we believe that this dissertation shows that identifying and designing abstractions for automatic analysis is a valuable approach to improve the performance, security, and reliability of computer systems.

## 5.1 Future directions

This section discusses some of the future directions that our techniques suggest.

**End-to-end verification for applications.** End-to-end proofs are immediately meaningful to the security of the applications. ExpressOS provides several verified security invariants, on top of which the application can implement their security policy with high assurance. While compatible with traditional POSIX abstractions, it might be difficult for the applications to leverage these proofs as lemmas, and to construct end-to-end proofs.

We believe that the techniques of ExpressOS still apply for end-to-end verification, thus it is possible to achieve these proofs without full-blown functional verification. We anticipate that several innovataions on the OS architecture [10, 58, 113] could help, but the problem remains an open research challenge.

**Integrating formal verification into programming languages.** Our experience shows that light-weight verification techniques (e.g., code contracts) are productive tools to verify common real-world invariants, but they fail to verify code that has even a few heap operations.

Our observation is that it is sufficient to augment the lightweight techniques by showing show isolation between objects on the heap when proving a common class of real-world invariants. Several examples include non-interference [46], determinism [14], and free of data-races [15]. Proving these properties today require using heavyweight formal methods, which hinders the productivity and applicability of formal verification. We believe that integrating recent advancement of program analysis [75] would open up an opportunity to allow formal methods to be used in a much boarder sets of systems.

## 5.2 A final remark

We leave the reader with one concluding remark.

Computer systems have to face the continuous challenges of performance, security, and reliability. Much of the work in this dissertation is to show that building abstractions that are friendly to automatic analysis is a valuable approach to improve the performance, security, and reliability of computer systems. We hope that this idea would influence the design and implementation of next-generation computer systems.

# Appendix A

# The complexity of reachability analysis

In this appendix, we discuss the complexity of the basic problem of determining reachability in a network given its data plane state.

The difficulty of determining reachability depends strongly on what functions we allow the data plane to perform. If network devices implement only IP-style longest prefix match forwarding on a destination address, it is fairly easy to show that reachability can be decided in polynomial time. However, if we augment the data plane with richer functions, the problem quickly becomes difficult. As we show below, packet filters make reachability NP-Complete; and of course, reachability is undecidable in the case of allowing arbitrary programs in the data plane.

It is useful to mention how this complexity relates to the approach of Xie et al. [123], whose reachability algorithm is essentially the same as ours, but written in terms of set union / intersection operations rather than SAT. As pointed out in [123], even with packet filters, the reachability algorithm terminates within $O(V^3)$ operations. However, this algorithm only calculates a formula representing reachability, and does not evaluate whether that formula is satisfiable. In [123], it was assumed that evaluating the formula (via set operations in the formulation of [123]) would be fast. This may be true in many instances, but in the general case deciding whether one vertex can reach another in the presence of packet filters is *not* in $O(V^3)$, unless P = NP. Thus, to handle the general case, the use of SAT or similar techniques is required since the problem is NP-complete. We choose to use an existing SAT solver to leverage optimizations for determining satisfiability.

We now describe in more detail how packet filters make reachability NP-Complete. The input to the reachability problem consists of a directed graph $G = (V, E)$, the boolean policy function $\mathcal{Q}(e, p)$ which returns true when packet $p$ can pass along edge $e$, and two vertices $s, t \in V$. The problem is to decide whether there exists a packet $p$ and an $s \rightsquigarrow t$ path in $G$, such that $\mathcal{Q}(e, p) = true$ for all edges $e$ along the path. (Note this problem definition does not allow packet transformations.) To complete the definition of the problem, we must specify what sort of packet filters the policy function $\mathcal{Q}$ can represent. We could allow the filter to be any boolean expression whose variables are the packet's fields. In this case, the problem can trivially encode arbitrary SAT instances by using a given SAT formula as the policy function along a single edge $s \to t$, with no other nodes or edges in the graph, with the SAT formula's variables being the packet's fields. Thus, that formulation of the reachability problem is NP-Complete.

One might wonder whether a simpler, more restricted definition of packet filters makes the problem easy. We now show that even when $\mathcal{Q}$ for each edge is a function of *a single bit* in the packet header, the problem is still NP-complete because the complexity can be encoded into the network topology.

**Property A.1.** *Deciding reachability in a network with single-bit packet filters is NP-Complete.*

*Proof.* Given a packet and a path through the network, since the length of the path must be $< |V|$, we can easily verify in polynomial time whether the packet will be delivered. Therefore the problem is in NP.

To show NP-hardness, suppose we are given an instance of a 3-SAT problem with $n$ binary variables $x_1, \ldots, x_n$ and $k$ clauses $C_1, \ldots, C_k$. Construct an instance of the reachability problem as follows. The packet will have $n$ one-bit fields corresponding to the $n$ variables $x_i$. We create $k+1$ nodes $v_0, v_1, \ldots, v_k$, and we let $s = v_0$ and $t = v_k$. For each clause $C_i$, we add three parallel edges $e_{i1}, e_{i2}, e_{i3}$ all spanning $v_{i-1} \to v_i$. If the first literal in clause $C_i$ is some variable $x_i$, then the policy function $\mathcal{Q}(e_{i1}, p) = \mathit{true}$ if and only if the $i$th bit of $p$ is 1; otherwise the first literal in $C_i$ is the negated variable $\overline{x}_i$, and we let $\mathcal{Q}(e_{i1}, p) = \mathit{true}$ if and only if the $i$th bit of $p$ is $0$. The policy functions for $e_{i2}$ and $e_{i3}$ are constructed similarly based on the second and third literals in $C_i$.

With the above construction a packet $p$ can flow from $v_{i-1}$ to $v_i$ if and only if $C_i$ evaluates to true under the assignment corresponding to $p$. Therefore, $p$ can flow from $s$ to $t$ if and only if all 3-SAT clauses are satisfied. Thus, since 3-SAT is NP-complete, reachability with single-bit packet filters is NP-complete. □

# References

[1] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, Hong Kong, China, Mar. 2004. 2, 30

[2] Alexa. `http://www.alexa.com/topsites`. 33, 48

[3] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, Cascais, Portugal, Oct. 2011. 4, 62

[4] A. M. Azab, P. Ning, and X. Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, Oct. 2011. 92

[5] F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot. The role of PASTA in network measurement. In *Proceedings of the ACM SIGCOMM 2006 conference*, Pisa, Italy, Sept. 2006. 31

[6] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards parallelizing the layout engine of Firefox. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, Berkeley, California, USA, June 2010. 3, 32, 35, 58

[7] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1999. 30

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, Santa Barbara, California, USA, Aug. 1996. 67

[9] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, Boston, Massachusetts, USA, Apr. 2009. 30

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, Dec. 1995. 63, 92, 94

[11] W. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15:1382–1396, 1989. 93

[12] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Amsterdam, Netherlands, Mar. 1999. 31

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, July 1970. 41

[14] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Austin, Texas, USA, Jan. 2011. 95

[15] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 2008. 95

[16] Broadcom Inc. Bcm28150 - 1080p 4g hspa+ smartphone processor, 2011. `http://www.broadcom.com/products/Cellular/3G-Baseband-Processors/BCM28150`. 3

[17] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, York, UK, Mar. 2009. 9, 20

[18] R. Bush and T. G. Griffin. Integrity for virtual private routed networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, Mar. 2003. 29

[19] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Computer Communication Review*, 40(2):12–20, Apr. 2010. 58

[20] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: a parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, 2013. 3

[21] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Big Sky, Montana, USA, Oct. 2009. 62

[22] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, Mar. 2000. 42

[23] S. Checkoway and H. Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, Texas, USA, Mar. 2013. 4, 75, 92

[24] B. X. Chen and N. Bilton. Et tu, Google? android apps can also secretly copy photos. `http://bits.blogs.nytimes.com/2012/03/01/android-photos`. 1, 4

[25] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural support for Programming Languages and Operating Systems*, Seattle, Washington, USA, Mar. 2008. 4, 92

[26] Cisco Systems Inc. Spanning tree protocol problems and related design considerations. http://www.cisco.com/en/US/tech/tk389/tk621/technologies_tech_note09186a00800951ac.shtml, August 2005. Document ID 10556. 2

[27] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, Aug. 2009. 61, 69, 93

[28] Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org. 6, 61

[29] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2006. 62, 92

[30] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, Washington, USA, Oct. 2007. 92

[31] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: improving reliability through operating system structure. In *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, San Diego, California, USA, Dec. 2008. 89

[32] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Mar. 2008. 61

[33] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, Snowbird, Utah, USA, June 2001. 92

[34] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Austin, Texas, USA, Jan. 2011. 91

[35] S. Dubey. AJAX performance measurement methodology for internet explorer 8 beta 2. CODE Magazine, Vol. 5 Issue 3, 2008. http://www.code-magazine.com/Article.aspx?quickid=0811102. 32, 33

[36] J. Duffy. BGP bug bites Juniper software. *Network World*, Dec. 2007. 2

[37] Eclipse Software Foundation. Jetty - servlet engine and http server. http://www.eclipse.org/jetty. 85

[38] M. Ettrich and O. Taylor. XEmbed protocol specification. http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html. 43

[39] J. Evers. Trio of Cisco flaws may threaten networks. *CNET News*, Jan. 2007. 2

[40] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, Mar. 2000. 8

[41] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, Boston, Massachusetts, USA, May 2005. 2, 30

[42] N. Feamster and J. Rexford. Network-wide prediction of BGP routes. *IEEE/ACM Transactions on Networking*, 15:253–266, 2007. 2

[43] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the 1979 National Computer Conference*, New York City, NY, June 1979. 60, 62, 93

[44] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proceedings of the ACM SIGCOMM 2004 conference*, Portland,Oregon,USA, Sept. 2004. 31

[45] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, Atlanta, Georgia, USA, Dec. 2010. 32, 35, 58

[46] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, Oakland, California, USA, Apr. 1982. 95

[47] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, Mar. 2003. 30

[48] Google Inc. Chrome multi-process architecture. `http://www.chromium.org/developers/design-documents/multi-process-architecture`. 43

[49] Google Inc. SPDY: an experimental protocol for a faster web, 2011. `http://www.chromium.org/spdy/spdy-whitepaper`. 58, 59

[50] Google V8 Team. `http://code.google.com/p/v8`. 34, 59

[51] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2008. 3

[52] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of IPSec and VPN security policies. In *Proceedings of the 13th IEEE International Conference on Network Protocols*, Boston, Massachusetts, USA, Nov. 2005. 29

[53] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, Dec. 1986. 36

[54] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, Texas, USA, Mar. 2013. 4, 75, 92

[55] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, UK, July 2005. `http://www.cs.ru.nl/H.Tews/Plos-2005/ecoop-plos-05-letter.pdf`. 93

[56] X. Hu and Z. M. Mao. Accurate real-time identification of IP prefix hijacking. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007. 31

[57] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, San Francisco, California, USA, June 2010. 33, 58

[58] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, Apr. 2007. 60, 62, 63, 92, 94

[59] IBM. Build server-side mashups with Geronimo and REST. `http://www.ibm.com/developerworks/opensource/library/os-ag-mashup-rest/index.html`. 42

[60] Intel. The all new 2010 Intel Core vPro processor family: Intelligence that adapts to your needs, 2010. `http://www.intel.com/Assets/PDF/whitepaper/311710.pdf`. 8

[61] C. Jiang. BigPipe: Pipelining web pages for high performance. `http://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919`, 06 2010. 42, 48, 59

[62] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *Proceedings of the 1st USENIX conference on Hot Topics in Parallelism*, Berkeley, California, Mar. 2009. 32, 33

[63] Juniper Networks, Inc. JUNOS: MPLS fast reroute solutions, network operations guide. `http://www.juniper.net/techpubs/en_US/junos10.0/information-products/topic-collections/nog-mpls-frr/frameset.html`, 2007. 8

[64] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997. 89

[65] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, 2012. 30

[66] G. Keizer. Google pulls 22 more malicious android apps from market. `http://www.computerworld.com/s/article/9222595/Google_pulls_22_more_malicious_Android_apps_from_Market`. 1, 4

[67] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, Lombard, IL, 2013. 30

[68] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992. 78

[69] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, USA, Oct. 2009. 60, 62, 64, 92, 93

[70] E. Kohler. HotCRP conference management software. `http://www.read.seas.harvard.edu/~kohler/hotcrp`, 2007. 85

[71] M. Krigsman. Amazon s3 web services down. bad, bad news for customers. `http://blogs.zdnet.com/projectfailures/?p=602`, 2008. 1

[72] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, Illinois, USA, Oct. 2011. 64

[73] M. Lasserre and V. Kompella. Virtual private LAN service (VPLS) using label distribution protocol (LDP) signaling. RFC 4762, Jan. 2007. 8

[74] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, USA, Mar. 2004. 18

[75] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chigago, Illinois, USA, June 2005. 95

[76] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, Dakar, Senegal, Apr. 2010. 61, 63

[77] F. Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation. In *Proceedings of the 12th Conference on Verification, Model Checking and Abstract Interpretation*, Austin, Teaxs, USA, Jan. 2011. 60

[78] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, Pennsylvania, USA, Jan. 2012. 90

[79] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 conference*, Pittsburgh, Pennsylvania, USA, Aug. 2002. 2

[80] H. Mai, C. Gao, X. Liu, X. Wang, and G. M. Voelker. Towards automatic inference of task hierarchies in complex systems. In *Proceedings of the 4th Workshop on Hot Topics in System Dependability*, San Diego, California, USA, Dec. 2008. 1, 2

[81] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference*, Toronto, Ontario, Canada, Aug. 2011. 1

[82] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, Texas, USA, Mar. 2013. 1

[83] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, Berkeley, CA, June 2012. 1

[84] Y. Mandelbaum, S. Lee, and D. Caldwell. Adaptive parsing of router configuration languages. In *Proceedings of the 2008 Internet Network Management Workshop*, Orlando, Florida, USA, Oct. 2008. 30

[85] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011. 92

[86] Z. M. Mao, D. Johnson, J. Rexford, J. Wang, and R. Katz. Scalable and accurate identification of AS-level forwarding paths. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, Hong Kong, China, Mar. 2004. 30

[87] M. Mayer. Google I/O '08 keynote, 2008. `http://www.youtube.com/watch?v=6x0cAzQ7PVs`. 3

[88] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2010. 4, 92

[89] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008. 8

[90] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, Raleigh, North Carolina, USA, Apr. 2010. 3, 32, 33, 35, 58

[91] Microsoft. Measuring browser performance, March 2009. `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=cd8932f3-b4be-4e0e-a73b-4a373d85146d`. 48

[92] A. Moshchuk, S. D. Gribble, and H. M. Levy. Flashproxy: transparently enabling rich web content via remote execution. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, Breckenridge, Colorado, USA, June 2008. 58

[93] Nagios. `http://www.nagios.org`. 31

[94] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM 1997 conference*, Cannes, France, Sept. 1997. 59

[95] Opera Software. `http://www.opera.com/mobile`. 42

[96] OProfile. `http://oprofile.sourceforge.net`. 34

[97] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). In *Proceedings of the 7th DoD/NBS Computer Security Initiative Conference*, Sept. 1984. 93

[98] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *Proceedings of ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation*, Seattle, Washington, USA, June 2013. 90, 91

[99] Quagga Routing Suite. http://www.quagga.net. 24

[100] Renesys. Longer is not always better. `http://www.renesys.com/blog/2009/02/longer-is-not-better.shtml`. 2

[101] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: enabling controlled networking. *SIGCOMM Computer Communication Review*, 33(1):65–70, Jan. 2003. 29

[102] Ruby-Prolog. `https://rubyforge.org/projects/ruby-prolog`. 20

[103] Samsung Inc. Samsung introduces high performance, low power dual cortex - a9 application processor for mobile devices, Sept. 2010. `http://www.samsung.com/global/business/semiconductor/newsView.do?news_id=1195`. 3

[104] E. Schurman and J. Brutlag. Performance related changes and their user impact. `http://velocityconf.com/velocity2009`. 3, 32

[105] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, Washington, USA, Oct. 2007. 4, 75, 91

[106] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charleston, South Carolina, USA, Dec. 1999. 92, 93

[107] F. Silveira, C. Diot, N. Taft, and R. Govindan. ASTUTE: detecting a different class of traffic anomalies. In *Proceedings of the ACM SIGCOMM 2010 conference*, New Delhi, India, Sept. 2010. 31

[108] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011. 92

[109] SkyFire Labs, Inc. `http://www.skyfire.com`. 42

[110] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proceedings of the ACM SIGCOMM 2002 conference*, Pittsburgh, Pennsylvania, USA, Aug. 2002. viii, 26, 29

[111] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, Paris, France, Apr. 2010. 92

[112] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, USA, Oct. 2003. 62, 92

[113] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010. 89, 92, 94

[114] The jQuery Project. `http://jquery.com`. 52

[115] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, Jan. 2007. 74

[116] P. Tune and D. Veitch. Towards optimal sampling for flow size estimation. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, Vouliagmeni, Greece, Oct. 2008. 31

[117] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM symposium on Operating Systems Principles*, Asheville, North Carolina, USA, Dec. 1993. 62

[118] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, Feb. 1980. 60, 62, 93

[119] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, Phoenix, Arizona, USA, Mar. 2011. 32

[120] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, Dec. 2002. 62

[121] J. Wu, Z. M. Mao, J. Rexford, and J. Wang. Finding a needle in a haystack: pinpointing significant bgp routing changes in an IP network. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, Boston, Massachusetts, USA, May 2005. 2

[122] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974. 92

[123] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, Miami, Florida, USA, Mar. 2005. 9, 11, 28, 96

[124] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. 31

[125] Yahoo! Best practices for speeding up your web site. `http://developer.yahoo.com/performance/rule.html`. 59

[126] Z. Yang. Every millisecond counts, 2009. `http://www.facebook.com/note.php?note_id=122869103919`. 3

[127] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *SIGCOMM Computer Communication Review*, 40(3):34–40, June 2010. 2, 24, 30

[128] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, USA, Oct. 2009. 85

[129] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, Pennsylvania, USA, Mar. 2010. 31

[130] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2006. 2, 30

[131] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical report, Columbia University, June 1998. CUCS-021-98. 69

[132] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation*, Seattle, Washington, USA, Nov. 2006. 92

[133] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011. 4, 92

[134] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th USENIX Conference on Operating systems Design and Implementation*, Seattle, Washington, USA, Nov. 2006. 92

[135] E. Zmijewski. Reckless driving on the Internet. `http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml`, February 2009. 2