DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

CREATING TRUSTED SYSTEMS IN UNTRUSTED ENVIRONMENTS

Citation
Ko, Ronny (Hajoon). 2021. CREATING TRUSTED SYSTEMS IN UNTRUSTED ENVIRONMENTS. Doctoral dissertation, Harvard University Graduate School of Arts and Sciences.

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37370238

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

HARVARD UNIVERSITY

Graduate School of Arts and Sciences

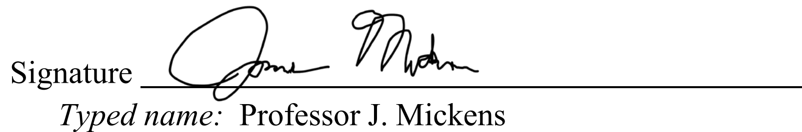
DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences
have examined a dissertation entitled:

"Creating Trusted Systems in Untrusted Environments"

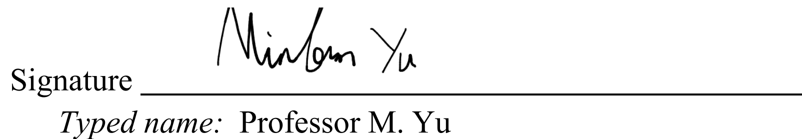
presented by: Ronny (Hajoon) Ko

Signature 

Typed name: Professor J. Mickens

Signature

Typed name: Professor H.T. Kung

Signature 

Typed name: Professor M. Yu

Signature

Typed name: Professor S. Chong

August 4, 2021

PHD THESIS

CREATING TRUSTED SYSTEMS IN UNTRUSTED ENVIRONMENTS

Ronny (Hajoon) Ko

Harvard University

hrko@g.harvard.edu

August 4th 2021

Dissertation Adviser: James Mickens

Ph.D Candidate: Ronny (Hajoon) Ko

Abstract

This dissertation illustrates how to improve the security and privacy of user data in modern Internet services. Three specific domains are examined: client-side IoT deployments, server-side application stacks, and middlebox acceleration proxies for HTTPS traffic. The dissertation highlights each domain's unique challenges, and proposes three distinct platforms for safeguarding user data: DeadBolt, Riverbed, and Oblique. DeadBolt makes IoT deployments more secure, quarantining IoT devices unless those devices are running up-to-date software or are protected by security middleware that interposes on the devices' network traffic. Riverbed leverages information flow control and a simple policy language to enforce user-defined privacy policies in legacy applications. Oblique uses symbolic execution to allow third-party analysis of HTTPS web content without revealing concrete values associated with sensitive user data like cookies.

Contents

1 Introduction **1**

1.1 Internet of Things 2

1.1.1 Problem 2

1.1.2 Proposed Solution: DeadBolt 3

1.2 Datacenters 4

1.2.1 Problem 4

1.2.2 Proposed Solution: Riverbed 5

1.3 Web Proxying 6

1.3.1 Problem 6

1.3.2 Proposed Solution: Oblique 6

2 DeadBolt: Securing IoT Deployment **8**

2.1 Motivation 8

2.2 Background 10

2.2.1 IoT Deployment Models 10

2.2.2 Remote Attestation Schemes 11

2.2.3 Control Flow Integrity 12

2.2.4 IoT Network Traffic Security 14

2.3 Design 15
2.3.1 Remote Attestation and Device Quarantine 16
2.3.2 Fast OS Patches Using Two VMs 18
2.3.3 Virtual Drivers 20
2.3.4 Control Flow Integrity 22
2.4 Implementation 22
2.5 Evaluation 23
2.5.1 Attestation Costs: Traditional Reboot 23
2.5.2 Fast VM-based Reboot 24
2.5.3 The Overhead of Virtual Drivers 25
2.5.4 Code Shuffling 27
2.5.5 Preventing Attacks 28
2.6 Conclusion 29

3 Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services 30

3.1 Motivation 30
3.2 Background 32
3.2.1 Explicit Labeling 32

3.2.2 Implicit Labeling . . . 33

3.2.3 Formal Verification . . . 34

3.3 Design . . . 34

3.3.1 Riverbed-amenable Services . . . 35

3.3.2 Expressing Policies . . . 35

3.3.3 Server Attestation . . . 39

3.3.4 Universes . . . 41

3.3.5 Taint Tracking . . . 44

3.4 Discussion . . . 47

3.5 Implementation . . . 50

3.6 Evaluation . . . 51

3.6.1 Attestation Overhead . . . 51

3.6.2 Case Studies . . . 52

3.6.3 PyPy Benchmarks . . . 54

3.6.4 Universe Overhead . . . 55

3.7 Conclusion . . . 58

4 Oblique: Accelerating Page Loads Using Symbolic Execution **59**

4.1 Motivation 59

4.2 Background 62

4.3 Design 64

4.3.1 Overview of Concolic Execution 65

4.3.2 Analyzing Client-side Behavior 66

4.3.3 Nondeterministic JavaScript Functions 71

4.3.4 Analyzing Server-side Behavior 73

4.3.5 The Workflow 74

4.3.6 Templating Engines 75

4.3.7 Security Analysis 77

4.4 Discussion 78

4.5 Implementation 79

4.6 Evaluation 80

4.6.1 Methodology 80

4.6.2 Client-only Analysis 81

4.6.3 Oblique in First-party Mode 87

4.7 Conclusion 89

5 Conclusion **90**

1 Introduction

The number of worldwide Internet users has reached 5 billion [1], with a typical individual spending 2.6 hours online per day [2]. As people increasingly use the Internet for various services, they increasingly share *private data* that must be safeguarded. Private data refers to pieces of information which belong to an individual or are closely related to an individual. Examples of private data include login passwords, credit card numbers, email messages, and photos. Given the sensitivity of such information, users expect Internet services to handle the data securely. For example, users expect services to protect the data from theft by hackers. Users also expect services to not share private data with third parties unless the user consents to the sharing.

Recognizing threats to data privacy, governments have enacted privacy laws, but unfortunately, those laws are insufficient to protect data privacy. This is because privacy laws do not guarantee that systems actually adhere to those laws; law enforcement can detect lack of compliance, but such detection may be too late if private data has already been compromised. Privacy laws also do not provide detailed technical guidance about how developers should build applications to comply with the laws. Thus, even well-intentioned developers may struggle to understand how to create applications which adequately protect user data.

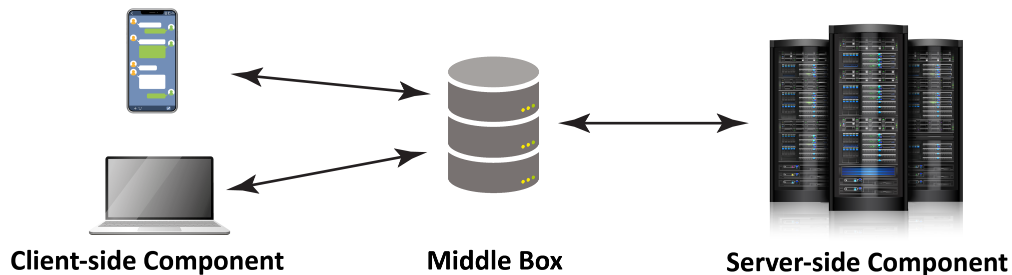


Figure 1: A general infrastructure of Internet services

This dissertation introduces new frameworks that protect sensitive user data in distributed systems. In order to design such frameworks, one must first understand how Internet services are currently built. At a high level, a service often consists of three components: a client-side

component, a server-side component, and middleboxes (see Figure 1). In this deployment model, a client (e.g., a smartphone or desktop) sends a request to a server (e.g., a web server), who in turn processes the request and sends a response to the client. A middlebox (e.g., a web proxy) intermediates the communication between a client and a server. A middlebox is a networking device which accesses the client/server traffic to improve performance, security, or another aspect of service behavior. Examples of middleboxes are firewalls, load balancers, and Internet proxies.

In my dissertation, I improve the security of data handling in all three components of a modern Internet service. I focus on data security in three specific domains. For the client-side IoT deployments, my goal is to update device software to secure state with a minimal system downtime and to transparently protect its real-time network traffic. For server-side application stacks, I aim to make it easy for developers to design web services that respect user-defined privacy policies. For middlebox acceleration proxies for HTTPS traffic, I envision to outsource page analysis to third parties without leaking client data. As explained below, each of these domains offers unique challenges to secure data handling.

1.1 Internet of Things

1.1.1 Problem

An Internet of Things (IoT) deployment contains sensors and actuators in a geographic environment like a home, a factory, a vehicle, or an entire city [3–9]. There are currently 11.6 billion IoT devices, and the number will reach 20 billion by 2025 [10]. IoT devices exchange data with a variety of network endpoints, such as user devices and cloud servers. Unfortunately, the network is often a vector for malicious inputs. These network-based attacks often exploit IoT vulnerabilities involving unpatched software, unencrypted network protocols, or memory corruptions. These exploit vectors are old ones, having plagued traditional network servers for decades. Unfortunately, IoT platforms often prioritize functionality over security, with devices riddled by known exploit

vectors. Another problem is that some IoT devices are too under-resourced to update their software or run heavyweight security mechanisms like anti-virus scanners. Although prior works proposed various IoT security techniques, they have certain limitations. Some research works focused on efficiently verifying the software stack of IoT devices [11, 12], but they failed to extend their goal to secure and fast software update for devices using insecure software stacks. Other research works explored how to secure real-time network traffic [13–15], but their framework do not allow developers to modularize security functions they design and develop into reusable parts, which leads to inefficient development and deployment for supporting heterogeneous IoT applications.

1.1.2 Proposed Solution: DeadBolt

DeadBolt is a security framework for IoT deployments. The goal of DeadBolt is to protect IoT devices from network-based attacks. DeadBolt places all IoT devices behind a custom DeadBolt access point (AP). The AP only allows devices to talk to the external network if the AP can verify that such network communication would be secure. To ensure this property, DeadBolt must distinguish between *heavyweight* devices and *lightweight* devices. A heavyweight device is one that has a TPM chip and updatable software (including updateable firmware); all other devices are lightweight. When a heavyweight IoT device initially connects to the network, the AP verifies the device's software stack via a remote attestation protocol, checking whether the device's software stack is up-to-date and trusted. If not, the AP pushes updates to the device, and refuses to let the device contact external hosts until the device proves via remote attestation that the device is running a secure stack. In order to minimize the externally-perceptible downtime of updating heavyweight devices, DeadBolt implements a fast software-patching technique. On each heavyweight device, DeadBolt runs a hypervisor that is responsible for managing the software on that device. A foreground, network-facing VM runs at all times; when a patch is required, DeadBolt launches a background VM, installs the patch on that VM, and then swaps the background VM to the foreground. To safeguard lightweight devices, the DeadBolt AP uses *virtual drivers* (i.e.,

security-enforcing network functions); these drivers sanitize or drop potentially-insecure network packets sent to or by a lightweight device. For example, a driver can drop cleartext outbound packets containing sensitive data. DeadBolt is the IoT security framework that enables fast and secure software update and makes development of real-time network security highly customizable and reusable. DeadBolt prevents realistic IoT attacks with around 10% average CPU overhead on heavyweight devices and less than 3% network overhead.

1.2 Datacenters

1.2.1 Problem

Cloud computing allows users to interact with their data on client-side devices, while pushing the bulk of the computation and storage to datacenter servers. This design reduces the computation overhead on client-side devices and guarantees better availability of services. This design also co-locates various users' data in the remote datacenter, making cross-user analytics easier. Unfortunately, when user data is pushed to remote machines, users surrender the ability to directly examine or control how that data is processed. For example, users cannot verify whether their data is shared with third-parties, or fed to first-party machine learning algorithms in a way that violates user preferences.

To protect user data privacy, the EU passed the General Data Protection Regulation (GDPR); other countries have passed similar laws [16–21]. Ideally, application-level developers could comply with laws like the GDPR by leveraging policy-enforcing code in the OS or in middleware libraries. However, no such general-purpose framework exists. Information flow control (IFC) is a well-known technique for restricting how sensitive data may spread throughout a system. Many prior works have proposed IFC-based techniques [22–31] to track data (and their derived data) which are to be secured or sanitized in applications. However, IFC techniques often require developers or users to reason about partial orderings between security labels. The lack of

widespread adoption for IFC systems suggests that label-based reasoning is too burdensome for most large-scale web services. Without framework support for enforcing data privacy policies, developers have to manually implement policy-enforcing code. However, data protection laws do not provide developers with low-level technical guidance about how user data should be managed. As a result, the developers of distinct applications are forced to reinvent this subtle, bug-prone data management code.

1.2.2 Proposed Solution: Riverbed

Riverbed is a framework for building privacy-protecting web services. A user Alice defines a high-level policy that restricts how server-side code stores her data, processes her data, and sends her data to remote servers. Riverbed uses server-side IFC to enforce the policy, but importantly, Riverbed is compatible with legacy applications that were not written with IFC in mind. The key insight is that users who choose the same data policy can all be handled by the same instance of a web service. Inside an instance, information flows involving sensitive data can be observed using simple taint tracking, with policy compliance checked before data would pass through sinks like the network or the disk. By creating a separate logical instance of the web service for each distinct policy, neither Riverbed nor developers have to worry about what would happen if a single web service had to compute over two different pieces of data that had two different policies. Thus, unlike prior IFC systems, Riverbed does not require humans to reason about tag lattices or manually label application state with tags. Riverbed uses containers, a lightweight virtualization mechanism, to cheaply spawn new instances of complex, multi-tier web services. Riverbed imposes an average CPU overhead of 10% for real applications.

1.3 Web Proxying

1.3.1 Problem

Mobile devices using cellular networks often suffer from high last-mile latency. Loading a web page is usually latency-bound, not bandwidth-bound. As a result, mobile page loads are often slower than their desktop counterparts. Prior research has introduced a variety of techniques to accelerate mobile page loads [32, 33, 33–37]. At a high level, these techniques analyze a page's content and identify the HTTP objects that are referenced by that page; the objects are then placed on the client before the actual page load, either by client-side prefetch or by server-side push. To reduce administrative overhead, many web service providers would like to outsource the analysis work to third parties. However, the analysis work requires access to cleartext page content; providing such access to third-parties would violate the secrecy of first-party HTTPS content.

1.3.2 Proposed Solution: Oblique

Oblique is a framework that accelerates mobile page loads by securely outsourcing page analysis to third parties. During the outsourced analysis, Oblique symbolically executes the client-side page load, enabling a third party to simulate a client browser's activity without actually seeing any sensitive client-side data (e.g., `navigator.userAgent`, `screen.width`, `document.cookie`). At the end of the symbolic analysis, Oblique outputs a list of objects which a client should prefetch. The list contains two types of URLs:

- **Static URLs** are referenced by the `src` attributes in HTML tags.
- **Dynamic URLs** represent objects that are fetched during the execution of Javascript code. Dynamic URLs may embed symbols corresponding to sensitive client-side state like cookies.

The static and dynamic URLs represent the objects which a page should prefetch.

Oblique rewrites a page's top-level HTML to contain a prefetching library. This library contains the prefetch list for a page, as well as JavaScript code that concretizes the dynamic URLs using a client's actual sensitive values. Once those URLs are concrete, the library fetches the associated objects, pulling them into the browser cache. As the browser parses the rest of the page's HTML content, the browser will try to download the referenced HTTP objects, which will hit in the pre-warmed browser cache. Oblique improves baseline page load times by 32%, outperforming prior state-of-the-art proxies while guaranteeing better privacy for outsourced page analysis. Oblique is the first web accelerator that simultaneously achieves both end-to-end security (between the client and the server) and performance (of fast page load).

2 DeadBolt: Securing IoT Deployment

2.1 Motivation

The Internet of Things (IoT) is a framework for pervasive computing where sensors, actuators and devices interact and share information. Examples of IoT include smart home systems, street lighting, traffic congestion detection and control, noise monitoring, citywide waste management, real time vehicle networks, and smart city frameworks [3–6].

Unfortunately, IoT systems have prioritized functionality over security. As a result, IoT applications have suffered from significant cyber attacks, some of which directly threatened human lives. For example, in November 2016, cybercriminals compromised the heating system of buildings in Lappenranta, Finland, disabling the heating controllers for the buildings in the middle of cold winter [38]. In 2017, the Mirai botnet attack turned thousands of IP cameras into botnets to launch DDoS attacks on DNS servers, taking down Etsy, GitHub, Netflix, Shopify, SoundCloud, Spotify, Twitter, and many other major websites [39]. In 2017, Brickerbot malware [40] infected hundreds of IoT devices per day and made them permanently unusable. A similar attack, *botnet barrage*, was launched within an university that compromised 5,000 IoT devices and turned them into DNS attackers [41]. In 2015, security researchers succeeded in remotely car-jacking a jeep running on the highway and took control of the dashboard functions, steering, and brakes [42]. There are many other examples of IoT attacks [43, 44].

Most IoT compromises are caused by network-based attacks. This chapter introduces a new IoT framework, called DeadBolt, to make IoT applications more resilient against these attacks. The primary goals of DeadBolt are to: 1) minimize the threat surface which IoT devices expose to a remote attacker, and 2) stop a potentially subverted device from subverting other devices in the IoT deployment. DeadBolt distinguishes *heavyweight* devices and *lightweight* devices. A heavyweight device is one that has a TPM chip and updatable software (including updateable firmware); all other

devices are lightweight. DeadBolt protects IoT deployments using the following techniques:

Reinforcing network traffic security with virtual drivers: Lightweight devices frequently do not support updateable firmware, or do support updateable firmware, but have been abandoned by their manufacturers. To protect such devices, DeadBolt forces them to communicate with the outside world via a DeadBolt access point (AP). The DeadBolt AP executes virtual drivers, which are network functions that monitor all network traffic exchanged between local IoT devices and remote endpoints. Each virtual driver has a specific task, such as traffic encryption or the dropping of malicious packets. For example, a TLS [45] virtual driver encrypts all traffic exchanged between a local IoT device and a remote endpoint. This is especially useful for lightweight IoT devices that do not have native TLS support [46]. Such devices can seamlessly communicate with a remote endpoint in the TLS protocol with the help of DeadBolt's TLS virtual driver. Within the DeadBolt AP, multiple virtual drivers can be stacked to form a customized packet-processing pipeline. For example, a TLS virtual driver can be stacked atop an intrusion detection driver, so that cleartext data can be scrutinized for malicious content before the data is encrypted.

Remote attestation with device quarantine: Many IoT devices get compromised because they do not regularly update their software [47]. The DeadBolt AP forces heavyweight devices to attest [48–50] before granting them access to the Internet. The DeadBolt AP allows a device to connect to the external Internet only if the device's software stack is trusted (§2.3.1). If the device's software is untrusted (e.g., because it contains unpatched code), the DeadBolt AP fetches the latest software updates and sends them to the device. The device installs them, reboots, re-associates with the AP, and retries remote attestation.

Dynamically protecting control flow integrity: Buffer overflows [51, 52] and return-oriented programming (ROP) attacks [53,54] are common approaches for subverting the control flow integrity of IoT programs. DeadBolt prevents such control-flow attacks by forcing IoT code to use stack

canaries [55], Address Space Layout Randomization (ASLR) [56], and no-execute bits for non-code pages [57]. DeadBolt also periodically re-randomizes a program's code layout [58], making it difficult for attackers to find and exploit ROP gadgets.

Fast patching: Software updates often causes service downtime especially when they involve updating the OS. DeadBolt reduces this downtime by installing patches on a background VM while a foreground VM handles interactions with external clients; once the background VM has finished updating itself, DeadBolt moves the VM to the foreground. DeadBolt uses CRIU snapshots [59] to transfer live application state between VMs so that session state involving external clients is preserved during the VM switch.

A DeadBolt prototype running on real IoT devices shows that strong IoT security can be enforced with minimal performance overhead. DeadBolt can stop realistic attacks without requiring costly hardware that would drive up the costs of IoT deployments; a DeadBolt AP can run on a $90 Minnowboard. Experiments show that, on heavyweight devices, VM switches occur in 0.98 seconds, which is much faster than a standard reboot cycle.

2.2 Background

2.2.1 IoT Deployment Models

There have been efforts from major corporations such as Intel, Huawei and Microsoft to standardize an IoT threat model [60]. Yet, there has been no concrete solution for IoT security which is comprehensive, generic, and open-source. Thus, IoT developers are forced to secure their applications using vendor-specific, proprietary solutions.

Many IoT deployments are based on a hub-spoke model [61–63], where a remote user or a cloud server issues commands to or retrieves data from the IoT hub; the IoT hub processes the

Name	CPU	RAM	Onboard TPM?	Price
Conga IA4	4x1.04 GHz x86 (64 bit)	4 GB	Yes	$455
Intel STK2m364CC	2x900 MHz x86 (64 bit)	4 GB	Yes	$259
Minnowboard Turbot	**2x1.46 GHz x86 (64 bit)**	**2 GB**	**Yes (firmware TPM)**	**$90**
Raspberry Pi 3	4x1.2 GHz ARMv8-A (64 bit)	1 GB	Installable	$35
C.H.I.P.	1 GHz ARMv7-R (32 bit)	512 MB	No	$9
LinkIt One	260 MHz ARM7EJ-S (32 bit)	4 MB	No	$59
Flora	16 MHz Atmel AVR (8 bit)	2.5K	No	$15
Arduino Uno	20 MHz Atmel AVR (8 bit)	2 KB	No	$25
Flir FX camera	N/A	N/A	No	$140
Garadget garage door opener	N/A	N/A	No	$89
Monnit temperature sensor	N/A	N/A	No	$49
Sabre motion sensor	N/A	N/A	No	$30

Table 1: Examples of popular IoT devices. Our experiment in § 2.5 used Minnowboard Turbot.

received commands, communicates with locally-connected IoT devices and collects results from them [61, 64, 65]. In other IoT deployment models, individual IoT devices directly communicate with the remote user or cloud via a local AP. DeadBolt requires that all IoT traffic goes through the DeadBolt AP. This design allows the AP to quarantine insecure devices and manage the traffic for devices that have network access.

2.2.2 Remote Attestation Schemes

Table 1 lists several popular IoT devices. In this chapter, we classify IoT devices as heavyweight and lightweight. Heavyweight devices can run traditional OSes such as Linux or Windows, and support onboard or installable TPM chips. Lightweight devices do not have such capabilities. In Table 1, the first four devices are heavyweight, and the rest are lightweight.

A TPM [48] is a tamper-resistant cryptographic processor. A TPM has 24 Platform Configuration Registers (`PCR`s) and a public/private RSA key pair. A TPM exposes three main interfaces:

- **`extend(index, value)`**: Sets `PCR[index] = hash(PCR[index] || value)`.

- **read(index)**: Returns the value of PCR[index].
- **quote(index, nonce)**: Signs (PCR[index] || nonce) with the TPM's RSA private key and returns it.

A TPM's tamper-proof hardware guarantees that PCR values can be modified only via extend; furthermore, the TPM's private key is never revealed to the outside world. These two security features enable a device to remotely attest a chronological list of the software that the device has launched. In particular, the binary value of each loaded file is extended to PCR[10]. At the end of a device's boot, PCR[10]'s hash value represents a cumulative summary of the device's software stack. If a malicious binary or configuration file is loaded, its hash value will be extended to PCR[10], which will eventually be detected when quote(index, nonce) is requested by a remote verifier.

As shown in Table 1, onboard TPM chips are common in mid-to-high-end IoT devices [66, 67]. An external TPM module [68, 69] can also be installed on the I/O port of TPM-less IoT device, like a Raspberry Pi.

TyTAN [11] and Sancus [12] use symmetric cryptography to enable remote attestation for lightweight devices that are too weak to support public-key cryptography. In TyTAN and Sancus, IoT administrators must install a shared key on an IoT device and the remote endpoint that wish to verify the device's software stack. DeadBolt is compatible with such devices. However, neither TyTAN nor Sancus deals with the practical issues that are addressed by DeadBolt's virtual drivers, device quarantine, and CFI enforcement.

2.2.3 Control Flow Integrity

Remotely attesting a device's software stack is insufficient to block attacks that dynamically tamper with a target program's memory state. For example, a program that does not properly check the

maximum length of input strings can illegally overwrite a function pointer or a return address inside its own stack; this allows an attacker to hijack the program's control flow and jump to an attacker-controlled memory buffer that contains malicious code [70, 71]. Simple buffer overflow attacks are stopped by defenses such as stack canaries, address space layout randomization (ASLR), and no-execute (NX) bits [51].

ROP attacks are a more advanced form of control flow hijacking [53, 54]. ROP attacks leverage the fact that a victim program naturally contains small instruction sequences, called *gadgets*, that can be chained together to execute malicious code. Chaining and executing multiple gadgets in a specific order can launch an attacker-controlled system call or a shell. For example, the attacker can overwrite the stack with the addresses of gadgets, and ensure that the return address of the exploited function will jump to the address of the first gadget. ROP attacks are not prevented by NX bits because the attacker is not injecting new code into the application. ROP attacks do typically require the attacker to derandomize the address space and read a canary value; however, there are a variety of techniques for doing so [72]. ROP attacks have been discovered for many applications that run on IoT devices [73], like Nginx [74] and OpenCV [75].

Various control flow integrity (CFI) schemes have been proposed to prevent ROP attacks [58, 76–79]. Their core idea is to insert a few instructions before or after each control flow instruction (e.g., `jump` or `call`). Those inserted instructions check whether each control flow shift is legitimate; if not, the program is terminated. DeadBolt protects against control flow attacks using Shuffler [58]. The Shuffler runtime not only instruments all control flow instructions as described above, but also randomizes the code's layout frequently (e.g., every 10 ms or 100 ms), to make it difficult for the attacker to find the memory location of gadgets within each shuffling period. Shuffler is attractive because it incurs a small performance overhead and can be directly applied to an executable without recompilation. While DeadBolt could use other CFI schemes besides Shuffler, DeadBolt's novelty lies in orchestrating an existing CFI scheme with security techniques like remote attestation and VM-based patching.

2.2.4 IoT Network Traffic Security

In an IoT network, a local wireless gateway is an ideal place to monitor potentially malicious network traffic, because all IoT devices have to access the Internet via the gateway. On Cisco routers, access control lists [13] can drop suspicious packets based on the contents of IP headers. Cisco's Snort program [14] can drop packets that contain malicious data. In-Hub Security Manager [15] allows an IoT gateway to rewrite traffic and apply extra security policies, such as transparently converting a weak password to a strong password. In DeadBolt, a virtual driver can provide equivalent services. However, DeadBolt provides additional functionality (e.g., fast patching, CFI enforcement) that cannot be provided by traffic inspection alone.

An IoT network traffic monitor can also be implemented within a smartphone. Hanguard [80] is a network monitor that runs on a smartphone and manages a whitelist of verified phone apps which may open a network connection to particular IoT devices. FlowFence [81] implements privilege separation within an IoT device, splitting each IoT application into privileged and unprivileged components and restricting how sensitive data flows to unprivileged components. DeadBolt is compatible with such schemes.

Like a DeadBolt AP, a HomeOS [82] gateway uses the driver abstraction to interact with individual IoT devices. For each low-level device protocol like Z-Wave or UPnP, HomeOS defines a connectivity driver that implements a standard interface for device discovery and device communication. Connectivity drivers are used by functionality drivers; each functionality driver implements the services provided by a broad class of device, e.g., a video camera or a temperature sensor. Other IoT frameworks use similar driver abstractions [61, 82]. DeadBolt is compatible with HomeOS-style driver decompositions. However, HomeOS and related frameworks lack most of DeadBolt's security mechanisms. For example, a HomeOS gateway forces a device to register before using the network, but HomeOS does not use remote attestation with a hardware root of trust to validate the device's code. HomeOS does not force devices to protect dynamic state using

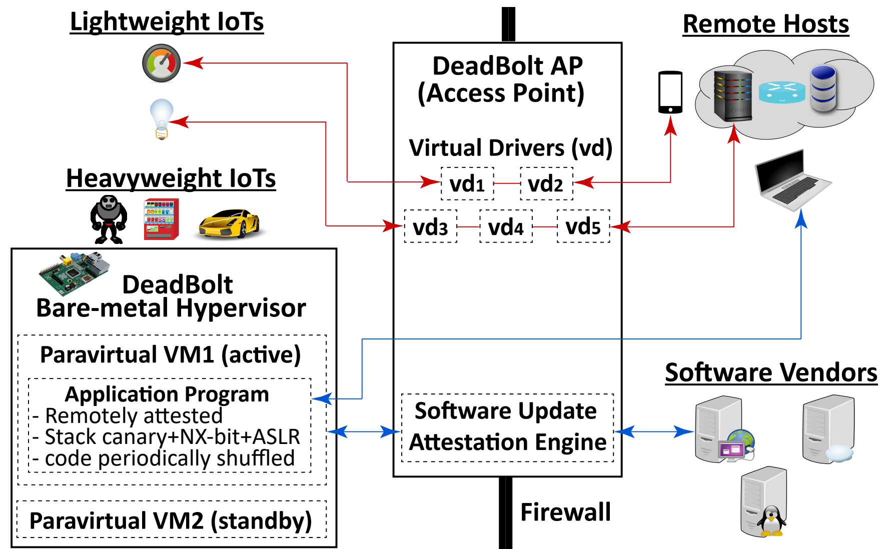


Figure 2: The DeadBolt architecture.

techniques like forced reboots. HomeOS also does not support virtual drivers to safely expose devices that would otherwise be vulnerable.

IoTSec [83] runs security-focused network functions inside of virtual machines. In contrast, DeadBolt runs such functions inside of namespaced processes; these processes consume fewer resources than full VMs, at the cost of weaker isolation guarantees.

2.3 Design

Figure 2 depicts DeadBolt's architecture, which is comprised of three components: a DeadBolt AP, heavyweight IoT devices, and lightweight IoT devices.

Each heavyweight device uses a hypervisor to launch application code within a VM (§2.3.2). Heavyweight devices attest the code in the hypervisor, the guest OS, and guest applications. Using background/foreground VMs, heavyweight devices can apply software updates quickly, with little application-visible downtime. The DeadBolt AP handles device quarantine and network firewalling, only allowing up-to-date, attested devices to exchange traffic with the outside world (§2.3.1). The AP's attestation daemon communicates with software vendors or third-party update

services using TLS. Lightweight devices (which cannot attest or update their software) are protected using virtual drivers that run on the AP (§2.3.3). Virtual drivers can provide additional security to heavyweight devices too.

2.3.1 Remote Attestation and Device Quarantine

By default, a heavyweight device has no ability to exchange network traffic with other endpoints. To gain network access, the device must remotely attest its software stack to the DeadBolt AP. Below, we provide an overview of how attestation works.

A device's software stack consists of its read-only firmware (e.g., CRTM [84]), its updatable firmware (e.g., UEFI), its bootloader, its OS, and its user-level applications. At boot time, the read-only firmware extends `PCR[0]` with the hash of the read-only firmware code. Then, the read-only firmware extends `PCR[0]` with the hash of the updatable firmware code, and executes that code. The updatable firmware runs, and extends `PCR[1-7]` with the hardware configuration data. When the updatable firmware completes execution, it reads the bootloader from storage into memory, extends `PCR[8]` with the hash of the bootloader code, and then jumps to the first bootloader instruction. This recursive extension of `PCR`s occurs for all of the software that is loaded by the machine: The bootloader extends the hash of the OS to `PCR[8-9]`. The OS extends the concatenation of `PCR[0-9]` into `PCR[10]`, and then extends hashes of the various kernel modules and user-level programs that the kernel launches to `PCR[10]`. At the end of the boot process, `PCR[10]` contains a hash value that represents the entire software stack that has run on the device since boot time. Whenever the device extends a new value to `PCR[10]`, it also appends a log entry to its software stack log file, which has the structure depicted in Figure 3. Each log entry has the file name and its hash value being extended to `PCR[10]`.

In order to gain network privileges, the device has to remotely attest its software stack to the DeadBolt AP. To do so, the AP sends a `nonce` to the device, which in turn asks its TPM to

Filename	SHA1(SHA1(file) || filename)
/bin/bash	883b6070003ee8f0618689a59603eb2d78475aca
/usr/sbin/httpd	6c3164a3fd220e184024d6290c18f7b9065eb08c
/usr/sbin/sshd	4f9bac3303b709afc6076987d03a5d91dd1c5be6
/bin/ls	56ac498205af05b2dcb767fd3456d5e798d47d84

Figure 3: Example of the software stack log file generated by a heavyweight device.

sign the current `PCR[10]` value using the `quote(index, nonce)` interface. The device then forwards the quote and the software stack log file to the AP. The AP verifies the quote's signature to ensure that it was generated by a trusted TPM private key. If so, the DeadBolt AP analyzes the log file to ensure that the extension of the hash values specified in the log file actually results in the `PCR[10]` value specified by the quote. Note that a corrupt or tampered log file does not compromise the security of remote attestation; since the TPM uses a cryptographically strong hash function, it is highly unlikely for two different software logs to possess the same `PCR[10]` value. After verifying the log file, the DeadBolt AP further analyses the software stack represented by the log file, to determine if that stack is trustworthy.

A DeadBolt AP can use a variety of different approaches for determining the trustworthiness of a device's software. The simplest approach is to have a list of whitelisted and blacklisted software. A more sophisticated approach would use a tool like Cobweb [85] to analyze richer contextual relationships between different software components. For example, the AP might check parent/child relationships between processes, or perform semantic analysis on the contents of important configuration files.

If the AP determines that a device's stack has become insecure, the AP notifies the device. If the device does not update itself during a grace period, the AP revokes the device's network access. To regain access, the device must query the AP to determine which blacklisted software should be deleted, or which required software should be installed, or which preexisting software should be updated. After deleting the blacklisted items, the device essentially treats the AP as a local software repository, downloading new or updated software from it. Afterwards, the device

IoT $\leftrightarrow$ DeadBolt AP	: Establish a WPA2 (EAP-TTLS) connection
IoT $\rightarrow$ DeadBolt AP	: Request an IP address
IoT $\leftarrow$ DeadBolt AP	: Assign an local-network-only IP address, IP_{IoT}
IoT $\rightarrow$ DeadBolt AP	: Request permission for IP_{IoT} to interact with external networks
IoT $\leftarrow$ DeadBolt AP	: $\{qc\}$, where qc is a quote challenge
IoT $\rightarrow$ DeadBolt AP	: $\{IMA, signedQ\}$, where:
	: IMA = IoT's IMA log
	: [Case IoT_H]: $signedQ = \textbf{QUOTE}(\{\texttt{PCR[10]} \mid qc \mid PubKey_{vTPM}\})$
	: [Case IoT_{VM}]: $signedQ = \textbf{QUOTE}(\{\texttt{vPCR[10]} \mid qc\})$
DeadBolt AP	: If $signedQ$ is valid, allow IP_{IoT} to interact with external networks

Table 2: Remote attestation when using VM-based state rejuvenation. IoT_H denotes a hypervisor; IoT_{VM} denotes a virtual machine. *vTPM* denotes a virtual TPM; `vPCR` denotes a virtual TPM's `PCR`.

must reboot and re-attest to gain network access.

2.3.2 Fast OS Patches Using Two VMs

Remote attestation forces devices to run updated software. If the AP determines that a device is *not* running the latest software, the AP refuses to grant the device general network access until the device downloads and installs the necessary patches. Unfortunately, installing updates often requires device reboots, which cause downtime. For example, updating a dynamic library used by the kernel's core processes requires a reboot after the update. Reboots reduce device availability. For example, an IoT device which acts as a web server or a streaming video source will have to interrupt live network connections to handle a reboot.

DeadBolt uses a fast reboot mechanism to minimize the visible downtime that is incurred by reboots. The fast reboot technique uses two virtual machines (VMs) running on a bare-metal hypervisor. In the steady state, a device's hypervisor only executes one VM. However, when that VM's state needs to be refreshed (e.g., to install a patch in the guest OS), the hypervisor launches a new VM in the background. As the foreground VM continues to execute and interact with external network clients, the background VM updates itself, e.g., by calling `apt-get` update to install new patches. Once the background VM has finished its boot, the foreground VM uses CRIU [59] to

snapshot user-level applications like web servers. The foreground VM writes the snapshots to a disk partition that is shared with the hypervisor. The hypervisor then kills the foreground VM, and allows the new VM to read the CRIU snapshots and resurrect the associated user-level applications. CRIU snapshots restore each application's stack, heap, memory mapping, file descriptors, and sockets, including their open network connections.

Note that, for a VM to access the network, the underlying hypervisor must also attest; then the VM must attest. DeadBolt leverages preexisting techniques [86] to securely perform this two-phase attestation. In particular, DeadBolt assumes each VM has a vTPM (virtual TPM), which is the VM's virtual hardware. Each vTPM's primordial root of trust is the physical TPM of the hypervisor. The DeadBolt hypervisor first extends the physical TPM with its loaded software files, the vTPM image, and the public key of vTPM. Each VM extends its vTPM with its software stack. This way, the trust chain extends from the physical TPM to the DeadBolt hypervisor, vTPMs, VMs and finally each VM's application programs. All of these are recorded to the physical TPM's `PCR[10]` and the vTPM's `vPCR[10]`, and are attested to the DeadBolt AP whenever a VM reboot occurs. The background VM and the hypervisor remotely attest to the DeadBolt AP while the foreground VM is running. Thus, the remote attestation delay is not synchronously paid during the VM switch.

DeadBolt's fast rebooting technique minimizes visible application downtime, but the snapshot mechanism can possibly propagate corrupted dynamic state across reboots. For example, a web server may be exploited via a buffer overflow triggered by a remote attacker; the corrupted server memory will persist across fast reboots. Remote attestation will not detect such attacks, because attestation detects *load-time* problems, not *run-time* problems. To recover from run-time corruptions of memory state that aren't prevented by CFI (2.3.4), applications must be restarted from scratch.

Source	Destination	Proto	Driver Stack
local.iot:ANY	remote.server:101	TCP	CAP-TLS
ANY:ANY	local.iot:2017	TCP	TLS-CAP-Snort
local.iot:ANY	remote.server:101	UDP	CAP-DTLS

Table 3: Example of the configuration state that the DeadBolt AP maintains for virtual drivers.

2.3.3 Virtual Drivers

The DeadBolt AP uses virtual drivers to manage the network traffic that is associated with IoT devices. Virtual drivers are particularly useful for securing traffic belonging to lightweight devices that cannot attest or update their software.

The DeadBolt prototype implements three virtual drivers:

- The **TLS/DTLS [45] driver** encrypts network traffic using the TLS/DTLS protocol (where DTLS is equivalent to TLS over UDP). This driver can be used to protect lightweight IoT devices that do not support network encryption protocols by encapsulating their traffic flow within an encrypted tunnel.
- The **ZIP driver** applies *gzip* compression to network traffic. This driver can reduce the bandwidth consumption of lightweight devices that do not support data compression.
- The **Snort [14] driver** monitors network data, looking for packet signatures that are indicative of attacks.

Note that if a device's traffic is mediated by the TLS and/or ZIP driver, the remote endpoint must also support the TLS or ZIP protocol. This improves security and/or network efficiency for both parties, but may require legacy remote endpoints to be modified to support TLS and/or ZIP.

Virtual driver policies (Table 3) can be configured by the AP's administrator or provided

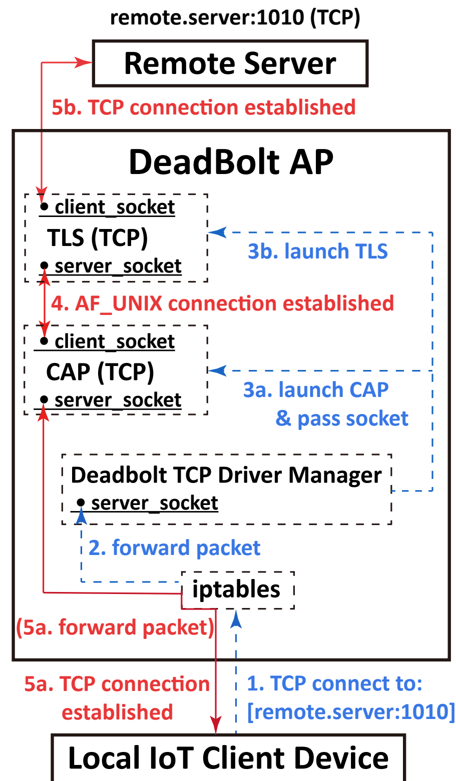


Figure 4: An example of a virtual driver stack comprised of two virtual drivers: TLS and ZIP. The virtual drivers mediate the traffic between a local IoT client device and a remote server. DeadBolt's TCP driver manager uses Netfilter [87] to map packets to the appropriate driver stack.

by a third party (e.g., Google, Ubuntu). The DeadBolt prototype provides template source code for a virtual driver, so that third party developers can easily design their own drivers. The DeadBolt AP runs each virtual driver stack in a separate container, so a compromised or poorly written virtual driver cannot corrupt other virtual driver stacks or other parts of the DeadBolt AP.

UDP driver stacks resemble TCP driver stacks; however, UDP is a connectionless protocol, so a UDP server has to serve multiple clients with a single socket. Due to this constraint, the AP uses *iptables* to redirect all outgoing UDP traffic to a local DeadBolt driver manager. The manager uses `recvmsg()` with the `SO_ORIGINAL_DST` flag to determine the source IP address and port for each UDP packet; the manager then forwards the packet to the appropriate driver stack.

2.3.4 Control Flow Integrity

In DeadBolt, remote attestation ensures that a device cannot access the network unless the device's software uses stack canaries, address space layout randomization (ASLR), and no-execute (NX) bit. These defense mechanisms effectively block simple control flow violation attacks, like buffer overflows. However, these defensive techniques cannot prevent advanced control flow attacks like ROP. To protect against such attacks, the DeadBolt AP forces attesting heavyweight devices to use advanced CFI techniques like the Shuffler [58] runtime. Shuffler slices the application code into many pieces and re-randomizes their locations frequently (e.g., every 50 ms). Shuffling makes it difficult for an attacker to successfully leverage a memory disclosure attack to launch a ROP exploit.

DeadBolt could employ other CFI techniques such as CCFI [88] or COTS [76]. DeadBolt's novelty lies not in the CFI technique itself, but in its integration of CFI techniques with a broader security infrastructure. Whenever a heavyweight device does fast reboot, its foreground VM creates snapshots of its application programs; each application will contain a copy of the Shuffler runtime in its address space. After a VM switch, the background VM restores the snapshots and the application programs resume with the Shuffler runtime automatically restarting. When CRIU restores a process's snapshot, CRIU reads the original (unshuffled) binary to extract the program code and various pieces of ELF metadata such as the program header and section headers. Shuffer compares the on-disk ELF metadata with the metadata in the snapshot. CRIU resumes the snapshot only if both sets of information match each other. Shuffling does not change the hash value that attestation will assign to a binary, because the IoT device attests the original binary, not snapshots.

2.4 Implementation

DeadBolt heavyweight devices ran Xen 4.10-unstable [89], patched to support vTPMs; inside a virtual machine, the guest OS was Ubuntu Server 17.04. The DeadBolt AP was a Minnowboard Turbot [90], and used HostAP [91], WPA Supplicant [92], and FreeRadius [93] to act as a WPA2

Stage	Hypervisor	VM
WPA2 connection	1878 ms	2051 ms
IP address registration	172 ms	197 ms
Attested Software Update	854 ms	641 ms
Total	**2904 ms**	**2889 ms**

Table 4: Network access latencies after a physical hardware reboot is completed. The WPA2 connection used EAP-TTLS.

access point.

Both sides of DeadBolt's remote attestation protocol were built using the IBM TSS library [94] and the IBM ACS library [95]. On the AP, we modified 10 lines of source code in DNSMasq [96] to integrate remote attestation with standard AP functionality involving DHCP and DNS. Heavyweight IoT devices used TPM-enabled GRUB2 [97] to measure the kernel image and the boot-time ramdisk. Afterwards, devices used Linux's Integrity Measurement Architecture [98] kernel module to extend `PCR[10]`. Devices used Shuffler [58] to periodically rerandomize a process's code offsets, and CRIU [59] to snapshot in-memory application state.

2.5 Evaluation

This section demonstrates that Deadbolt imposes minimal performance overheads while preventing realistic attacks.

2.5.1 Attestation Costs: Traditional Reboot

Table 4 shows the network access latencies for a heavyweight device (a Minnowboard Turbot) when it connected to the DeadBolt AP after a hard reboot of the physical hardware. The device first created a WPA2 (link layer) connection with the DeadBolt AP, and then an IPv4 (network layer) connection. The device then attested its software stack, which took 854ms for the DeadBolt hypervisor and 641ms for the VM which ran application code. In this experiment, we assumed that

Stage	Delay
VM_1: Snapshot the NGINX server	0.37 s
VM_1: Disable primary NIC, signal VM_2	0.12 s
VM_2: Enable the primary NIC	0.39 s
VM_2: Restore the NGINX snapshot	0.10 s
Total	**0.98 s**

Table 5: The application downtime induced by VM-based software patching.

NGINX Throughput (2 worker processes)	
Steady state	**As a background VM loads**
1843 requests/sec	1614 requests/sec

`sysbench` CPU (2 worker processes)	
Steady state	**As a background VM loads**
107.6 secs	135.0 secs

`sysbench` Random access I/O (2 worker processes)	
Steady state	**As a background VM loads**
Reads: 117.2 KB/s	Reads: 73.7 KB/s
Writes: 78.1 KB/s	Writes: 52.5 KB/s

Table 6: Performance slowdowns for a foreground VM as a background VM is launched. The IoT device was a Minnowboard Turbot. We used 2 application processes to ensure that both cores of the dual-core Minnowboard were contended by the foreground VM and the background VM.

all of the device's software was up-to-date; therefore, the DeadBolt AP did not need to fetch and send updated software packages to the heavyweight device.

2.5.2 Fast VM-based Reboot

Table 5 shows the VM switch delay incurred when a heavyweight device (a Minnowboard Turbot) ran an NGINX web server. For each VM switch, the foreground VM saved the web server's snapshot in the shared data partition, turned off its NIC, and then rebooted. Then, the background VM turned on its NIC and restored the web server's snapshot from the shared data partition. The VM switch delay was 0.98s.

The attestation latencies in Table 4 do not add to visible reboot-caused downtime, because

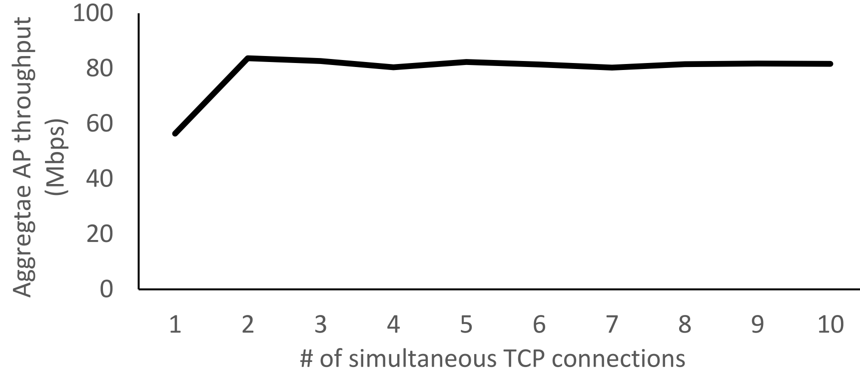


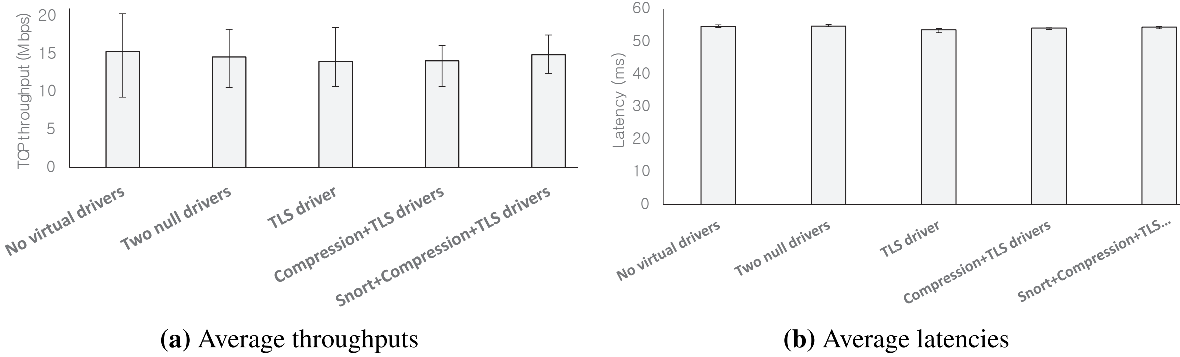
Figure 5: DeadBolt's virtual driver overhead as a function of the number of concurrent `iperf` flows.

the background VM reboots and attests in the background while the foreground VM is still running. Thus, the client-visible reboot downtime is just the delay caused by the VM switch (0.98s), as shown in Table 5.

Table 6 measures the foreground VM's performance slowdown when a background VM is rebooting. This slowdown occurs because the two VMs contend for the same physical resources. The experiment measured three types of overheads: CPU (sysbench [99]), network I/O (Apache Benchmark [100]), and disk I/O (sysbench). The background VM took 52 seconds to fully reboot. During this period, the foreground VM's CPU slowdown was 20.2%, the network I/O slowdown was 12.4%, and the disk I/O slowdown was 34%. After the reboot was complete, the NGINX server in the foreground VM performed similarly to a web server running on bare metal.

2.5.3 The Overhead of Virtual Drivers

Virtual drivers impose a per-flow computational overhead. To measure the impact on flow throughput and latency, we ran `iperf3` and `paping` to generate flows between a Raspberry Pi 3 client and an LG laptop; the DeadBolt AP was a Minnowboard Turbot. We used `netem` [101] to simulate a 50 ms RTT between the network endpoints. Across the various driver stacks that we



(a) Average throughputs **(b)** Average latencies

Figure 6: The average throughput and latency of various virtual driver stacks. In this experiment, we used `netem` [101] to simulate a wide-area network latency of 50ms. A NULL driver is an empty driver that simply forwards traffic to the next driver.

TCP video streaming throughput		
No driver	**TLS+CAP driver**	**TLS+CAP driver + Shuffler**
1.1Mbit/s	1.2Mbit/s	1.1Mbit/s

UDP command latency		
No driver	**TLS+CAP driver**	**TLS+CAP driver + Shuffler**
60.6 ms	61.8 ms	62.1 ms

Table 7: DeadBolt's overhead for an AR Drone 2.0.

tested, the throughput overhead was approximately 2%, whereas the latency overhead was roughly 1%.

We also examined how virtual drivers impact the aggregate throughput of a DeadBolt AP. We measured the overall AP throughput as we varied the number of concurrent TCP flows from 1 to 10. As Figure 5 shows, the virtual driver stacks added negligible throughput overhead. Note that the network throughput increased between 1 and 2 TCP connections. This happened because, , with only a single TCP connection, the connection reduced its window size whenever packet loss occurred between the local IoT device and the remote `iperf3` server and thereby ended up underutilizing the available network I/O. In contrast, when multiple TCP connections concurrently ran on the AP, the occurrence of packet loss on one connection would create available bandwidth for other connections to use, ensuring that the device could leverage the maximum network I/O throughput supported by the AP.

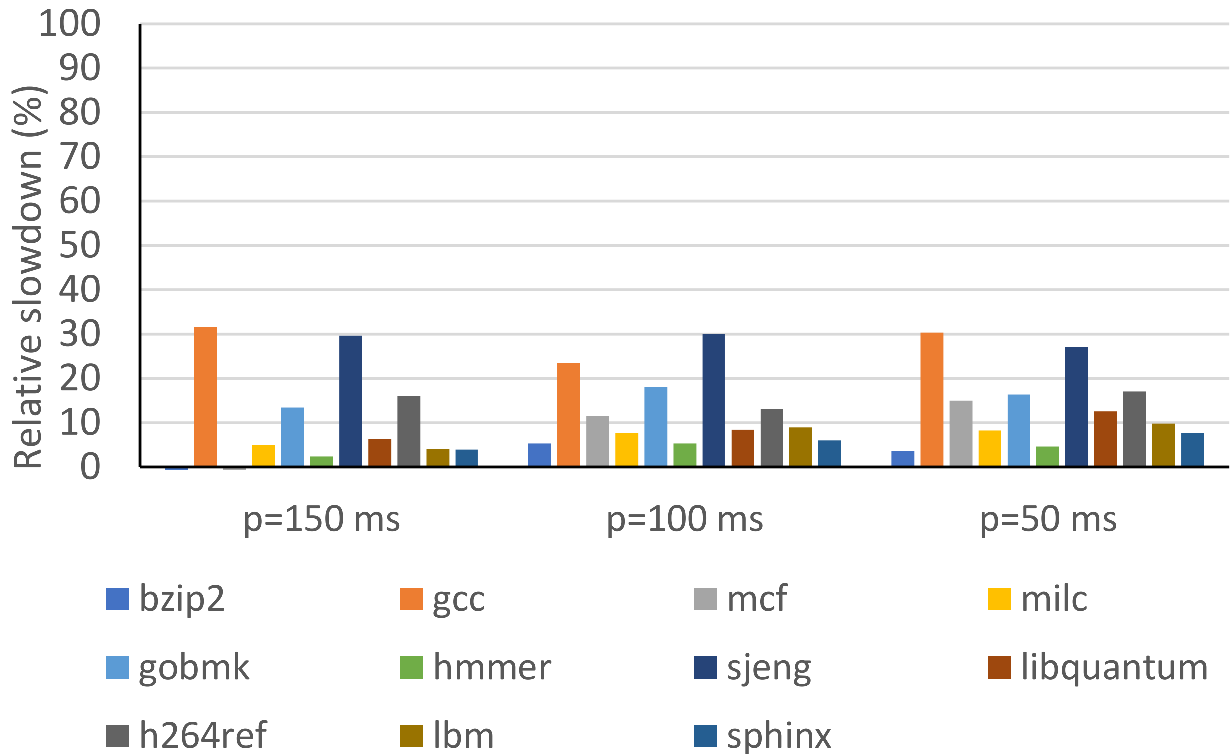


Figure 7: Overheads of code shuffling on a Minnowboard.

2.5.4 Code Shuffling

Code shuffling makes it hard for attackers to hijack the control flow of running programs. However, code shuffling imposes a periodic computational overhead. Figure 7 shows a Minnowboard's shuffling overheads for the CPU 2006 benchmark [102]. The overheads ranged between 2.3% and 31.5%; benchmarks with more I/O activity suffered from lower shuffling penalties, because the computational delays of shuffling were partially overlapped with I/O wait times. Realistic IoT applications typically require non-trivial amounts of I/O. For example, Table 7 depicts the shuffling overhead induced while the Minnowboard processed an AR Drone 2.0's UDP command packets and TCP video streams. The TCP video throughput decreased by 9.1%, whereas the UDP control traffic latency increased by only 0.01%. The TCP throughput suffered more because its video streaming task was more CPU-intensive than sending small UDP command packets.

2.5.5 Preventing Attacks

DeadBolt can stop many kinds of real-world attacks on IoT deployments. For example, IoT devices often run web servers like NGINX. A recent version of NGINX was vulnerable to an ROP attack [74]. When a DeadBolt-enabled device ran this vulnerable version of NGINX program, the attack failed due to Shuffler-based code randomization. Also note that using remote attestation and device quarantine, a DeadBolt AP could isolate a vulnerable IoT device until the device updated its NGINX installation.

The Shodan IoT scanner [103] probes a network, looking for IoT devices and trying to collect their configuration information; such information can later be exploited by remote attackers. DeadBolt's firewall can detect and drop incoming packets from the IP addresses of Shodan servers.

The *Cold-in-Finland* attack [38] was a DDoS attack that sent reboot commands to a building's smart heating system. A DeadBolt AP could drop reboot packets from unknown sources. More interestingly, a virtual driver can be used to force authentication of the endpoint who is sending reboot commands.

The *Mirai Botnet* [39] and *Brickerbot* [40] attacks targeted IoT devices that hosted telnet servers. *Mirai Botnet* and *Brickerbot* tried to log into vulnerable devices using a set of factory-default passwords. DeadBolt's NIDS driver could prevent this attack by looking for network traffic belonging to `telnet` logins. The NIDS driver could drop all such traffic, or only allow it if both endpoints reside within the local network.

DeadBolt implements techniques like virtual drivers and device quarantine using special AP software. Many IoT deployments already place devices behind an AP, meaning that DeadBolt's software can run on a preexisting gateway. However, if an IoT deployment directly exposes devices to the Internet, then the deployment operator must purchase and install a new device to serve as a DeadBolt AP. We believe that adding such a security monitor using (say) a $90 Minnowboard

(or a $40 Raspberry Pi 2 featuring an equivalent hardware spec) is financially prudent, given the potentially devastating costs associated with an exploited IoT deployment.

2.6 Conclusion

DeadBolt is an high-performance and cost-effective system that addresses critical security issues affecting IoT deployment. Using attestation-based quarantine and VM-based fast patching, DeadBolt ensures that device software is up-to-date before those devices communicate with the external world. Using ASLR, DEP, stack canaries, and code shuffling, DeadBolt thwarts control flow attacks. DeadBolt protects the IoT applications' underlying OS from being corrupted or buffer-overflowed by applying VM-based periodic instant reboot. DeadBolt protects the network traffic belonging to both heavyweight and lightweight devices by applying stackable virtual drivers. In aggregate, these mechanisms significantly improve the security of IoT deployments.

3 Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services

3.1 Motivation

Many of today's client-side applications interact with remote datacenters which process and store user data. This design has several benefits: it reduces the computation overhead on individual user devices, provides better availability of services, and co-locates various users' data in the remote datacenter to make cross-user analytics easier. However, once users send their data to remote servers, those users cede direct control over how their data is processed. For example, users cannot verify if their data was used in ways that violate their preference settings. As another example, users cannot tell whether their data is exported to untrusted third-parties.

To protect the privacy of data manipulated by Internet services, many nations have passed laws such as the General Data Protection Regulation (GDPR) [16–21]. These laws require service providers to give users the right to decide how their private data (e.g., shipping addresses, click histories, payment information) are processed by a service. For example, the GDPR states that *(i)* users must give consent for their data to be accessed; *(ii)* users must have the ability to know how their data is used; *(iii)* a user must have the right to request her data to be deleted; (iv) a service provider must implement "appropriate" security measures for data-handling pipelines. However, regulations like the GDPR lack corresponding system-level definitions and enforcement mechanisms. This deficit of technical guidance makes it hard for developers to implement services that comply with such laws.

Ideally, developers could use policy-enforcing middleware frameworks that ease compliance with data privacy laws. Unfortunately, no such general-purpose frameworks exist. In the research community, information flow control (IFC) [104, 105] has been used to restrict how sensitive data may flow throughout a system. IFC assigns labels to variables or OS-level objects

(e.g, processes, files, and pipes) and defines partial-ordering policies over label lattices to prevent inappropriate data flows. However, such label-based IFC is not a good fit for modern, large-scale web services, because reasoning about label hierarchies in complex systems is burdensome for both the service clients and developers. Indeed, there have been no large-scale commodity services that employ IFC to manage user data. As a consequence, complying with laws like the GDPR requires developers to hand-craft bug-prone data management code.

To address this problem, this chapter proposes Riverbed, a web service framework for systematically protecting client data privacy. The Riverbed framework is comprised of a client device (e.g., web browser), a web server, and a transparent Riverbed proxy between them. The Riverbed proxy tags each user request with a user-defined policy before sending it to the web server; users define policies using a simple language which defines restrictions on how data can be persistently stored, aggregated with the data of other users, or sent to third-party network servers. On the server-side, the Riverbed runtime uses IFC to enforce user data policies, placing all data with compatible policies into the same universe (i.e., the same isolated instance of the full web service). The universe technique enables Riverbed to transparently run unmodified application code, because there is no possibility that two pieces of information with conflicting policies will ever been computed upon in the same universe. Thus, Riverbed avoids the need for developers or users to reason about complex label hierarchies. Before a Riverbed proxy shares data with a server, the proxy forces the server to remotely attest that the server uses Riverbed's IFC runtime.

To demonstrate Riverbed's practicality, several real-world applications (MiniTwit [106], Ionic Backup [107], Thrifty P2P [108]) have been ported to Riverbed. Experiments show that Riverbed enforces realistic policies with worst-case user-perceived latency overheads of 10%.

3.2 Background

In this section, we compare Riverbed to representative instances of prior IFC systems. At a high level, Riverbed's innovation is the leveraging of universes and human-understandable, user-defined policies to enforce data flow constraints in IFC-unaware programs. Riverbed enforces these constraints without requiring developers to add security annotations to source code.

3.2.1 Explicit Labeling

In a classic IFC system, developers explicitly label program state, and construct a lattice which defines the ways in which differently-labeled state can interact. Roughly speaking, a program is composed of assignment statements; the IFC system only allows a particular assignment if all of the policies involving righthand objects are compatible with the policies of the lefthand side.

IFC-visible assignments can be defined at various levels of granularity. For example, Jif [22], Fabric [23], and similar frameworks [24–27] modify the compiler and runtime for a managed language, tracking information flow at the granularity of individual program variables. In contrast, frameworks like Thoth [28], Flume [29], Camflow [30], and DStar [31] modify the OS, associating labels with processes, IO channels, and OS-visible objects like files. Taint can be tracked at even high levels of abstraction, e.g., at the granularity of inputs and outputs to MapReduce tasks [109].

All of these approaches require developers to reason about a complex security lattice which captures relationships between a large number of privileges and privilege-using entities like users and groups. Porting a complex legacy application to such a framework would be prohibitively expensive, and to the best of our knowledge, there is no large-scale, deployed system that was written from scratch using IFC with explicit labeling. Developer-specified labels are also a poor fit for our problem domain of *user-specified* access policies.

Tracking data flows at too-high levels of abstraction can introduce problems of overtainting—to avoid false negatives, systems must often use pessimistic assumptions about how outputs should be tainted. For different reasons, overtainting is also a challenge for ISA-level taint tracking [110, 111], For example, if taint is accidentally assigned to `%ebp` or `%esp`, then taint will rapidly propagate throughout the system, yielding many false positives [112]. To avoid these problems, Riverbed taints at the managed runtime level, a level which does not expose raw pointers, and defines data types with less ambiguous tainting semantics.

In Jeeves [113, 114], a developer explicitly associates each sensitive data object with a high-confidentiality value, a low-confidentiality value, and a policy which describes the contexts in which a particular value should be exposed. An object's value is symbolic until the object is passed to an output sink, at which point Jeeves uses the context of the sink to assign a concrete value to the object. Riverbed avoids the need for developers to label objects with policies or concrete values with different fidelities; via the universe mechanism, Riverbed applications always compute on high-fidelity data while satisfying user-defined constraints on data propagation.

3.2.2 Implicit Labeling

Some IFC systems use predefined taint sources and IFC policies. For example, TaintDroid [115] uses a modified JVM to track information flows in Android applications. TaintDroid predefines a group of sensors and databases that generate sensitive data; examples of these sources include a smartphone's GPS unit and SMS database. The only sink of interest is the network, because TaintDroid's only goal is to prevent sensitive information from leaking via the network. Because TaintDroid uses a fixed, application-agnostic set of IFC rules, TaintDroid works on unmodified applications. Riverbed also works on unmodified applications. However, TaintDroid operates on a single-user device, whereas Riverbed targets a web service that has many users, each of whom may have unique preferences for how their data should be used. Thus, Riverbed requires users (but not developers or applications) to explicitly define information flow policies. Riverbed also requires

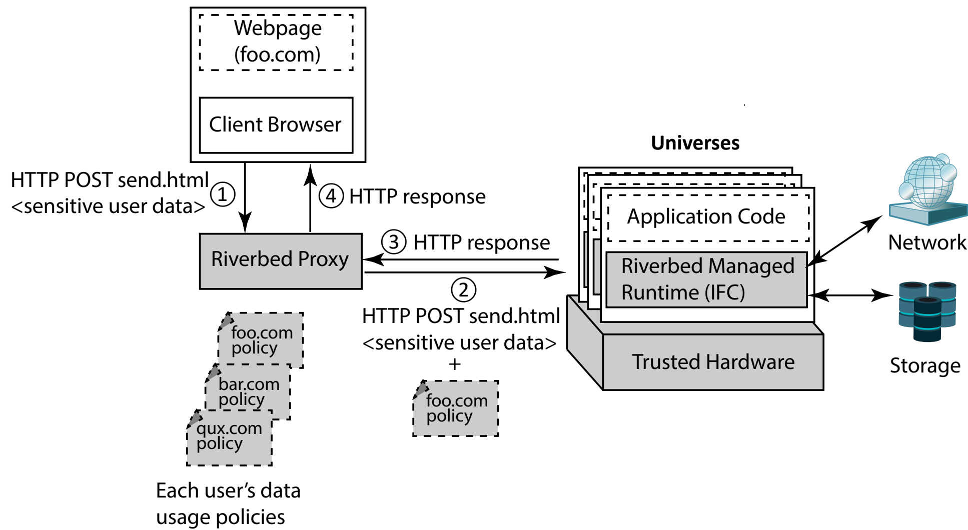


Figure 8: Riverbed's architecture. The user's client device is on the left, and the web service is on the right. Unmodified components are white; modified or new components are grey.

the universe mechanism (§3.3.4) to prevent the mingling of data from users with incompatible flow policies.

3.2.3 Formal Verification

IronClad [116] servers, like Riverbed servers, use remote attestation to inform clients about the server-side software stack. In Ironclad, server-side code is written in Dafny [117], a language that is amenable to static verification of functional correctness. Nothing prevents Riverbed from executing formally-verified programs; however, Riverbed's emphasis on running complex code in arbitrary managed languages means that Riverbed is generally unable to provide formal assurances about server-side code.

3.3 Design

Figure 8 provides a high-level overview of the Riverbed architecture. In this section, we provide more details on how users specify their policies, and how Riverbed enforces those polices on the

server-side.

3.3.1 Riverbed-amenable Services

Riverbed is best-suited for certain types of web services.

- **Services with per-user silos for application state, and no cross-user sharing:** Examples include back-up services like Ionic [107], and private note-taking apps like Turtl [118]. Riverbed prevents information leakage between per-user silos (although an individual silo may span multiple server-side hostnames and cloud providers).
- **Services that silo user data according to explicitly-defined group affinities:** For example, a social networking site can create a universe for the state belonging to a corporation's private group. The corporation's users map to the same Riverbed user (§3.3.2), with no data flows between different corporations. Financial analysis sites and email services can use this decomposition to isolate data belonging to a particular business or social group.
- **Services which aggregate unaffiliated users by shared polices:** For example, in a news site, users can define policies that impact whether the site may aggregate user data for targeted advertising. Riverbed places users with equivalent policies into the same universe, ensuring that the site respects each user's preferences.

Child policies (§3.3.2) can whitelist communication between server-side endpoints with otherwise incompatible policies. However, such whitelisting is easier when the server-side application consists of small, well-defined components, so that whitelisting individual components has well-understood security implications.

3.3.2 Expressing Policies

```
USER-ID: ALICE
AGGREGATION: False
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
ALLOW-TO-NETWORK: y.com
TRUSTED-SERVER-STACK: {
    83145c082bbf608989f05e85c3c211f83,
    c8cd7ac93cab2b94f65a5b2de5709767f,
        ...
    590f01d8d18b1141988ee1975b3ce3b30
}
```

Figure 9: An example of a Riverbed policy. For simplicity, we elide graph-based contextual attestation predicates (§3.3.3).

Figure 9 provides an example of a Riverbed policy. A policy consists of several parts, as described below.

The `USER-ID` field describes the owner of the policy. User ids only need to be unique within the context of a particular web service. Riverbed is agnostic about the mechanism that a service uses to authenticate users and log them into the service. However, Riverbed's server-side reverse proxy must know who owns the policy that is associated with each user request, so that the proxy can forward the request to the appropriate universe (§3.3.4).

Since Riverbed is agnostic about a service's login mechanism, a `USER-ID` field could actually be bound to a group of users. In this scenario, the users in the group would have different service-specific usernames, but share the same `USER-ID` field in their Riverbed policies. From Riverbed's perspective, the sensitive data of each individual user would all belong to a single logical Riverbed user.

The `AGGREGATION` flag specifies whether a user's data can be involved in server computations that include the data of other users. For example, suppose that a server wishes to add two numbers, each of which was derived from the data of a different user. If both users allow aggregation, Riverbed can execute the addition in the same universe. If one or both users disallow aggregation, then Riverbed must create separate universes for the two users. The `AGGREGATION` field specifies

a yes/no policy—either arbitrary aggregation is allowed, or all aggregation is disallowed.

The binary `PERSISTENT-STORAGE` flag indicates whether server-side code can write a user's data to persistent storage. If so, the user expects that when the data is read again by the server-side application, the application will treat the data as tainted. A Riverbed managed runtime terminates applications that try to write tainted data to persistent storage, but lack the appropriate permissions.

A policy can optionally include an email address that belongs to the policy owner. If a Riverbed managed runtime must terminate policy-violating code, Riverbed can email the policy owner, informing the user about the thwarted policy breach. The user can then complain to the service operator, or take another corrective action.

The `ALLOW-TO-NETWORK` field is optional, and allows a user to whitelist network endpoints to which user data may flow. Endpoints are represented by hostnames; each whitelisted hostname is expected to have a valid X.509 certificate, e.g., as used by HTTPS. Before a Riverbed managed runtime allows tainted data to externalize via a socket, the runtime will check whether the remote endpoint is whitelisted by the tainted data's policy. If so, the runtime forces the remote endpoint to attest its software stack. If that stack is whitelisted by the policy, the runtime allows the transfer to complete. Otherwise, the runtime terminates the application. Note that Riverbed allows untainted data to be sent to arbitrary remote servers.

The final item in a policy is typically one or more `TRUSTED-SERVER-STACK` entries. Each trusted stack is represented by a list of hash values; see Section 3.3.3 for more details about how these hash values are generated by servers, and later consumed by the attestation protocol.

As discussed in Sections 3.3.3 and 3.4, a client-side proxy leverages attestation to validate the server-side software stack up to, but not including, the application-defined managed code. Once the proxy determines that Riverbed's taint-tracking managed runtime is executing on the server, the proxy will trust the runtime to enforce the policies described earlier in this section. However, the

policies from earlier in this section only enable aggregation at a binary granularity (i.e., "allowed" or "disallowed"); a universe which disallows aggregation can never permit data to flow to a universe which *does* allow aggregation. This restriction prevents several useful types of selective aggregation. For example, two email servers in separate no-aggregate universes could ideally send emails to a trusted spam filter application which trains across all inboxes, and then returns a filter to each universe. To allow such aggregation by explicitly trusted components, Riverbed policies can decorate an `ALLOW-TO-NETWORK` field with a child policy. The child policy can override settings in the parent policy, allowing aggregation to occur at the endpoint. The child policy must specify a full-stack attestation record, to allow Riverbed to verify the identity of a *particular type* of trusted application-level code (e.g., SpamAssassin [119]). Data received from a trusted aggregator is marked with the taint descriptor of the receiving universe.

Riverbed allows a user to define her own policy for each web service that she uses. However, some policies may be fundamentally incompatible with certain services. As a trivial example, a Dropbox-like service that provides online storage is intrinsically incompatible with a `PERSISTENT-STORAGE: False` policy.[1] In the common case, we expect users to rely on trusted outside authorities, called *policy generators*, to define reasonable policies for sites. For example, consider a web site that wants to deliver targeted advertising via a third-party ad network `evil-ads.com`. A consumer advocacy group can advise users to avoid policies that whitelist `evil-ads.com`. Consumer advocacy groups can also publish suggested policy files for particular sites, based on research about what reasonable permissions for those sites should be.

Note that modern web browsing is already influenced by a variety of curated policies. For example, Google maintains a set of known-malicious URLs; multiple browser types consult this list to prevent accidental user navigation to attacker-controlled pages [120]. As another example, ad blockers [121] interpose on a page load, blocking content from sites deemed objectionable by the creators of the ad blocker. Riverbed introduces a new kind of web policy, but does not shatter prior

[1]... unless the service is intentionally exporting a RAM-only storage abstraction.

expectations that web browsing must be an unmediated experience.

3.3.3 Server Attestation

The client-side proxy shepherds the interactions between the client and server portions of a Riverbed application. In this section, we describe the proxy in the context of a traditional web service whose client/server protocol is HTTP. Proxies are easily written for other protocols like SMTP (§3.3.5). We assume that the reader understands the basics of remote attestation.

A user configures her browser to use the Riverbed proxy to connect to the Internet. At start-up time, the proxy searches a well-known directory for the user's policy files; the proxy assumes that each filename corresponds to the hostname in a server-side X.509 certificate (e.g., `x.com`). When the proxy receives an HTTP request that is destined for `x.com`, the proxy opens a TLS connection to `x.com`'s server, and forces that server to remotely attest its software stack. If the attestation succeeds, the proxy issues the HTTP request that triggered the attestation. Later, upon receiving a response from the server, the proxy forwards the response to the browser. By default, the proxy assumes that an attestation is valid for one day before a new attestation is necessary.

Riverbed strives to be practical, but traditional remote attestation [122, 123] has some unfortunate practical limitations. Consider the following challenges.

Server-side ambiguity: In traditional attestation, servers establish trust with clients by providing an explicit list of server-side software components. However, servers may not wish to share a perfectly-accurate view of their local software environment. For example, servers might be concerned that a malicious client will launch zero-day attacks against vulnerable (and precisely-identified) server components.

Potentially safe code: A server-side component may be intrinsically secure, but currently unvetted by the creator of a user's Riverbed policies. Alternatively, a server-side executable might

be intrinsically insecure, but perfectly safe to run if launched within a sandboxed environment like a virtual machine. Traditional attestation protocols are ill-suited to handle cases like these, since trust decisions are binary—a hash value in an attestation message corresponds to a categorically trusted component, or a categorically untrusted component.

Policy updating: A virtuous server administrator will be diligent about applying the latest patches to server-side code. If the user's policy generator is not as diligent, then users will reject legitimately trustworthy stacks as suspicious. Similarly, if users are more aggressive about updating policies than a server administrator, then out-of-date server-side stacks will be legitimately rejected as untrustworthy, but the server administrator will lack an immediate explanation for why. Traditional attestation protocols focus on the cryptographic aspects of client-server communication, but cannot resolve these kinds of policy disputes.

Riverbed uses the Cobweb attestation system [85] to handle these practical concerns. In traditional attestation, the attestor sends a TPM-signed `PCR[10]` value, and a list of *<filename, filehash>* tuples representing the objects that are covered by the cumulative hash in `PCR[10]`. Cobweb allows the attestor to augment the traditional attestation report with a *contextual graph* that provides additional information about the attestor's software stack. For example, a contextual graph might represent a process tree, where each vertex is a process and each edge represents a parent/child `fork()` relationship. An edge could also represent a dynamic information flow, e.g., indicating that two processes have communicated via IPC. Attestation verifiers specify policies as graph predicates that look for desired structural properties in the contextual graph or the regular attestation list of *<filename, filehash>* tuples.

Riverbed uses contextual graphs, and policy specification via graph predicates, to eliminate some of the practical difficulties with traditional attestation. For example:

- If attestation fails (i.e., if a client-side Riverbed proxy discovers that a graph predicate cannot be satisfied), the proxy sends the failed predicate to the server. The server can then initiate

concrete remediating steps, e.g., by updating software packages, or removing a blacklisted application.

- A Riverbed server can also dispute the failure of a graph predicate. For example, if a user's proxy believes that a particular server-side component is out-of-date, the server can respond with a list of signed, vendor-supplied updates for which the user's proxy may be unaware. The proxy can then ask the user's policy generator for a new policy.

- A user's Riverbed policy can tolerate an unknown or normally untrusted server binary if that binary is launched within a sandbox that isolates the component from other components which the user does require to be trusted. To provide confidence in the sandbox, the server's contextual graph should contain the `fork()/exec()` history for the server, as well as the configuration files for the sandbox environment. As a concrete example, suppose that a server needs to run a `telnet` daemon to communicate with a legacy internal service. The `telnet` protocol is known to be insecure, but a Riverbed proxy can trust the server's Apache instance if the server uses a virtual machine or a Docker container to isolate `telnetd`.

Riverbed also leverages Cobweb's support for server-side software ambiguity, but we refer the reader to the Cobweb paper [85] for a discussion of how Cobweb implements this feature.

3.3.4 Universes

Consider Alice, Bob, and Charlie, three Riverbed users whose policies are shown in Figure 10. The policies of these users are *almost* the same—they differ only with respect to the `AGGREGATION` token. Alice and Bob allow aggregation, but Charlie does not. How should Riverbed handle the data of these users on the server-side?

Riverbed could optimistically assume that the server-side application code will never try to aggregate Charlie's data with that of Alice or Bob. Riverbed executes the code atop a

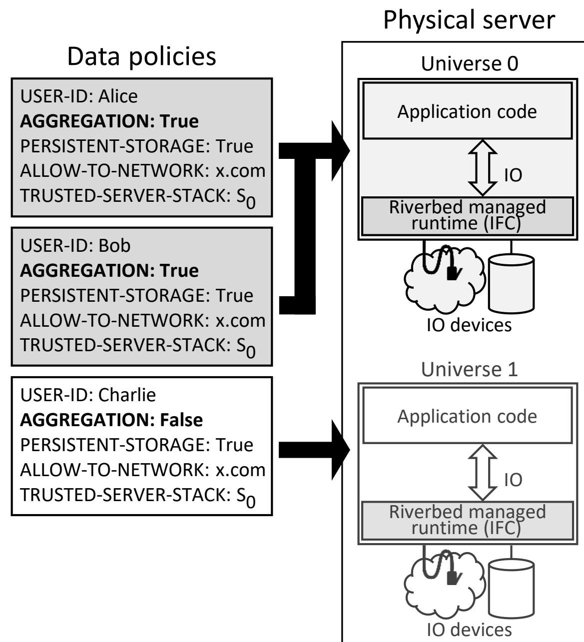


Figure 10: Alice and Bob have compatible policies, so Riverbed maps them to the same universe. Charlie has an incompatible policy because he disallows aggregation. Thus, Charlie must receive his own universe.

taint-tracking runtime (§3.3.5), so Riverbed could synchronously detect attempted violations of Charlie's policy. Unfortunately, attempted violations are likely, since Riverbed executes unmodified applications that are unaware of Riverbed policies. If a violation occurs, Riverbed would lack good options for moving forward. Riverbed could permanently terminate the application, which would prevent the disallowed aggregation of Charlie's data. However, all three users would be locked out of the now-dead service. To avoid this outcome, Riverbed could try to synchronously clone the application at policy violation time, creating two different versions: one for Alice and Bob, and another for Charlie. However, determining which pieces of in-memory and on-disk state should belong in which clone is difficult without application-specific knowledge; a primary goal of Riverbed is to enforce security in a service-agnostic manner.

Riverbed's solution emerges from the insight that Riverbed does not need to run any code to determine whether a set of policies might conflict. Instead, Riverbed can simply examine the policies themselves. For example, if a policy does not allow aggregation, then Riverbed can *pre-emptively* spawn a separate copy of the service for the policy's owner. Riverbed can spawn this copy

on-demand, upon receiving the first request from the owner. Now consider a policy *P* that allows aggregation and a particular set of storage and network permissions (e.g., `PERSISTENT-STORAGE: True` and `ALLOW-TO-NETWORK: x.com`). All users whose policies match *P* can be placed in the same copy of the service. Riverbed can spawn the copy upon receiving the first request that is tagged with *P*.

We call each service copy a *universe*. To implement the universe mechanism, Riverbed places a reverse proxy in front of the actual servers which run application code. Clients send their requests to the reverse proxy; the reverse proxy examines the policy in each request, spawns a new universe if necessary, and then forwards the request to the appropriate universe. Our Riverbed prototype instantiates each universe component inside of a Docker [124] instance that contains a taint-tracking runtime (§3.3.5) and the component-specific code and data. Docker containers are much smaller than traditional virtual machines since Docker virtualizes at the POSIX layer instead of the hardware layer. As a result, creating, destroying, and suspending universes is fast (§3.6.4). Docker runs each container atop a copy-on-write file system that belongs to the host [125]. Thus, universes share the storage that is associated with application code and other user-agnostic files.

Universes provide a final advantage: since all of the sensitive data in a universe has the same policy, a universe's taint-tracking runtime only needs to associate a single logical bit of taint with each object ("tainted" or "untainted"). If data from all users resided in the same universe, the runtime would have to associate each object with a value that represented a specific taint pattern.

The relationship between the number of users and the number of universes is application-specific. Some web services will specifically target a 1-1 mapping. For example, in a "private Dropbox" service that implements confidential online storage, users will naturally specify data policies that prevent aggregation (and thus require a universe per user). In contrast, social networking applications intrinsically derive their value from the sharing of raw user data, and the extraction of interesting cross-user patterns. For these applications, users must allow aggregation (although the scope of aggregation can be restricted using groups (§3.3.2)).

3.3.5 Taint Tracking

A managed language like Python, Go, or Java does not expose raw pointers to applications, or allow those applications to directly issue system calls. Instead, the language runtime acts as a mediation layer, controlling how a program interacts with the outside world. Like much of the prior work on dynamic tainting [115, 126–128], Riverbed enforces information flow control inside the managed runtime. Our Riverbed prototype modifies PyPy [129], a state-of-the-art Python interpreter, to extract Riverbed policies from incoming network data, and assign taint to derived information.

PyPy translates Python source files to bytecodes. Those bytecode are then interpreted. Riverbed adds taint-tracking instrumentation to the interpreter, injecting propagation rules that are similar to those of TaintDroid [115]. For example, in a binary operation like `ADD`, the lefthand side of the assignment receives the union of the taints of the righthand sides. Assigning a constant value to a variable clears the taint of the variable. If an array element is used as a righthand side, the lefthand side receives the taint of both the array and the index.

We call Riverbed's modified Python runtime PyRB. If a Python application tries to send tainted data to remote host `x.com`, PyRB first checks whether externalization to `x.com` is permitted by the tainted data's policy. If so, PyRB forces `x.com` to remotely attest its software stack; in this scenario, PyRB acts as the client in the protocol from Section 3.3.3. If `x.com`'s stack is trusted by the tainted data's policy, PyRB allows the data to flow to `x.com`. Otherwise, PyRB terminates the application. Riverbed provides a standalone attestation daemon that a server can use to respond to attestation requests.

PyRB must also taint incoming network data that was sent by end-user clients like web browsers. To do so without requiring modifications to legacy application code, PyRB assumes two things. First, clients are assumed to use standard network protocols like HTTP or SMTP. Second, PyRB assumes that when clients send requests using those protocols, clients embed Riverbed policies in a known way. Ensuring the second property is easy if unmodified clients run

atop Riverbed proxies; for example, when an unmodified web browser sends requests through a client-side Riverbed proxy, the proxy will automatically embed Riverbed policies using the Riverbed-`policy` HTTP header. Similarly, a client-side SMTP proxy can attach Riverbed policies using a custom SMTP command.

On the server-side, PyRB assumes that traffic intended for well-known ports uses the associated well-known protocol. Upon receiving a connection to such a port, PyRB reads the initial bytes from the socket before passing those bytes to the application. If the initial bytes cannot be parsed as the expected protocol, PyRB forcibly terminates the connection. Otherwise, if PyRB finds a Riverbed policy, PyRB taints the socket bytes and then hands the tainted bytes to the higher-level application code. If there is no policy attached to the bytes, PyRB hands untainted bytes to the higher-level code. Importantly, the application code is unaware of the tainting process, and cannot read or write the taint labels.

If policies allow server-side code to write to persistent storage, PyRB taints the files that the application writes. PyRB does whole-file tainting, storing taint information in per-file extended attributes [130]. PyRB prevents application code from reading or writing those attributes. Whole-file tainting minimizes the storage overhead for taints, but Riverbed is compatible with taint-aware storage layers (§3.6.2) that perform fine-grained tainting, e.g., at the level of individual database rows; the use of such storage layers will minimize the likelihood of overtainting.

When an application reads data from a tainted file, PyRB taints the incoming bytes, preventing the application from laundering taint through the file system. Note that, even though a policy contains multiple constraints (§3.3.2), all of the users within a universe share the same policy; thus, PyRB only needs to associate a single logical bit with each Python object (§3.3.4). PyRB does need to store one copy of the full policy, so that the policy can be consulted when tainted data reaches an output sink.

Managed languages sometimes offer “escape hatches” that allow an application to directly

interact with the unmanaged world. For example, in Java, the JNI mechanism [131] enables applications to invoke code written in native languages like C. In Python, interfaces like `os.system()` and `subprocess.call()` allow managed code to spawn native binaries. A Riverbed runtime can use one of three strategies to handle a particular escape hatch.

- The runtime can disallow the escape hatch by fiat.
- Alternatively, the runtime can whitelist the binaries that can be launched by the escape hatch. Each whitelisted binary must have a pre-generated taint model attached to it [115], such that the runtime can determine whether the binary is safe to launch given a particular set of tainted inputs, and if so, how taint should be assigned in the managed world when the binary terminates.
- The runtime can track instruction-level information flows in binaries launched by an escape hatch. To do so, the runtime must execute the native instructions via emulation [132, 133]. Strictly speaking, the runtime only needs to emulate instructions that touch sensitive data; the runtime can use page table permissions to detect when native code tries to access tainted data [134–136]. This optimization allows most native code to execute unemulated, i.e., directly atop the hardware.

PyRB could use any or all of these strategies. Our current PyRB prototype uses the first two. PyRB disallows C bindings by fiat, and only allows applications to spawn a child process if that process will be an instance of the PyRB interpreter (with the Python code to run in the child process specified as an argument to the child process). The parent and child PyRB interpreters will introspect on cross-process file descriptor communication, encapsulating the raw bytes within a custom protocol which ensures that taint is correctly propagated between the two runtimes.

3.4 Discussion

The necessity of IFC: A Riverbed server attests its systems software and its Riverbed managed runtime. However, the server does not attest the contents of higher-level code belonging to the web service. At first glance, this approach might seem odd: why not have the server attest all application code as well? If clients trust the attested application code, then server-side IFC might be unnecessary. However, in many cases, application code is not open source, e.g., because the code contains proprietary intellectual property that confers a competitive advantage to the web service owner. Code like this cannot be audited by a trusted third party, so end-users would gain little confidence from remote attestations of that code. Even if the server-side code were open source and publicly auditable, there are many more server applications than OSes and low-level systems software. Given a finite amount of resources that can be devoted to auditing, those resources are best spent inspecting the lowest levels of the stack. Indeed, if those levels are not secure, then even audited higher-level code will be untrustworthy. Also note that, even if the web service code has been audited, Riverbed provides security in depth, by catching any disallowed information flows that the audit may have missed.

Universe migration: Due to server-side load balancing or fail-over, a container belonging to a universe can migrate across different physical servers. From a user's perspective, migration is transparent if a user-facing container is placed on a server with a trusted stack—attestation involving the new server and the user's Riverbed proxy will succeed as expected. However, before migration occurs, the old server must force the new server to attest; in this fashion, the old server ensures that the new server runs a trusted Riverbed stack (and will therefore respect the data policies associated with the universe being migrated).

Preventing denial-of-service via spurious universe creation: Attackers might generate a large number of fake users, each of which has a policy that requires a separate universe; the attacker's goal would be to force the application to exhaust resources trying to manage all of the

universes. Fortunately, in a given Riverbed application, each universe employs copy-on-write storage layered atop a base image. As a result, a new universe consumes essentially zero storage resources until the universe starts receiving actual client requests that write to storage. Riverbed also suspends cold universes to disk. Thus, a maliciously-created universe that is cold will consume no CPU cycles and no RAM space; storage overhead will be proportional to the write volume generated by client requests, but this overhead is no different than in a non-Riverbed application. Regardless, a Riverbed application should perform the same user verification [137–139] that a traditional web service performs.

Hostname management: Applications which use a microservice architecture will contain many small pieces of code that are executed by a potentially large number of hostnames. An application that uses elastic scaling may also dynamically bind service state to a large set of hostnames. User policies can employ wildcarded TLS hostnames [140] to avoid the need for a priori knowledge of all possible hostnames.

Taint relabeling: Consider a user named Alice. A Riverbed service assigns Alice to a universe upon receiving the first request from Alice (§3.3.4). What happens if Alice later wants to re-taint her data, i.e., assign a different policy to that data?

Suppose that Alice lives in a singleton universe that only contains herself. Further suppose that her policy modification keeps her in a singleton universe. In this scenario, re-tainting data is straightforward. If storage permissions were enabled but now are not, Riverbed deletes Alice's data on persistent storage. If network permissions changed, then Riverbed will only allow tainted data to flow to the new set of whitelisted endpoints. Nothing special must be done to handle tainted memory in the managed runtime—since Alice still lives in a singleton universe, there is no way for the service to combine her in-memory data with the data of others. If Alice later wants to invoke her "right to be forgetten," Riverbed just destroys Alice's universe.

The preceding discussion assumed that Alice only has universe state in a single TLS

domain (e.g., `x.com`). However, Alice's singleton universe will span multiple domains if Alice's original policy enabled cross-domain data transfers. In these scenarios, Riverbed must disseminate a policy modification request to all relevant domains. Doing so is mostly straightforward, since the relevant domains are explicitly enumerated in Alice's original policy. Riverbed does need to pay special attention to wildcarded network sinks like `*.x.com`; such domains must expose a directory service that allows Riverbed to enumerate the concrete hostnames that are covered by the wildcard.

Now consider a different user Bob who wants to change his policy. If Bob lives in a universe that is shared with others, then re-tainting is harder, regardless of whether Bob wishes to transfer to a shared universe or a singleton one. The challenges are the same ones faced by a synchronous universe clone at policy-violation time (§3.3.4): since Riverbed is application-agnostic, Riverbed has no easy way to cleanly splice a user's data out of one universe and into another. Thus, if Bob lives in a shared universe and wishes to move to a different one, Riverbed must first use application-specific mechanisms to extract his data from his current universe. Then, Riverbed deletes Bob's current universe. Finally, Riverbed must reinject Bob's data into the appropriate universe via application-specific requests. This migration process may be tedious, but importantly, Riverbed narrows the scope of data finding and extraction. When re-tainting must occur, the application only needs to look for Bob's data within Bob's original universe, not the full set of application resources belonging to all users. Before and after re-tainting, Riverbed ensures that Bob's IFC policies are respected.

CDNs: Large-scale web services use CDNs to host static objects that many users will need to fetch. CDN servers do not run application logic, but they do see user cookies which may contain sensitive information. So, by default, client-side Riverbed proxies force CDN nodes to attest. However, a proxy can explicitly whitelist CDN domains that should not be forced to attest.

Policy creep: Traditional end-user license agreements represent a crude form of data consent. In a EULA, a service provider employs natural language to describe how a service will handle user data; a user can then decide whether to opt into the service. Riverbed tries to empower

users by giving *users* the ability to define policies for data manipulation. However, Riverbed cannot force a service to regard a user-defined policy as acceptable. Furthermore, the history of traditional EULAs suggests that, in a Riverbed world, services will prefer less restrictive Riverbed policies. For example, a service may refuse to accept a user if the user's Riverbed policy will not allow data flows to a particular advertising network. In this situation, the service can mandate that a less restrictive policy is the cost of admission to the service. Riverbed cannot prevent such behavior. However, Riverbed does force services to be more transparent about data promiscuity, because any service-suggested policy must be explicit about how data will be used. Riverbed also uses IFC to force services to adhere to policies.

Deployment considerations: Riverbed assumes that datacenter machines have TPM hardware. This assumption is reasonable, since TPMs are already present in many commodity servers.

In a complex, multi-tier application, components may span multiple administrative domains. The failure of some domains to run up-to-date stacks may lead to cascading problems with the overall application, as trusted stacks refuse to share data with unpatched ones. This behavior is actually desirable from the security perspective, and it incentivizes domains to keep their software up-to-date.

3.5 Implementation

The core of our Riverbed prototype consists of a client-side proxy (§3.3.3), a server-side reverse proxy (§3.3.4), and a taint-tracking Python runtime (§3.3.5). The two proxies, which are written in Python, share parts of their code bases, and comprise 773 lines in total, not counting external libraries to handle HTTP traffic [141] and manipulate Docker instances [142]. PyRB is a derivative of the PyPy interpreter [129], and contains roughly 500 lines of new or modified source code.

To implement remote attestation, servers used LG's UEFI firmware, which implemented the TPM 2.0 specification [143]. At boot time, the firmware extended a PCR with a TPM-aware version of the GRUB2 bootloader [144]. GRUB2 then extended the PCR with a TPM-aware version of the Linux 4.8 kernel. The kernel used Linux's Integrity Management Architecture [98] to automatically extend the PCR when loading kernel modules or user-mode binaries. Contextual attestation graphs were generated by Cobweb [85], with servers and client-side Riverbed proxies using the Cobweb library to implement the attestation protocol.

3.6 Evaluation

In this section, we demonstrate that Riverbed induces only modest performance penalties, allowing Riverbed to be a practical security framework for realistic applications. In all experiments, server code ran on an Amazon `c4` instance which had a 4-core Intel Xeon E5-2666 processor and 16 GB of RAM. The client was a 3.1 GHz Intel Core i7 laptop with 16 GB of RAM. The network latency between the client and the server was 14 ms.

3.6.1 Attestation Overhead

Before a client-side Riverbed proxy will send data to a server, the proxy will force the server to attest. We evaluated attestation performance under a variety of emulated network latencies and bandwidths. The client's policy required the attesting server to run a trusted version of `/sbin/init`, as well as trusted versions of 31 low-level system binaries like `/bin/sh`. The policy also used a Cobweb graph predicate (§3.3.3) to validate the process tree belonging to the Docker subsystem, ensuring that the tree contained no extraneous or missing processes.

Due to space restrictions, we only provide a summary of the results. Attestations were small (112 KB), so attestation time was largely governed by network latency, the cost of the slow

TPM `quote()` operation (which took 215 ms on our server hardware), and Cobweb overheads for graph serialization, deserialization, and predicate matching (which required 562 ms of aggregate compute time on the server and the client-side proxy). On a client/server network link with a 14 ms RTT, the client-perceived time needed to fetch and validate an attestation was 846 ms. Proxies cache attestation results (§3.3.3), so this attestation penalty is amortized.

3.6.2 Case Studies

To study Riverbed's post-attestation overheads, we ported three Python applications to Riverbed.

- MiniTwit [106] is a Twitter clone that implements core Twitter features like posting messages and following users. Application code runs in Flask [145], a popular server-side web framework. MiniTwit uses a SQLite database to store persistent information. We defined a Riverbed policy which allowed user data aggregation, and allowed tainted data to be written to storage and to other network servers in our MiniTwit deployment.
- Ionic Backup [107] is a Dropbox clone that provides a user with online storage. Ionic allows a user to upload, download, list, and delete files on the storage server. The Ionic client uses HTTP to communicate with the server. For this application, we defined a Riverbed policy which allowed user data to be written to disk, but disallowed aggregation, and prevented user data from being sent to other network servers.
- Thrifty P2P [108] implements a peer-to-peer distributed hash table [146, 147]. The primary client-facing operations are `PUT(key,value)` and `GET(key)`. Internally, Thrifty peers issue their own traffic to detect failed hosts, route puts and gets to the appropriate peers, and so on. For this application, we defined a Riverbed policy which allowed aggregation and storage, but only allowed tainted data to be written to endpoints that resided in our test deployment of Thrifty servers.

Operation	Without Riverbed	With Riverbed
MiniTwit view timeline	229 ms	252 ms
Ionic download	82.5 ms	83.1 ms
Ionic ls	14.1 ms	14.2 ms
Thrifty GET request	27.5 ms	28.0 ms

Figure 11: End-to-end response times for processing various user requests. For MiniTwit, the user viewed her timeline. For Ionic, the user downloaded a 300 KB file, or asked for a list of the contents of a server-side directory. For Thrifty, the client fetched a 20 byte value from a DHT that contained 2 nodes; the DHT was intentionally kept small to emphasize the computational overheads of Riverbed. The client/server network latency was 14 ms. Each result is the average of 50 trials.

Operation	Regular PyPy	PyRB (no taint)	PyRB (taint)
MiniTwit post message	14 ms	15 ms	15 ms
MiniTwit view timeline	4.1 ms	4.2 ms	4.2 ms
MiniTwit follow user	13 ms	15 ms	15 ms
Ionic upload	2.3 ms	2.5 ms	2.5 ms
Ionic download	4.8 ms	5.0 ms	5.0 ms
Ionic ls	0.43 ms	0.50 ms	0.50 ms
Thrifty PUT request	0.16 ms	0.17 ms	0.19 ms
Thrifty GET request	0.19 ms	0.24 ms	0.24 ms

Figure 12: Server-side overheads for processing various user requests. The workloads are a superset of the ones in Figure 11. Each result is the average of 50 trials.

Ionic required no modifications to run atop Riverbed. Thrifty peers used a custom network protocol to communicate; so, we had to build a proxy for the Thrift RPC layer [148] that injected Riverbed policies into outgoing messages, and tainted incoming data appropriately. MiniTwit's core application logic required no changes, but, to reduce the likelihood of overtainting, we did modify MiniTwit's Python-based database engine to be natively taint-aware, e.g., so that each database row had an associated on-disk taint bit, and so that query results were tagged with the appropriate union taints, based on the items that were read and written to satisfy the query. Our modifications are hidden beneath a narrow abstraction layer, making it easy to integrate the Python-level MiniTwit logic with off-the-shelf taint-tracking databases [149–151].

Figure 11 depicts end-to-end performance results for MiniTwit, Ionic, and Thrifty. The results demonstrate that Riverbed imposes small client-perceived overheads (1.01x–1.10x). Fig-

Benchmark	Overhead
Django	1.14x
Render HTML table	1.16x
Code run in PyPy interpreter	1.08x
JSON parsing	1.13x
Python git operations	1.01x
SQL Alchemy	1.05x
Spitfire	1.19x
Twisted	1.17x
Fractal Generation	1.18x
Spectral Norm	1.10x
Raytracing	1.19x

Figure 13: PyRB's performance on representative benchmarks from the Performance benchmark suite [152]. PyRB's performance is normalized with respect to that of regular PyPy. No data was tainted in these experiments.

ure 12 isolates Riverbed's server-side computational penalties. For each request type, we compare server-side performance when using unmodified PyPy, PyRB in which no data is tainted, or PyRB in which data is tainted according to the policies that we described earlier in this section. For MiniTwit, Riverbed had overheads of 1.02x–1.15x. For Ionic, Riverbed imposed overheads of 1.04x–1.16x. For Thrifty, puts and gets had slowdowns of 1.18x and 1.26x respectively. Riverbed imposed the least overhead for Ionic's "remove" and "delete" operations, since PyRB could handle these operations merely by issuing file system calls, without handling much in-memory data that had to be checked for taint. In contrast, operations that involved reading or writing network data required PyRB to interpose on data processing code, even if no data was tainted, and perform extra work at data sources and sinks.

3.6.3 PyPy Benchmarks

For a wider perspective on PyRB's performance, we used PyRB to run the benchmarking suite from the Performance project [152]. The suite focuses on real Python applications, downloading the necessary packages for those applications and then running the real application code. Figure 13 shows PyRB's performance on a representative set of benchmarks. The benchmarks that are above

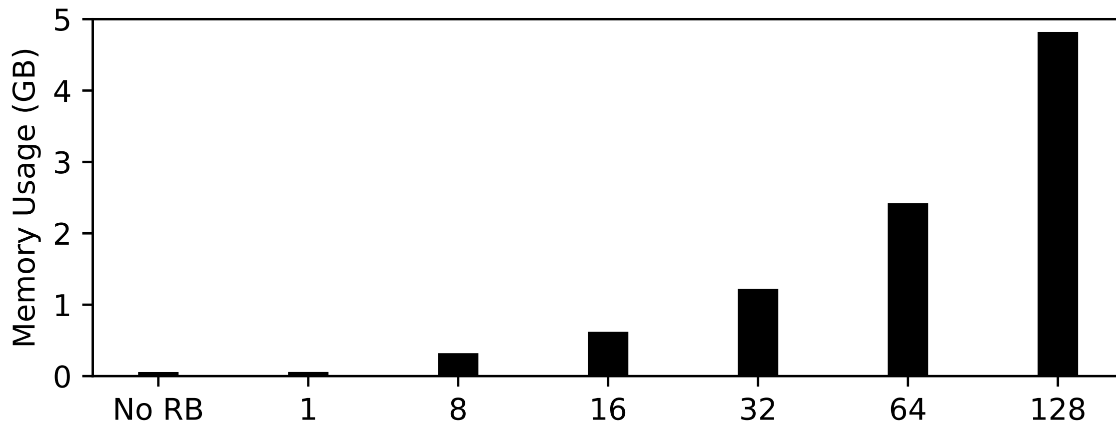


Figure 14: Physical memory pressure in MiniTwit when run without Riverbed, or with Riverbed using various numbers of universes. Note that in MiniTwit, each universe requires only one container. In each test configuration, we measured memory pressure after submitting 1000 requests to each MiniTwit instance that existed in the configuration.

the thin black line resemble applications that might run inside of a Riverbed universe; these benchmarks perform actions that are common to web services, like parsing HTML, responding to HTTP requests, and performing database queries. These benchmarks tend to be IO-heavy, with occasional CPU idling as code waits for IOs to complete. In contrast, the benchmarks beneath the thin black line are CPU-intensive. PyRB does not affect the speed of IOs, but does affect the speed of computation, so PyRB has slightly higher overhead for the bottom set of benchmarks. Overall, PyRB is at most 1.19x slower. These results overestimate PyRB's overheads because clients and servers resided on the same machine (and thus incurred zero network latency).

3.6.4 Universe Overhead

The size for a base Riverbed Docker image is 212 MB. The image contains the state that belongs to the PyRB runtime, and is similar in size to the official PyPy Docker image [153]. Each Riverbed service adds application-specific code and data to the base Riverbed image. However, a live Docker instance uses copy-on-write storage, so multiple Riverbed universes share disk space (and in-memory page cache space) for common data.

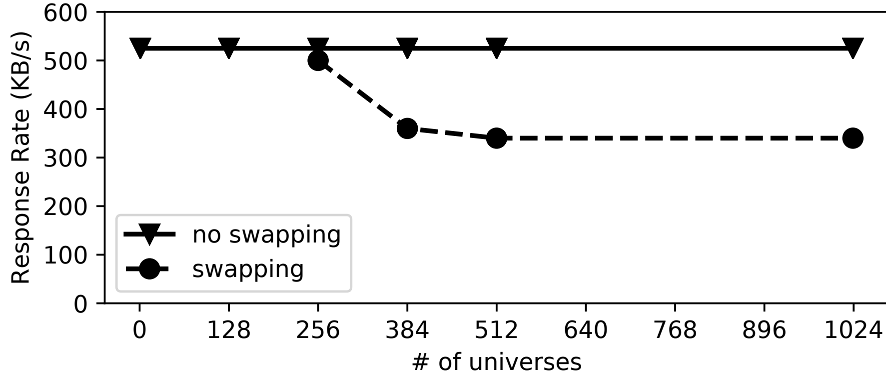


Figure 15: MiniTwit server response rate as a function of (1) the number of universes, and (2) whether the server had 60 GB of RAM or 16 GB of RAM. We used the Apache Benchmark tool [100] to simulate clients that requested MiniTwit timelines which had 100 messages. In each trial, we submitted 1000 requests, with 100 outstanding requests at any given time. For the server with 16 GB of RAM, swapping began with 256 universes.

We believe that for most Riverbed applications, the universe abstraction will not increase overall storage requirements; in other words, the space needed for per-universe data plus shared-universe data will be similar to the space needed for the non-Riverbed version of the application. For example, in MiniTwit, for a given number of timelines with a given amount of posts, the storage requirements are the same if the timelines are partitioned across multiple Riverbed universes, or kept inside a single, regular MiniTwit deployment. However, Docker's copy-on-write file system does result in slower disk IOs. As a concrete example, we measured MiniTwit's database throughput when MiniTwit ran directly atop ext4, and when MiniTwit ran inside a universe that used Docker's overlayfs file system [125]. We examined database workloads with read/write ratios of 95/5 and 50/50, akin to the YCSB workloads A and B [154]. The targeted database rows were drawn from a Zipf distribution with $\beta = 0.53$, similar to the distribution observed in real-life web services [155, 156]. We found that, inside a Riverbed universe, transaction throughput slowed by 7.7% for the 95/5 workload, and by 17.3% for the 50/50 workload.

For our three sample applications, spawning a new Docker container required 260–280 ms on our test server. In Riverbed, the container creation penalty is rarely paid; the reverse proxy only has to create a new universe upon seeing a request with a policy that is incompatible with all

pre-existing universes. Subsequent requests which are tagged with that policy will be routed to the pre-existing universe.

Creating new universes is rare, but pausing and unpausing old ones may not be. If an application has many universes, and memory pressure on a particular physical server is high, then temporarily-quiescent universes can be suspended to disk. On our test server with 512 live containers, pausing or unpausing a single Docker instance took roughly 30 ms. However, recent empirical research has shown that in datacenters, a tenant's resource requirements are often predictable [157]. Thus, universes can be assigned to physical servers in ways that reduce suspension/resumption costs.

Docker virtualizes at the POSIX level, so the processes inside of a Riverbed universe are just processes inside of the host OS. As a result, the RAM footprint for a Riverbed universe is just the memory that is associated with the host processes for the universe. Our Riverbed prototype was able to spawn up to 1023 live containers on a single server. This 1023 bound is a well-known limitation of the current Docker implementation. Docker associates a virtual network card with each instance, and attaches the virtual card to a Linux network bridge [158]; a Linux bridge can only accept 1023 interfaces. Regardless, the current bound of 1023 containers per machine does not imply that a single application can have at most 1023 universes. The bound just means that, if an application has more than 1023 universes, then those universes must be spread across multiple servers. Riverbed's reverse proxy (§3.3.4) considers server load when determining where to create or resurrect a universe; thus, the per-server container limit is not a concern in practice.

Figure 14 demonstrates that Riverbed's memory pressure is linear in the number of active containers. As shown in Figure 15, a large number of universes has no impact on server throughput if all of the hot universes fit in memory. Unsurprisingly, throughput drops if active universes must be swapped between RAM and disk. However, a Docker container is just a set of Linux processes that are constrained using namespaces [159] and cgroups [160]; thus, the memory overhead for launching a Riverbed universe with *N* processes is similar to the memory overhead of scaling out

a regular application by creating N regular processes. That being said, a Riverbed application does create processes more aggressively than a normal application. In Riverbed, incompatible policies require separate universes (and therefore separate processes), even if aggregate load across all universes is low.

3.7 Conclusion

Riverbed is a platform that simplifies the creation of web services that respect user-defined privacy policies. A Riverbed universe allows a web service to isolate the data that belongs to users with the same privacy policy; Riverbed's taint tracking ensures that the data cannot flow to disallowed sinks. Riverbed's client-side proxy will not divulge sensitive user data until servers have attested their trustworthiness. Riverbed is compatible with commodity managed languages, and does not force developers to annotate their source code or reason about security lattices. Experiments with real applications demonstrate that Riverbed imposes no more than a 10% performance degradation, while giving both users and developers more confidence that sensitive data is being handled correctly.

4 Oblique: Accelerating Page Loads Using Symbolic Execution

4.1 Motivation

More than 50% of all web requests now originate from smartphones [161]. Thus, optimizing mobile page loads is important. Many mobile users (particularly in emerging markets) are still stuck behind slow 3G and 4G links; even high-bandwidth 5G links often suffer from 4G latencies [162]. Unfortunately, page load times are usually determined by latency, not bandwidth [33, 163]. A variety of mobile page accelerators try to mask last-mile latency by (1) analyzing the objects (e.g., HTML and JavaScript files) that are contained by a page, and then (2) reducing the perceived fetch latencies for those objects (e.g., using server-side pushing [33–35, 164, 165] or client-side prefetching [163, 164]).

More than 90% of web requests now use HTTPS instead of HTTP [166]. The shift from HTTP to HTTPS threatens the viability of traditional web acceleration techniques, creating tensions between security, performance, and the financial cost of hosting a web site.

- Some accelerators like Silk [34] that perform remote dependency resolution [33, 35, 37] route client traffic through third-party proxies; these proxies are owned by browser vendors or mobile providers, and are operated for the benefit of customers. The proxies require access to cleartext HTTPS content to determine which objects to prefetch (§4.2). Thus, content providers that use HTTPS are faced with a dilemma: allow third-party proxies to man-in-the-middle TLS connections, or forgo the performance benefits provided by outsourced web accelerators. The former choice breaks end-to-end TLS security, and the latter option hurts page load times.
- Other accelerators like Vroom [164] do not rely on third-party proxies, but instead use

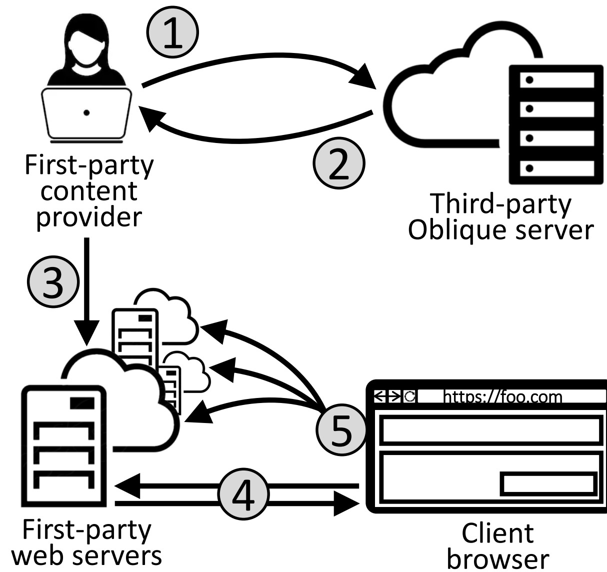


Figure 16: Overview of Oblique's design. A developer uploads page content to Oblique's analysis server (①). Oblique returns the path constraint tree for the page (②). The developer uploads the page content to web servers (③), injecting Oblique's JavaScript library into the page's HTML. Later, when a user loads the page (④), the prefetching library uses the path constraint tree to prefetch objects (⑤).

first-party analysis servers to identify the list of resources required for pageload. For example, Vroom uses offline first-party analysis to identify an initial set of prefetchable objects in a page; later, when a real client loads the page's top-level HTML, the first-party Vroom web server analyses the returned HTML on-the-fly to discover an additional set of prefetchable objects. While this approach does not expose cleartext TLS data to third parties, it prevents outsourcing of the prefetching analysis, requiring first parties to pay for VM cycles to run the analysis code.

This chapter proposes Oblique, a new system for accelerating page loads. Oblique's goal is to improve the load time reductions provided by state-of-the-art accelerators, while enabling *cheaper and more secure outsourcing* of the analyses which identify the objects that a client should prefetch. Figure 16 depicts Oblique's architecture. When a content provider creates a new page, the provider feeds the new content to a third-party Oblique server. The server performs a *symbolic page load*, exploring the possible behaviors of a web browser and a web server during the page load process. The output of the symbolic page load is a *path constraint tree*, as shown in Figure 17.

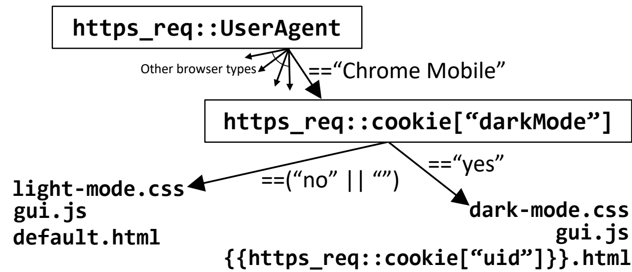


Figure 17: A simplified example of a path constraint tree. At page load time, Oblique's client-side JavaScript library traverses the tree, using load-time concrete values to trace a path to a leaf. Each leaf in the tree enumerates which URLs Oblique should prefetch. Those URLs may need to be concretized with load-time values (e.g., a cookie value in this example).

Each leaf is a set of URLs that a client should prefetch, and each path from root to leaf represents symbolic constraints on the actual client-side and server-side state that is observed at the time of a real page load. During an actual page load, the server-side code and the client-side browser inspect concrete state like a client's cookie, traces a path through the constraint tree, and prefetches the relevant objects using a client-side JavaScript library.

Oblique provides three benefits:

Performance: Oblique reduces page load times by up to 31%. Oblique does so without requiring any changes to end-user browsers.

Security: Oblique's offline analysis server does not see concrete values for uniquely-identifying client data. For example, the server does not observe concrete values for any user's cookies or `User-Agent` string. Instead, Oblique's analysis server only sees page content that could have been fetched by any actor on the Internet who can issue HTTPS requests. Later, during an actual page load, Oblique's analysis server is totally uninvolved, and receives no information about sensitive concrete values.

Financial cost: Oblique's offline symbolic analysis occurs when a new page version is created. The analysis cost (as measured by VM rental fees) is amortized across all client loads of the page. For popular pages, this amortized expense will be less than the aggregate per-page-load costs incurred by third-party RDR or first-party accelerators like Vroom.

4.2 Background

A web page's *dependency graph* [163] captures the load-order relationships between a page's constituent objects. For example, a page's top-level HTML might contain references to a JavaScript file and an image. To load the page, a browser must fetch and evaluate both objects. Evaluating the JavaScript file might generate additional fetches, e.g., because the executed JavaScript code uses the `Fetch` API to issue new HTTP requests. Evaluating the image file causes the associated pixels to be displayed; the reception of the image data may also trigger JavaScript `onload` event handlers. Those handlers can generate more fetches. The overall page load completes when a critical subset of a page's objects have been fetched and evaluated. Different load metrics use different criteria to identify the critical subset (§4.6).

Web accelerators leverage knowledge of a page's dependency graph to reduce a page's load time. One popular approach is remote dependency resolution (RDR) [32–37]. An RDR system deploys a proxy server that has low-latency paths to the Internet core. An end-user's browser sends each page load request to the proxy. Upon receiving such a request, the proxy launches a headless browser (i.e., a browser that lacks a GUI). The proxy-side browser loads the requested page and streams the fetched objects to the user's browser. By doing so, the proxy can partially mask the user's high last-mile latency: the page's dependency graph is resolved via the proxy's fast network links, and the bytes in each discovered object are pushed to the client as soon as the proxy receives those bytes.

RDR can reduce page load times by up to 40% [33]. Unfortunately, RDR proxies are

computationally expensive to run, because web browsers (even headless ones) are complex, resource-intensive applications. A proxy can use backwards program slicing [167] to try to only execute the JavaScript code that influences calls to functions like `Fetch()`. However, slices are often inexact, and the degraded prefetching underperforms traditional RDR for 34% of pages [165].

An RDR proxy must act as a man-in-the-middle for TLS connections. Doing so allows the headless browser to parse cleartext web content and fetch the same objects that a user's browser will eventually want to fetch. However, breaking TLS's end-to-end security is obviously problematic; it allows RDR proxies to see user cookies and other sensitive HTTPS content.[2] This security violation also plagues non-RDR accelerators that perform third-party analysis of dependency graphs [168–170]. Cryptographic schemes that allow middlebox computation over encrypted TLS data [171] are insufficiently expressive to analyze dependency graphs; prefetch analysis requires a Turing-complete language to parse HTML and evaluate JavaScript.

Incremental Resource Updates: Micro-cache [172] is a technique that updates a resource's only changed portions of code or layout instead of its entirety. For example, Fawkes [173]'s webpages download and execute a Javascript library that updates only changed HTML tags. Wang et al. [174] leverages smart-caching [175] to cache intermediate results of style formats and thereby reduces re-computed CSS layouts. Oblique is orthogonal to these techniques and complements them to improve prefetching success rates of dynamic URLs.

Energy-saving Techniques: Batching data transfer [176] reduces a device's energy consumption by reducing the physical network usage duration. Oblique, as a side effect, partially achieves this goal by prefetching sub-resources all at once in the beginning of a pageload. Oblique's resource-prefetching technique doesn't conflict with other energy-saving techniques such as trimming unused Javascript/CSS code portions [177], adaptively adjusting a device screen's frame painting rate [178], or throttling certain graphic-intensive Javascript routines by inserting

[2]WatchTower [33] allows each HTTPS origin to run its own RDR proxy. This approach solves the security problem by exacerbating the computational overhead problem, since now *every* HTTPS origin must run a proxy.

sleeps [179].

Vroom [164] is a first-party web accelerator: dependency analysis runs on infrastructure belonging to the content provider. For each page, Vroom performs both offline and online analysis. The offline phase runs periodically (e.g., once an hour), using a headless browser to collect the set of URLs loaded by a page. Across multiple offline page loads, Vroom identifies a "stable set" of URLs that were fetched during each load. When a client initiates a real page load, a Vroom-modified web server parses HTML on-the-fly while streaming it to the client, extracting the embedded URLs. These embedded URLs, plus the ones found during offline analysis, comprise the set of URLs to prefetch. The web server induces the client to speculatively load these URLs via a combination of HTTP/2 push [180] and `<link>` prefetch hints [181].

Vroom's analyses run on first-party machines, so HTTPS secrets are not leaked to third parties. However, Vroom's online analysis cannot be outsourced securely: a benevolent mobile provider who wants to run Vroom on behalf of its users will have to break the HTTPS confidentiality of real user page loads. Vroom's offline phase also requires hand-tuning to deal with the heterogeneity of client browsers. For example, many sites define mobile and desktop versions of each page. A server determines which version to return by examining the `User-Agent` header in a client's HTTP request. Vroom's offline phase must be manually configured to explore the state space of all client-specific parameters like `User-Agent` and the client's screen size. Oblique's symbolic analysis allows Oblique to automatically explore this state space.

4.3 Design

At a high level, Oblique's offline analysis generates a *prefetch tree* for a page. The tree informs a client which HTTPS objects to prefetch in which situations. The input to the tree traversal is client-specific, potentially-sensitive information like cookie values; the output is a set of URLs. Oblique generates the tree by symbolically evaluating the client-side of a page load (§4.3.2). The

URLs (found at the leaf nodes) are symbolic expressions that a client makes concrete by plugging in client-specific information that is never revealed to Oblique. If a page uses Node [182] (a popular server-side JavaScript framework) to generate HTML, Oblique can also symbolically evaluate server-side code (§4.3.4). Receiving visibility into both client and server execution allows Oblique to generate prefetch trees with more true positives and fewer false negatives: in other words, clients will fetch more useful objects and fewer unnecessary ones.

4.3.1 Overview of Concolic Execution

Oblique uses a particular variant of symbolic evaluation called concolic execution [183, 184]. In concolic execution, a program is given a concrete set of initial inputs. The program is then executed under the observation of the concolic framework. The concolic framework assigns a "shadow" symbolic expression to each input value and to each internal program variable. An input's initial symbolic expression is only constrained by the limitations of the input's type. For example, a `uint32` input x might receive an initial concrete value of 2, but an initial symbolic constraint of $(0 \leq x \leq 2^{32} - 1)$. During the program's execution, the assignment $y = x/2$ would result in y receiving the concrete value of 1, and the symbolic constraint $y == x/2$. When the program's execution hits a branch statement (e.g., `if(x >= 42){...}else{...}`), execution proceeds along the appropriate path, but the symbolic expressions for the branch-test variables are updated. In the running example, the `else` clause is executed because x (equal to 2) is less than 42; x's symbolic constraint is updated to become $(0 \leq x < 42)$. As the program continues execution, variables receive updated concrete values and updated shadow constraints. Eventually, the program halts or a timeout fires. The concolic framework then explores a different execution path by backtracking along the branch history and selecting a branch direction to invert. In the running example, the concolic framework might choose to explore the taken side of the branch `if(x >= 42){...}else{...}`. To do so, the framework inverts the relevant part of x's symbolic expression, generating the constraint $(42 \leq x \leq 2^{32} - 1)$. The framework consults an

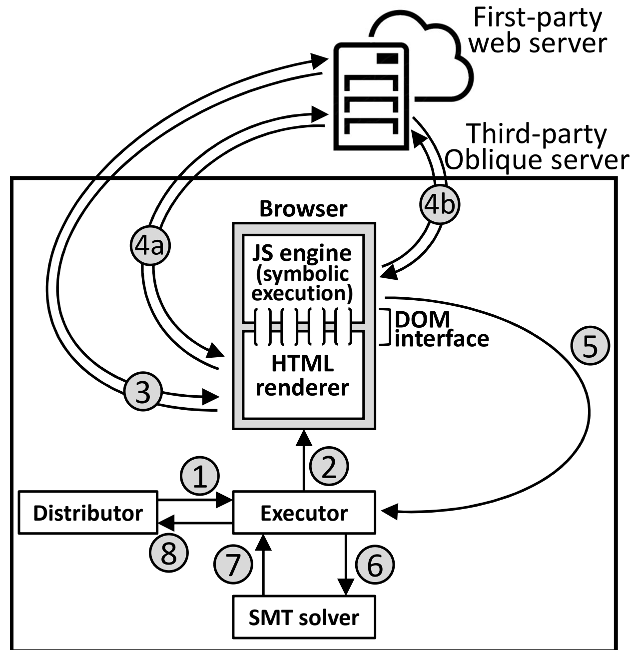


Figure 18: Overview of Oblique's approach for symbolically evaluating a client-side browser. See the mainline paper text for a description of each step.

SMT solver [185, 186] to generate a concrete value for x that satisfies the new constraints. Concolic execution then proceeds down the new branch until the program terminates or a timeout fires. This backtrack-and-explore pattern repeats until all execution paths have been discovered or (more likely) the overall time budget for concolic execution expires. For each discovered path, the framework records the *path constraints*, i.e., the symbolic constraints on all of the input variables which must be true for the path to be taken. Note that path constraints are different than the symbolic constraints on a particular variable. In our running example, the constraint on y is $y == x/2$. The path constraints for that execution path are the aggregate set of constraints placed on x and the rest of the program inputs.

4.3.2 Analyzing Client-side Behavior

In the context of a concolic page load, the program inputs are client-specific environmental variables. These environmental variables determine the content returned by web servers, and the execution

Input name	HTTP header	JavaScript variable	Description
User agent	User-Agent	`navigator.userAgent`	The local browser type, e.g., "Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/20010725 Netscape6/6.1"
Platform	Included in User-Agent	`navigator.platform`	The local OS, e.g., "Win64"
Screen characteristics	N/A	`window.screen.*`	Information about the local display, e.g., the dimensions and pixel depth
Host	Host	`location.host`	Specifies the host and port number used by request
Referrer	Referer	`document.referrer`	The URL of the page whose link was followed to generate a request for the current page
Origin	Origin	`location.origin`	Like Referrer, but only includes the origin, omitting path information
Last modified	Last-Modified (response)	`document.lastModified`	Set by the server to indicate the last modification date for the returned resource
Cookie	Cookie (request), Set-Cookie (response)	`document.cookie`	A string containing "key=value" pairs

Table 8: Symbolic inputs to a client-side page load.

paths taken by a page's JavaScript. For example, when a server receives the HTTP request for an HTML file, the server may examine the `User-Agent` header to determine whether to return the mobile-optimized HTML or the desktop-optimized HTML. The value for the local browser's `User-Agent` header is accessible to JavaScript via the `navigator.userAgent` variable; JavaScript code might inspect that variable to execute different code paths for different browsers. Thus, a client's user agent string is an input to the concolic page load. Table 8 enumerates the client-side inputs that Oblique considers.

Figure 18 depicts the life-cycle for a concolic page load. A distributor assigns concrete values to the inputs; the cookie value is set to an empty string, and other inputs are set to default values for mobile Chrome. The distributor hands these values to the executor (①). The executor launches a modified web browser (②) that fetches the page's top-level HTML (③). The HTTP request for the top-level HTML uses the environmental values selected by the distributor. Note that

the returned HTML will be a concrete string, not a symbolic one.

As the browser parses the HTML, the browser fetches and evaluates non-JavaScript files like CSS and images (4a). When a JavaScript file is fetched (4b), Oblique evaluates it using a modified version of the ExpoSE concolic engine [187]. As the JavaScript code executes, Oblique records the path constraints, and updates JavaScript variables with concrete values and symbolic constraints. When JavaScript code dynamically fetches an HTTP object (e.g., via `fetch(url)`), Oblique uses the *concrete* value of `url` to issue a real fetch. However, Oblique also records the symbolic constraints on `url`. These constraints, which represent a *symbolic URL*, are added to the prefetch list for the current execution path. As a contrived example, a symbolic URL might have the value "`x.com/?{{encodeURI(navigator.userAgent)}}`"; this URL would allow a web server to return different HTML to mobile clients and desktop clients.

In the prior example, the `{{}}` notation indicates a symbolic expression. The example also demonstrates how Oblique is enlightened about certain native functions like `encodeURI()`. Native functions are JavaScript-invocable methods whose implementations are provided by C++ code inside the browser. Oblique intentionally avoids the concolic execution of native code, since JavaScript-level semantics are the only ones of importance. However, to ensure that native methods correctly propagate JavaScript-level symbolic constraints, Oblique must associate a symbol policy with each native method. A policy describes how the symbolic inputs to a native method should be translated to symbolic outputs for the method. Oblique assigns policies to the most popular native methods that were seen in our test corpus (§4.6.1). Those methods include the ones defined by the `Math`, `String`, and `RegExp` objects. If a page invokes a native method that lacks a symbol policy, Oblique uses the concrete return value as the symbolic constraint; in other words, the native function acts as a black box that never returns symbolic data.

An HTML renderer maintains an internal data structure called the DOM tree. The DOM tree mirrors the structure of a page's HTML, with each HTML tag having a corresponding DOM node. JavaScript code uses the DOM interface to query or modify the DOM tree, e.g., to implement

animations and register event handlers for GUI activity. During a symbolic page load, Oblique associates the DOM tree with a concrete HTML string and a symbolic one; the latter allows JavaScript-level symbols to flow into and out of the DOM tree via DOM methods. For example, given a reference `r` to a `<div>` tag's DOM node, JavaScript code could display the browser type using the assignment `r.innerHTML = navigator.userAgent`. A read of `r`'s parent in the DOM tree (e.g., `r.parentNode.innerHTML`) would return a string whose symbolic value contains `{{UserAgent}}`.

As the page load unfolds, Oblique logs the symbolic URLs that are passed to network APIs like `fetch()`. Oblique also interposes on the DOM interface, and logs the symbolic URLs which cross that interface. For example, suppose that JavaScript code uses the `Node.appendChild(imgNode)` method to add a new `<img>` tag to the page. Oblique would log the symbolic URL associated with the `imgNode.src` attribute; logging the URL reflects the fact that executing `Node.appendChild(imgNode)` causes the browser to fetch an image from a remote server.

Oblique's HTML renderer also logs the static, non-symbolic URLs in a page. These URLs are directly specified in a page's static HTML (e.g., `<link rel="stylesheet" href="styles.css">`) or dynamically injected by JavaScript via the DOM interface. The prefetch list for an execution path contains the static, non-symbolic URLs and the dynamic, possibly-symbolic URLs that are fetched by the path.

Oblique declares the page load to be done when the JavaScript `onload` event fires. The browser fires this event when the browser has finished the HTML parse, fetched all objects discovered by the parse, and evaluated all of those objects. As shown in Figure 18, the JavaScript engine informs the executor about the path constraints for the page load (⑤). The executor asks the SMT solver to invert a branch direction at some point along the path (⑥). Inverting the branch direction changes the symbolic constraints on the input values (§4.3.1). The SMT solver generates concrete input values that satisfy the new constraints (⑦). The executor returns those concrete input

values to the distributor (⑧). These values represent a new test case that would cause the page to explore a different execution path.

The distributor launches many executors in parallel, running each one on a separate core. As the executors complete and return new test cases, the distributor launches new executors to explore new test cases. The distributor stops creating new executors once a predetermined time budget expires, or there are no more paths to explore. Higher budgets allow Oblique to discover more execution paths, but are more expensive in terms of VM costs. We evaluate these tradeoffs in Section 4.6.2.

When an executor completes its concolic page load, it logs two things: the list of symbolic URLs fetched by the page load, and the symbolic constraints on client-specific inputs like cookies. Once all executors have finished, the distributor analyses the aggregate set of executor logs to generate a tree of path constraints. Figure 17 provides an example of such a tree. Each leaf contains a set of symbolic URLs; each root-to-leaf path represents the client-specific input values which indicate that a page load will fetch the URLs at the leaf. The distributor translates the constraint tree into a JSON data structure. Finally, the distributor generates a JavaScript library that traverses the tree; at each node, the library applies regular expressions and comparison operators to the JavaScript representation of client-specific inputs (see Table 8). For example, the JavaScript code `/CriOS(54|55)/.test(navigator.userAgent)` determines whether the local browser is Chrome version 54 or 55 that runs atop iOS. Upon arriving at a leaf, the library concretizes the symbolic URLs in the leaf, and then prefetches those URLs using `XMLHttpRequest`.

Oblique sends the prefetching library (which embeds the JSON constraint tree) to the first-party web developer. The developer adds the library as an inline `<script>` tag at the beginning of the associated page's HTML. Later, when a real client browser loads the page, the library issues asynchronous prefetches, populating the local browser cache. As the browser's HTML parse examines the rest of the page and discovers references to external objects, the browser can pull those objects from its cache, avoiding wide-area fetch latencies.

4.3.3 Nondeterministic JavaScript Functions

JavaScript defines two categories of nondeterministic functions. Timestamp functions like `Date()` and `Performance.now()` read the system clock. Random number generators like `Math.random()` and `crypto.getRandomValues()` create pseudorandom or cryptographically-random byte sequences.

JavaScript code may consult nondeterministic functions during the construction of a dynamic URL. For example, a page might contain code like `if(Math.random() > 0.7){url="a.jpg"} else{url="b.jpg"}`. In that example, the URL embeds no symbols, but its value is controlled by the output of a nondeterministic function. Code like `url=Date() + ".jpg"` would create a URL that directly embeds the output of a nondeterministic function.

Both kinds of dynamic URLs will induce prefetch misses for RDR. The reason is that RDR uses a headless browser to generate a page's dependency graph (§4.2). The headless browser and the client-side browser will likely generate different nondeterministic values; thus, the two browsers will likely generate different dynamic URLs. To prevent such divergence, RDR could log the nondeterminism observed by the headless browser, and then force clients to use the logged sequence. This approach is the same one used by deterministic replay debuggers to faithfully recreate previously-observed program executions [188,189]. However, in the context of accelerating page loads, this approach can break functionality. Clients will receive old wall-clock readings, and calculate elapsed time periods that do not accurately reflect the client's true perception of time. As a result, clients may fetch stale content or improperly calculate frame rates for animations. From the security perspective, exposing a client's `crypto.getRandomValues()` sequence to a third party is undesirable, because the client might use the sequence to derive keys or nonces.

Vroom will also suffer prefetch misses for dynamic URLs that are influenced by nondeterministic functions. Vroom's offline analysis identifies a stable set of URLs that are fetched by several different loads of a page (§4.2). Vroom's stable set analysis will drop URLs that

only differ by a timestamp or a random number. The analysis will also drop URLs that do not directly embed nondeterminism, but are fetched via branching paths whose directions are chosen by nondeterminism.

Oblique handles these dynamic URLs without forcing clients to divulge their nondeterminism to third parties. During an offline symbolic execution, Oblique creates a unique, hidden variable for each invocation of a nondeterministic function. Oblique treats this variable as a client-specific input, akin to `document.cookie` or `User-Agent`. This approach enables Oblique to track how the outputs of nondeterministic functions influence branch decisions and the construction of dynamic URLs. For example, suppose that during symbolic execution, a page's JavaScript code invokes `Math.random()` twice, and then calls `Performance.now()`. Oblique generates the hidden variables $\texttt{rand}_0$, $\texttt{rand}_1$, and $\texttt{pnow}_0$. As the symbolic page load continues, the load may generate dynamic URLs like `https://foo.com/?{{rand`$_0$`}}.js`. Oblique places these URLs in the prefetch list as normal. The symbolic execution may also branch on the values of $\texttt{rand}_1$ and $\texttt{pnow}_0$, just like the symbolic execution might branch on `User-Agent`. Later, during a real client-side page load, Oblique's prefetch library concretizes hidden variables before traversing the path constraint tree. In the previous example, the prefetch library would make two calls to `Math.random()`, and one call to `Performance.now()`. With the hidden variables now concretized, and with client-specific values like `User-Agent` in hand, the prefetch library can now traverse the path constraint tree and concretize all of the URLs that reside at the appropriate leaf.

The library prefetches the concretized URLs. Finally, the library dynamically patches [188] nondeterministic functions like `Math.random()` and `Performance.now()`, forcing those methods to return the values in the log of concretized hidden variables. The prefetching library is the first JavaScript code that executes in a page. Thus, as the rest of the page's JavaScript code executes, that code will craft dynamic URLs using the same nondeterministic values that Oblique used to construct prefetched URLs.

This approach may still result in unnatural calculations of elapsed time. For example, a page's normal JavaScript code may call `Performance.now()`, execute a lengthy computation, call `Performance.now()` again, and then use the elapsed time to construct a dynamic URL. If Oblique's prefetching library concretizes the two hidden variables using back-to-back calls to `Performance.now()`, the elapsed time used to influence prefetching will be much smaller than the elaspsed time used by the page's normal JavaScript. At worst, this will cause a wasted prefetch; Oblique only prefetches HTTP GET requests which (unlike POST requests) cannot induce side effects on the server. In future work, we hope to devise mechanisms to allow concolic execution to estimate wall clock time. This ability would enable Oblique to concretize hidden timestamp variables with higher fidelity.

JavaScript is an event-driven language. Thus, the execution order of event handlers (e.g., timers and GUI events) is another source of nondeterminism. Oblique does not attempt to control these sources of randomness, because the event loop only goes live after a page's HTML parse completes. This means that event-loop nondeterminism cannot affect URLs fetched during the HTML parse (e.g., via the `.src` attribute of HTML tags, or `XMLHttpRequests` issued by JavaScript). Event-loop nondeterminism *can* affect URLs fetched after the HTML parse completes.

4.3.4 Analyzing Server-side Behavior

When a web server receives a request for a page's top-level HTML, the server might dynamically construct the returned HTML. For example, the server might inspect the `User-Agent` string in the HTTP request, and return mobile content or desktop content as appropriate. As another example, the server might use the request's cookie to populate the HTML with user-specific URLs, e.g., corresponding to images of a user's previous purchases on an e-commerce site. Oblique's analysis from the previous sections will not detect this potential diversity of embedded URLs. The reason is that the prior analysis assumes that a page has only one version of its top-level HTML, and thus only one set of embedded JavaScript files; if this assumption is true, then the only goal of symbolic

analysis is to explore branch paths in the fixed JavaScript code, identifying the dynamically-fetched URLs.

4.3.5 The Workflow

To generate more accurate prefetch lists for dynamically-generated pages, Oblique can optionally perform symbolic execution of both client-side JavaScript (that runs in a browser) and server-side JavaScript (that runs in the Node framework [182]). The end-to-end workflow looks like this:

- **Phase 1:** Oblique first performs a concolic execution of the server-side request handling code. For each test, the inputs are the HTTP request state, as well as nondeterministic function values (e.g., from Node's `crypto.randomBytes()` method). For each concolic path that is explored, Oblique logs the concrete HTML string that is generated, building a *server-side* path constraint tree. Each leaf contains a concrete HTML string, with each root-to-leaf path representing the constraints on server-side inputs that enable the concrete HTML string to be generated.
- **Phase 2:** Each concrete HTML string is fed to the client-side symbolic execution pipeline from Section 4.3.2. The output of that pipeline is a *client-side* path constraint tree. Each leaf contains symbolic URLs to prefetch, and each root-to-leaf path represents the symbolic constraints on client state that trigger the fetching of the leaf's URLs.
- **Phase 3:** Once Oblique has finished all of the symbolic executions (both client-side and server-side), Oblique creates a "super-constraint tree" which combines the knowledge gleaned from the individual constraint trees. The super tree maps Phase 1 path constraints on server-side inputs to the appropriate client-side path constraint tree from Phase 2; in other words, each leaf in the super tree is a client-side path constraint tree.

When a real client loads the page, the web server uses the values in the HTTP request to traverse the super tree; if the super tree branches on the return values of server-side nondeterministic

function, the web server concretizes those values using the approach from Section 4.3.3. When the server reaches a leaf in the super tree, the server injects the leaf's prefetching library into the dynamically-constructed HTML. The subsequent construction process for the HTML is guided by the values in the HTTP request, and possibly by nondeterministic functions; those functions return the already-concretized values which guided the traversal of the super tree. When the client receives the HTML, Oblique's prefetching library executes as described in Section 4.3.2.

4.3.6 Templating Engines

In Phase 1, Oblique symbolically executes the server-side request handler. A developer has two options for specifying an entry point into request-handling code. First, a developer can register an `http.Server.request` event handler with Oblique. When a client request arrives, Node creates a new `http.IncomingMessage` object and invokes the handler. Oblique uses the object's HTTP headers as test inputs for concolic execution of the handler.

The disadvantage of the prior approach is that, during the construction of dynamic HTML, a server may consult *IO-based* sources of nondeterminism. For example, the server may issue a database query, or send an RPC to an external server. Oblique does not log and replay such IO responses. Thus, the concretized Phase 1 HTML that Phase 2 consumes may be different than the dynamic HTML that is generated at the time of an actual page fetch. Such a mismatch would hurt Oblique's prefetching accuracy.

Oblique can avoid this problem if server-side code uses a template engine to generate dynamic HTML. For example, consider EJS [190], a popular template framework. EJS defines a `render(html, dict)` method. The first argument is a template string (e.g., "<html>Hello {{name}} at {{tstamp}}"). The second argument is a dictionary which maps template arguments to program variables (e.g., `{name: httpReq.cookie.uid, tstamp: Date.now()}`). EJS examines the template and automatically generates a JavaScript program; this program, which

```
------ Server-side JavaScript ------
app.get('/', function(req, res) {
  //...examine req and derive the template parameters,
  //and then...
  res.render('template.ejs',
               {userAgent: req.headers['user-agent'],
                userID: 'alice',
                userName: 'Alice',
                nonce: random_value});
});
----------- template.ejs -----------
<html>
  <head></head>
  <body>
    <p1> Welcome to foo.com, <%= userName %>! </p1>
    <% if (userAgent.includes('Android')) { %>
      <img src='site-logo-mobile.jpg'>
    <% }  else { %>
      <img src='site-logo-desktop.jpg'>
    <% } %>
    <img id='session-<%= nonce %>'
          src='<%= userID %>.jpg'>
  </body>
</html>
```

Figure 19: An example of dynamic HTML generation using EJS templates. EJS directives are shown in bold.

is executed by `render()`, performs the necessary computations to parse `dict` and emit the customized HTML. Figure 19 provides a more complex example of an EJS template.

If a developer uses EJS, then she can tell Oblique to concolically analyze the EJS-created templating JavaScript. The output of Phase 1 is now different: it consists of server-side path constraint trees that are associated with just the templating JavaScript, not the overall handler call chain. Each leaf still contains a concrete HTML string that is passed to the concolic client-side analysis in Phase 2. However, a leaf also contains the *symbolic* HTML string that was output by the Phase 1 analysis. The symbols in this string come from the `dict` argument to `render()`. In the example from Figure 19, the symbolic HTML references the `dict` arguments `userName`, `nonce`, and `userID`. Note that the `dict` argument `userAgent` does not appear in symbolic HTML; that argument is branched upon in the path conditions, but is not directly embedded in the HTML itself.

With template integration, Phase 3 is altered as well. When the web server receives a request, the server executes the request handler up to the invocation of `render()`. At that point, the server has queried any sources of nondeterminism (IO-based or otherwise); the server now possesses concrete values for all the inputs to `render()`. The server can then traverse the super tree, find the appropriate symbolic HTML, concretize it, extract the static URLs inside the concrete HTML, and then inject the appropriate prefetching library. Note that extracting static URLs from the concretized HTML is faster than a naïve top-to-bottom HTML parse, since Oblique has a priori knowledge of the offsets where the URLs will be.

4.3.7 Security Analysis

Oblique's security properties depend on whether symbolic analysis examines only client behavior, or both client and server behavior. Consider the scenario in which Oblique only analyzes client-side activity. In this case, Oblique only requires access to first-party content that is already publicly accessible via first-party web servers. From the perspective of a first-party web server, Oblique's third-party analysis engine looks like a normal end-user browser that issues normal HTTPS fetches. During a concolic page load, Oblique does track symbolic constraints on sensitive user values like cookies and `User-Agent` strings. However, these constraints represent a universe of possible values for sensitive variables; the constraints are insufficiently precise to allow Oblique to determine *the specific sensitive values that belong to a particular user*. For instance, we did empirically find JavaScript code which tested cookies for substrings that were user-agnostic; a common pattern was to inspect a cookie for a string representing the current date. However, JavaScript code did not contain the logical equivalent of a giant regular expression which scanned the local cookie, testing whether the cookie contained any value from an explicit list of valid user ids. Such JavaScript code does not exist because it would allow anyone to download the enclosing JavaScript file and learn all of the valid user ids for a site! Thus, Oblique's symbolic constraints on cookies are insufficient to induce concrete cookie values belonging to specific users. Similarly, if Oblique analyzes a page

and determines that a possible load path will target Android users that possess a certain set of screen dimensions, this information does not allow Oblique to infer the screen dimensions and platform value for a particular user.

To analyze server-side behavior, Oblique requires access to server-side code; that code is inaccessible to public web clients. During the concolic execution of that code, Oblique might also query sensitive databases, or contact sensitive network hosts that are inaccessible to public Internet hosts. Thus, if a developer wants Oblique to analyze both client-side and server-side behavior, Oblique should be run on first-party machines. Compared to Vroom (which is also a first-party accelerator), Oblique will provide faster page loads (§4.6.2).

4.4 Discussion

Oblique is not guaranteed to optimize every object fetch made by every page. For example, during concolic execution, a page's JavaScript may invoke unmodeled native functions, i.e., browser-provided C++ functions for which Oblique lacks a symbolic execution policy (§4.3.2). If concolic execution reaches one of those functions, Oblique must always treat the return value as fully concrete. Doing so will hurt path coverage if the program later branches on the value, since concrete values cannot be "inverted" to force a new branch direction to be explored.

Even if a page avoids unmodeled native functions, path coverage may suffer when symbolic path constraints are difficult to invert. If the constraint solver times out while trying to generate concrete inputs for a new path to explore, the path will not be explored. If this happens, Oblique can miss opportunities to discover prefetchable URLs. We evaluate Oblique's sensitivity to time-out parameters in § 4.6.2.

Oblique's prefetching library can detect when a user's concrete values result in a traversal of the constraint tree that hits an unexplored path. When Oblique operates in first-party mode,

the prefetching library can inform the first party about the concrete inputs which triggered the unexplored path; the first party can then use those inputs to concolically explore the previously-unvisited path. Oblique's client-side library should not share this information with the remote symbolic engine if Oblique runs in third-party mode in a third-party machine, since doing so would leak a user's concrete sensitive values to the untrusted third party.

During concolic execution, Oblique may trigger interactions with external entities. For example, a concolically-executed browser may issue `XMLHttpRequests` to remote servers. Oblique should only be used with pages for which such interactions are idempotent (either literally or for practical purposes). This limitation is shared by all prefetching systems which issue queries to live services to perform content analysis. To identify non-idempotent resources, Oblique's solution is to traverse each path in the finalized path constraint tree twice (by actually loading the page twice). Then, for each prefetched URL along a path, if its contents differ across two page loads for the same path, the URL is regarded to contain a non-idempotent resource and thus its node in the path constraint tree is removed.

4.5 Implementation

To implement Oblique's symbolic analysis, we modified ExpoSE [187]. ExpoSE performs concolic execution of pure JavaScript code, but does not handle environmental interactions like network IO. We modified ExpoSE to interface with two different environmental interfaces: the Node runtime and the Electron [191] HTML renderer. Oblique uses the Node runtime when analyzing server-side code, and uses the Electron runtime when simulating client-side loads. As explained in Section 4.3.2, we enlightened ExpoSE to model a DOM tree symbolically, so that JavaScript-level symbolic values can flow into and out of the DOM interface. Our changes to ExpoSE were non-trivial, totalling roughly 4,300 lines of code.

Oblique's client-side prefetching library is small, containing approximately 300 lines

of Javascript code. When Oblique runs in third-party mode (§4.3.2), web servers require no modifications (other than having to include Oblique's prefetching library at the top of each page's HTML). When Oblique runs in first-party mode (§4.3.4), web servers must be enlightened to traverse the super-constraint tree, concretize nondeterministic values, and interact with Oblique's templating infrastructure. To implement an Oblique-compatible web server, we created a front-end HTTP layer that sat in front of a commodity web server. The front-end layer used the nghttp2 HTTP library [192] and the myhtml HTML parser [193] to implement the activities described above.

4.6 Evaluation

In this section, we compare Oblique's performance to that of Vroom and RDR, two state-of-the-art accelerators for mobile page loads. Our evaluation primarily focuses on the variant of Oblique that only analyzes client-side behavior, since we can evaluate this variant on a large number of commercial sites. Using a corpus of 200 real pages, we find that Oblique reduces page loads by up to 31%, outperforming Vroom and RDR by up to 17% while also reducing VM costs for popular sites (§4.6.2). Oblique provides these advantages while also enabling secure outsourcing of prefetch analysis (§4.6.2). In Section 4.6.3, we use a site that we control to provide a case study of the benefits of analyzing both client-side behavior and server-side behavior. We demonstrate that, if first parties are willing to run Oblique, they can unlock even greater reductions in load time than what client-only analysis provides.

4.6.1 Methodology

Our experiments used a Galaxy S10e phone that ran Chromium v78. The browser ran atop Linux on Dex [194], a runtime that enables Samsung phones to execute traditional Linux executables; Linux on Dex made it easier for us to write testing scripts and other experimental infrastructure. We automated the initiation of page loads and the collection of load time metrics using the

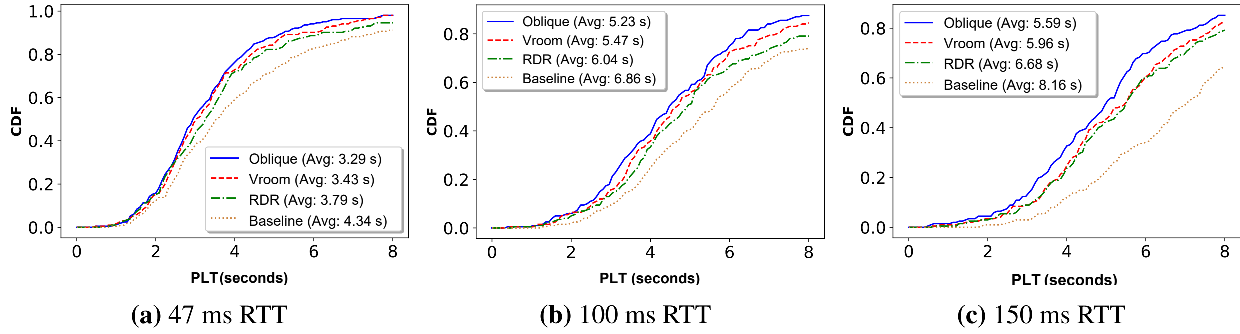
Browsertime [195] library. Internally, Browsertime manipulated Chrome via Selenium's WebDriver APIs [196, 197].

To test Oblique, Vroom, and RDR with real websites, we built a Mahimahi-style tool [32] to record the objects in live web pages. Afterwards, when our test phone sent an HTTP request to an Oblique web server, a Vroom web server, or an RDR proxy, the web server or proxy responded with recorded content if the request hit in the replay cache; otherwise, the web server or proxy issued a live fetch to the appropriate content server. We ran Oblique and Vroom web servers, and RDR proxies, on a Digital Ocean VM with 8 2.3 GHz cores, 16 GB of RAM, and a 2 Gbps NIC. The RDR proxy used headless Chrome [198] to load pages. Vroom's offline analysis also used headless Chrome. The online Vroom web server was a derivative of nghttp2 [192] that used MyHTML [193] and Katana [199] to parse HTML and CSS.

Our phone had an LTE connection with a round-trip time of 47 ms to our Digital Ocean VM. Our test corpus contained 200 pages from the Majestic Million [200]. We selected the 200 most popular pages for which the RTT between our phone and a page's web server was less than the the RTT between our phone and our Digital Ocean VM. This setup resulted in conservative estimates of the benefits provided by Oblique, Vroom, and RDR, relative to the baseline scenario in which our phone contacted normal web servers directly. For each combination of <page, load time metric, acceleration technique>, we loaded each page 5 times and recorded the average. By default, Oblique and Vroom pages were loaded one hour after the completion of offline analysis, but we perform sensitivity analysis on this parameter in Section 4.6.2.

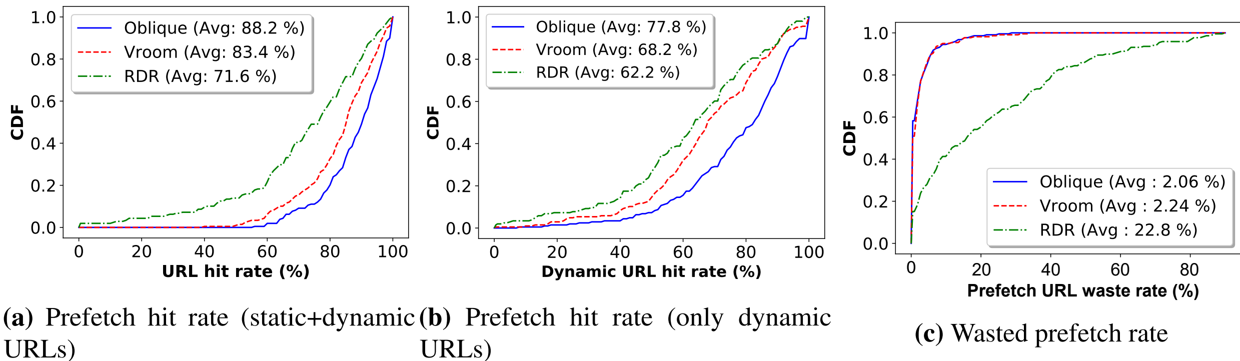
4.6.2 Client-only Analysis

PLT: We first explored Oblique's performance when only the client side of a page load is analyzed. Figure 20 shows results for the page load time (PLT) metric. PLT, as measured by the time to the browser's `onload` event, captures how long a page needs to fetch and evaluate all objects referenced



(a) 47 ms RTT **(b)** 100 ms RTT **(c)** 150 ms RTT

Figure 20: Cold-cache PLTs for Oblique, Vroom, RDR, and a baseline, non-accelerated browser.



(a) Prefetch hit rate (static+dynamic URLs) **(b)** Prefetch hit rate (only dynamic URLs) **(c)** Wasted prefetch rate

Figure 21: Prefetch efficiency for Oblique, Vroom, and RDR.

by a page's static HTML. Note that PLT only waits for some dynamically-generated fetches to complete. In particular, PLT waits for fetches triggered by the insertion of new DOM nodes (e.g., `document.body.appendChild(newImg)`), but not for fetches triggered directly by network APIs like `fetch(url)`. Thus, PLT underestimates the extent to which Oblique, Vroom, and RDR reduce overall fetch latencies for a page.

Figure 20a shows that, for a 47 ms RTT and a cold browser cache, Oblique provided the average page with a 24.1% reduction in PLT, relative to a baseline (i.e., non-accelerated) page load. Oblique reduced PLTs by 17.3% more than RDR, and 5.4% more than Vroom. To explore Oblique's benefits with higher RTTs; we connected the smartphone to a desktop machine via WiFi, and used netem [101] to inject additional latency along the smartphone/desktop link. As expected, Oblique's benefits improved as phone-server RTTs grew, because of the increasing value of hiding last-mile latency. For example, Figure 20c shows PLT results for an emulated RTT of 150 ms. Oblique improved the average baseline PLT by 31.4%, outperforming RDR by 16.3% and Vroom

	Precision	Recall
Oblique	0.882	0.979
Vroom	0.834	0.978
RDR	0.716	0.772

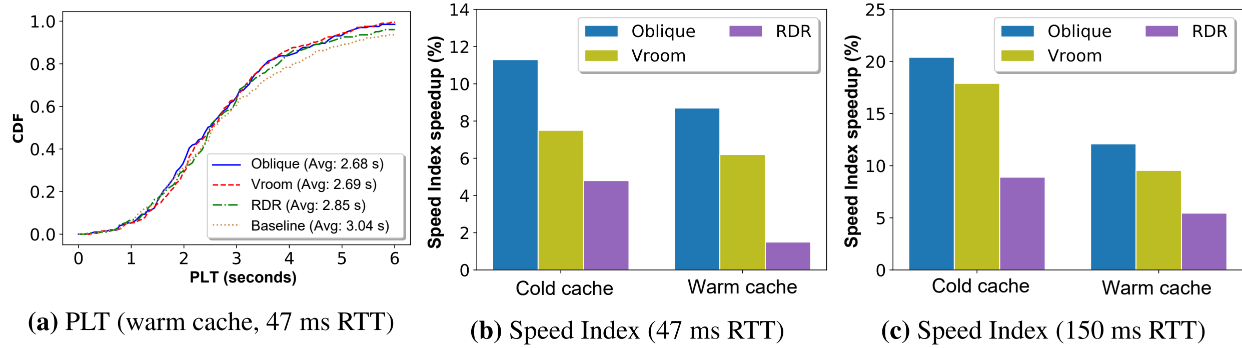
Table 9: Average precision and recall for URL hit rates.

by 6.2%. Table 9 reports the average precision and recall for URL hit rates for Oblique, Vroom, and RDR. Oblique shows the highest precision (0.882) and recall (0.979) out of three systems.

A page load's *prefetch hit rate* is the fraction of requested objects that hit in the browser cache due to a successful prefetch. As shown in Figure 21a, Oblique enjoyed better prefetch hit rates than both Vroom and RDR. Indeed, Oblique's primary advantage over Vroom was the ability to successfully prefetch dynamic URLs that embedded nondeterministic symbols (§4.3.3); this advantage is reflected in Figure 21b.

We define a page load's *wasted prefetch rate* as the fraction of prefetched objects that were never requested during the page load. Figure 21c demonstrates that RDR has a much higher percentage of wasted prefetches. The reason is that, for each client-initiated page load, RDR loads the page twice: once on the proxy, and once on the real client machine. Both client-side and server-side nondeterminism may cause the URLs fetched by the proxy's page load to be different than the URLs fetched by the client's browser. Oblique avoids this problem by handling nondeterministic URLs symbolically. In contrast, Vroom's stable-set algorithm simply filters out many nondeterministic URLs. Thus, Vroom has fewer wasted prefetches than RDR, because Vroom does not prefetch nondeterministic URLs that RDR erroneously pulls; however, as shown in Figure 21b, Vroom has a worse hit rate than Oblique due to worse handling of nondeterministic dynamic URLs.

In comparison to RDR, both Oblique and Vroom benefited from informing clients early about the URLs to prefetch. For example, Oblique discovered all of these URLs offline, and prefetched them via the first JavaScript code that executed on a page. Vroom included `<link>`



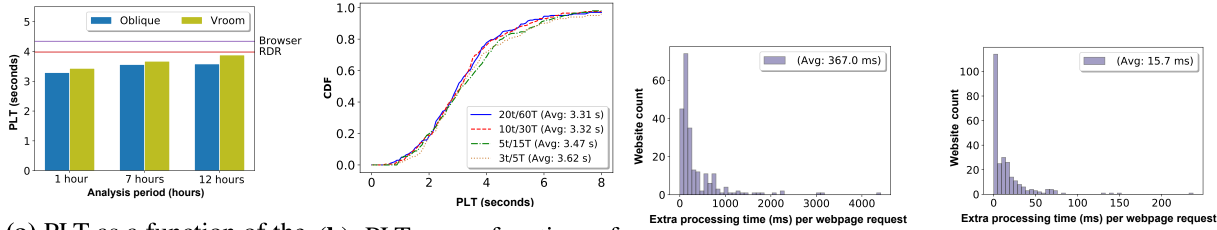
(a) PLT (warm cache, 47 ms RTT) **(b)** Speed Index (47 ms RTT) **(c)** Speed Index (150 ms RTT)

Figure 22: Warm-cache PLTs and Speed Indices (both cold and warm caches). Note that subfigures (b) and (c) have different y-axis scales.

preload tags at the beginning of a page's HTML, and server-pushed other objects to prefetch. In contrast, RDR streamed objects to a client as the proxy discovered those objects; the deeper a page's dependency graph was (§4.2), the larger the comparative advantage provided by Oblique offline discovery approach. Vroom discovered some prefetch URLs offline, and others during the online, server-side HTML parse. However, Vroom aggressively notified clients about the offline-discovered URLs using server push and `<link>` preload tags.

Warm caches: Figure 22a depicts PLTs for all four systems when browser caches were warm. As expected, all systems enjoyed lower PLTs. Oblique and Vroom had similar performance, but still outperformed RDR.

Speed Index: We also evaluated the ability of Oblique, Vroom, and RDR to improve a page's Speed Index [201]. Speed Index is a visual metric that represents how quickly a page's above-the-fold content is rendered. A page's Speed Index is $\int_0^{end} 1 - \frac{p(t)}{100}\,dt$, where *end* is the time of the last pixel change, and $p(t)$ is the percentage of pixels that have already received their final value; lower Speed Indices are better. The formula rewards page loads whose overall rendering time is fast (meaning that *end* values are small). Given two pages with the same *end* value, the formula rewards the page which renders more pixels earlier. Note that Speed Index ignores whether JavaScript files or below-the-fold content has arrived; thus, like PLT, Speed Index underestimates the extent to



(a) PLT as a function of the staleness of the offline analytic data. These results use a 47 ms RTT and cold caches.

(b) PLT as a function of the time budget given to Oblique's offline symbolic analysis. These results use a 47 ms RTT and cold caches.

(c) RDR: Per-page-load computational time required by a proxy.

(d) Vroom: Additional per-page-load computational time required.

Figure 23: The impact of stale Oblique/Vroom analyses, and the additional computational overheads of RDR and Vroom.

which accelerators have successfully prepositioned objects.

Figures 22b and 22c show Speed Index results for RTTs of 47 ms and 150 ms. The basic trend is the same one observed for PLT: Oblique has better performance than Vroom, and Vroom has better performance than RDR. However, all of the acceleration systems improve PLT more than Speed Index. For example, with cold caches and a 150 ms RTT, Oblique reduces PLT by 31.4%, but Speed Index by only 20.4%. The reason is that Speed Index only considers visual content, and only cares about the loading of JavaScript files to the extent that the evaluated code modifies a page's above-the-fold graphics. However, deep chains in a page's dependency graph are often caused by JavaScript files whose evaluation triggers the loading of additional JavaScript files [163]. All three accelerators let clients resolve those dependency chains more quickly, but this has less impact on Speed Index than PLT.

Stale analytic results: In Figures 20 and 22, the offline analyses for Oblique and Vroom occurred one hour before a page load. Figure 23a depicts average PLTs when Oblique and Vroom used analytic data from farther in the pass. Unsurprisingly, Oblique and Vroom performed better with more recent analytic data. However, for up to 12 hours of staleness, Oblique maintained its advantage over Vroom; both Oblique and Vroom also maintained their advantages over RDR and a non-accelerated browser.

Additional analysis time: Given an infinite amount of time, Oblique's offline analysis would be guaranteed to find a complete tree of path constraints; in other words, every possible concretization of client-side symbols would be covered by some root-to-leaf tree path. In practice, Oblique's symbolic analysis is constrained by two parameters: t represents the maximum execution time for a particular execution path,[3] and T represents the overall amount of time that Oblique will analyze the page. By default, Oblique uses $t = 10$ minutes and $T = 30$ minutes. If fully exploring a particular path requires more time than t, Oblique will only discover a subset of the URLs associated with the path. If discovering all paths in a page takes longer than T, Oblique will not generate a prefetch list for the undiscovered paths.

Figure 23b depicts Oblique's PLT benefits for different values of t and T. In those experiments, the PLT for a page was defined as the average PLT across all discovered paths; to test the PLT for a particular discovered path, our test browser used concretized client-side symbols that triggered the path. Figure 23b shows that Oblique is basically insensitive to t values above 10 minutes and T values above 30 minutes. The reason is that, for the average page in our test corpus, only 7 minutes were needed to completely explore a path; furthermore, the median page only contained 7 execution paths.

Economic costs: For a given version of a page, Oblique performs an offline analysis once, constructs a path constraint tree, and then incurs no online costs during a real client load. In contrast, RDR must launch an RDR proxy for each client load, and Vroom must perform online HTML parsing. Figures 23c and 23d depict those per-page-load CPU costs.

A VM owner pays for a virtual CPU by the second or by the hour. Once a virtual CPU is fully loaded, any additional computation to perform will force the VM owner to rent more virtual CPU seconds. For a fully-loaded virtual CPU, Vroom requires a VM owner to pay for an additional 15.7 ms of additional compute time per page load. Thus, Oblique's offline analysis

[3]A timeout occurs when the SMT solver cannot negate a branch condition in the current path (§4.3.1).

becomes cheaper than Vroom's smaller (but repeated) online costs after $T/15.7$ page loads, where T is Oblique's offline analysis time in units of milliseconds. For example, with a T of 30 minutes, Oblique becomes financially cheaper after 114,650 page loads; with a T of an hour, Oblique becomes cheaper after 229,299 loads. Since RDR imposes much heavier computational overheads than Vroom, Oblique becomes cheaper much faster—after 4,904 loads or 9,809 loads for a T of 30 minutes or a T of an hour. Importantly, these estimates assume that, when a page changes, Oblique's prior analysis is totally invalidated. We are currently investigating how Oblique can use incremental symbolic execution [202] to amortize our analysis costs even more aggressively.

4.6.3 Oblique in First-party Mode

When Oblique runs on first-party infrastructure, Oblique can symbolically evaluate client-side and server-side behavior. However, to do this, Oblique must be able to examine back-end code. We had no access to server-side code for the commercial sites in our test corpus; thus, we had to evaluate first-party Oblique on a collection of modified open-source sites that we ran ourselves. We focus on a single case study of an open-source EJS site. In the text below, Oblique-C refers to a setup in which Oblique can only analyze client-side activity. Oblique-SC refers to a setup in which Oblique can evaluate both server and client behavior.

Gallery Viewer [203] is a site whose core functionality is displaying a rotating set of images. Each image is associated with metadata like an author, a category (e.g., "nature scenes"), and a description of the image; metadata is stored in on-disk JSON files. Users can also chat with each other in real time, and submit comments on particular images. From the perspective of Oblique, the site is interesting because of how it uses cookies and random number generators. The site assigns a unique cookie to each user. When a user requests the page's top-level HTML, the server uses the cookie to query a server-side table of user preferences. The table indicates the types of images that a user likes to view. Given those preferences, the server leverages a random number

generator to select random images from the user's preferred image categories. The server inserts the associated image URLs into the dynamically-generated HTML that is returned to the user's browser.

For this particular site, no client-side symbols are relevant to prefetching. However, two kinds of server-side symbols are relevant: the cookie value in the HTTP request for the top-level HTML, and the random numbers that are used to select image URLs.

- Oblique-SC correctly prefetches all of the image URLs. During Phase 1 of analysis (§4.3.4), Oblique-SC symbolically evaluates the templating JavaScript, creating a symbolic HTML string. In Phase 3, i.e., during a real page load, Oblique-SC runs the server-side event handler up to the call to `render()`. At that point, Oblique-SC concretizes the symbolic HTML using the live cookie data and logged values from the random number generator. Oblique-SC then extracts the image URLs from the concretized HTML, and creates a prefetch library that downloads those URLs.
- Oblique-C lacks visibility into server-side behavior. Thus, an Oblique-C client incorrectly prefetches the URLs in the concretized HTML that was seen during offline analysis.
- Vroom correctly prefetches the image URLs; the Vroom web server identifies the URLs during the on-the-fly HTML parse.
- RDR incorrectly prefetches the image URLs. The HTML returned to the proxy's headless browser will contain different URLs than the ones in the HTML returned to the user's browser; the URLs in the first HTML file are prefetched by the client.

For a cold browser cache and a 47 ms RTT, Oblique-SC and Vroom had similar performance, with PLTs of 2.01 seconds and 2.06 seconds, respectively. Oblique-C did only slightly better than RDR (2.29 seconds versus 2.37 seconds). The non-accelerated page load required 2.76 seconds.

Oblique-SC has larger computational costs than Oblique-C; during offline analysis, more

symbolic execution is required, and during an actual page load, web servers must participate in Phase 3 activity. The extent to which Oblique-SC is preferable to Oblique-C depends on whether first parties want to pay these costs, and the extent to which a site uses server-side symbols to generate HTML. However, the results from Section 4.6.2 demonstrate that Oblique-C alone can provide impressive reductions in page load time.

4.7 Conclusion

Oblique is a new system for accelerating mobile page loads. Oblique uses symbolic execution to analyze the various ways that a page load could proceed. For each potential outcome, Oblique creates a list of symbolic URLs that the corresponding page load would fetch. These URLs are concretized at the time of an actual page load, and then prefetched using Oblique's client-side JavaScript library. Oblique works on unmodified browsers, and provides faster page loads than current state-of-the-art approaches. When run in third-party mode, Oblique enables secure outsourcing of prefetch analysis while also enabling reductions in VM costs.

5 Conclusion

This dissertation explained how to improve the security and privacy of user data in modern Internet services. The dissertation examined three specific domains: client-side IoT deployments, server-side application stacks, and middlebox acceleration proxies for HTTPS traffic. Each domain presented distinct challenges, resulting in three distinct platforms for protecting user data: DeadBolt, Riverbed, and Oblique.

DeadBolt makes IoT deployments more resistant to attacks. A DeadBolt AP quarantines devices that run untrusted or out-of-date software, and uses traditional firewall techniques to prevent external attackers from probing for target IoT devices. Virtual drivers allow an AP to safely permit otherwise insecure lightweight devices to communicate with other parties. The AP forces heavyweight devices to remotely attest their software stacks, and to use periodic code randomization to thwart control flow exploits. To reduce the application downtime that is required to patch a heavyweight device, DeadBolt uses a VM swapping mechanism to overlap patching with normal application execution. Our evaluation shows that a DeadBolt AP can be efficiently implemented on a $90 Minnowboard; furthermore, device-side techniques like code shuffling and VM-based patching impose modest overheads. Thus, we believe that DeadBolt is a practical way to improve the security of IoT deployments.

The Riverbed platform allows developers to create web services that respect user-defined privacy policies. Each universe in Riverbed stores user data which has the same privacy policy. Within a universe, Riverbed uses taint tracking to prevent user data (or its derivatives) from flowing to disallowed sinks. This approach means that developers are not required to annotate their source code or reason about security lattices. A transparent Riverbed proxy sits between a server and an unmodified client; the proxy only reveals sensitive user data to a server only after the server has attested to running Riverbed's taint tracking runtime. Experiments with realistic applications demonstrate that Riverbed imposes a modest performance overhead, while demonstrating to users

and developers that sensitive user data is being handled according to its associated policy.

Oblique is a web accelerator that reduces page load times for HTTPS sites. Oblique allows the prefetching analysis to be outsourced to third parties without revealing cleartext HTTPS data to the analysis engine. To do so, Oblique loads a page symbolically on a third-party server. This offline analysis determines all possible objects that a page might load, representing those objects using symbolic URLs; the symbols in the URLs are sensitive client-side state like cookie values. During an actual page load, the client browser concretizes the symbolic URLs and prefetches them before they are discovered by the HTML parse. Evaluation results demonstrate that Oblique reduces page load time by 31%, and outperforms Vroom and RDR by up to 17%.

References

[1] Miniwatts Marketing Group, "World Internet Users and 2020 Population Stats." https://www.internetworldstats.com/stats.htm, 2020.

[2] Statista 2021, "Daily Time Spent with the Internet per Capital Worldwide from 2011 to 2021, By Device." https://www.statista.com/statistics/319732/daily-time-spent-online-device/, 2021.

[3] D. Bregman, "Smart Home Intelligence - The eHome That Learns," vol. 4, pp. 35–46, 2010.

[4] M. A. Zamora-Izquierdo, J. Santa, and A. F. Gomez-Skarmeta, "An Integral and Networked Home Automation Solution for Indoor Ambient Intelligence," *IEEE Pervasive Computing*, vol. 9, no. 4, pp. 66–77, 2010.

[5] J. Rosslin, R. , and T.-H. Kim, "Review: Context Aware Tools for Smart Home Development," *International Journal of Smart Home*, vol. 4, 2010.

[6] A. Zanella, "Internet of Things for Smart Cities," *IEEE Internet of Things*, vol. 1, no. 1, 2014.

[7] J. Lee, "Smart Factory Systems," *Informatik-Spektrum*, vol. 38, pp. 230–235, 2015.

[8] E. Sapir, "The Dawn of the World," *Science*, vol. 32, pp. 557–558, 1910.

[9] Spec Sensors, "High Performance Gas Sensors." https://www.spec-sensors.com/, 2021.

[10] Statista, "Forecast End-user Spending on IoT Solutions Worldwide from 2017 to 2025." https://www.statista.com/statistics/976313/global-iot-market-size, 2021.

[11] F. Brasser, B. E. Mahjoub, A. R. Sadeghi, C. Wachsmann, and P. Koeberl, “Tytan: Tiny trust anchor for tiny devices,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[12] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pp. 479–498, USENIX, 2013.

[13] C. Cimpanu, “Configure Commonly Used IP ACLs.” `https://www.cisco.com/c/en/us/support/docs/ip/access-lists/26448-ACLsamples.html`, 2020.

[14] Cisco, “Snort.” `https://www.snort.org/`.

[15] A. K. Simpson, F. Roesner, and T. Kohno, “Securing Vulnerable Home IoT Devices with an In-hub Security Manager,” in *Proceedings of the 2017 IEEE International Conference on Pervasive Computing and Communications Workshops*, pp. 551–556, IEEE, 2017.

[16] EU Parliament, “GDPR Portal.” `http://www.eugdpr.org/eugdpr.org.html`, 2018.

[17] California State Legislature, “California Legislative Information.” `https://leginfo.legislature.ca.gov`, 2018.

[18] Renato Leite Monteiro, “The New Brazilian General Data Protection Law – A Detailed Analysis.” `https://iapp.org/news/a/the-new-brazilian-general-data-protection-law-a-detailed-analysis/`, 2019.

[19] Australian Government, “Privacy Amendment.” `https://www.legislation.gov.au/Details/C2017A00012`, 2017.

[20] OneTrust Data Guidance, “Act on the Protection of Personal Information (APPI).” `https://free.dataguidance.com/laws/japan-act-on-the-protection-of-personal- information-2/`, 2017.

[21] Inside Privacy, “Thailand Adopts Personal Data Protection Act.” `https://www.insideprivacy.com/data-privacy/thailand-passes-personal-data-protection-act/`, 2019.

[22] A. Myers and B. Liskov, “Protecting Privacy Using the Decentralized Label Model,” *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, 2000.

[23] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. Myers, “Fabric: A Platform for Secure Distributed Computation and Storage,” in *Proceedings of the 22nd Symposium on Operating System Principles*, pp. 321–334, ACM, 2009.

[24] S. Chong, K. Vikram, and A. Myers, “SIF: Enforcing Confidentiality and Integrity in Web Applications,” in *Proceedings of the 16th USENIX Security Symposium*, USENIX, 2007.

[25] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo, "Hails: Protecting Data Privacy in Untrusted Web Applications," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pp. 47–60, USENIX, 2012.

[26] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving Application Security with Data Flow Assertions," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 291–304, ACM, 2009.

[27] A. Zdancewic, L. Zheng, N. Nystrom, and A. Myers, "Untrusted Hosts and Confidentiality: Secure Program Partitioning," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 1–14, ACM, 2001.

[28] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel, "Thoth: Comprehensive Policy Compliance in Data Retrieval Systems," in *Proceedings of the 25th USENIX Security Symposium*, pp. 637–654, USENIX, 2016.

[29] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, ACM, 2007.

[30] T. Pasquier, J. Singh, J. Bacon, and D. Eyers, "Information Flow Audit for PaaS Clouds," in *Proceedings of the IEEE International Conference on Cloud Engineering*, pp. 42–51, IEEE, 2016.

[31] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, "Securing Distributed Systems with Information Flow Control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 293–308, USENIX, 2008.

[32] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate Record-and-Replay for HTTP," in *Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference*, p. 417–429, USENIX, 2015.

[33] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan, "WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, p. 430–443, ACM, 2019.

[34] Amazon, "What is Amazon Silk?." `https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html`, 2020.

[35] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen, "PARCEL: Proxy Assisted Browsing in Cellular Networks for Energy and Latency Reduction," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, p. 325–336, ACM, 2014.

[36] Opera Norway, "Opera Mini." `https://www.opera.com/mobile/mini/android`, 2020.

[37] D. Bhattacherjee, M. Tirmazi, and A. Singla, “A Cloud-based Content Gathering Network,” in *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, p. 17, USENIX, 2017.

[38] R. Chirgwin, “Finns Chilling as DDoS Knocks out Building Control System.” https://www.theregister.co.uk/2016/11/09/finns_chilling_as_ddos_knocks_out_building_control_system/, 2016.

[39] L. Dignan, “Dyn Confirms Mirai Botnet Involved in Distributed Denial of Service Attack.” http://www.zdnet.com/article/dyn-confirms-mirai-botnet-involved-in-distributed-denial-of service-attack/, 2016.

[40] Z. Whittaker, “Homeland Security Warns of ’BrickerBot’ Malware That Destroys Unsecured Internet-connected Devices.” http://www.zdnet.com/article/homeland-security-warns-of-brickerbot-malware-that-destroys-unsecured-internet-connected-devices/, 2017.

[41] Data Breach Digest, “IoT Calamity: The Panda Monium.” http://www.verizonenterprise.com/resources/reports/rp_data-breach-digest-2017-sneak-peek_xg_en.pdf, 2017.

[42] A. Greenberg, “Hackers Remotely Kill a Jeep on the Highway.” https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, 2017.

[43] J. J. Roberts, “Light Bulbs Flash "SOS" in Scary Internet of Things Attack.” http://fortune.com/2016/11/03/light-bulb-hacking/, 2016.

[44] Wikipedia Community, “Automatic Fire Suppression.” https://en.wikipedia.org/wiki/Automatic_fire_suppression, 2021.

[45] Dierks, T. and Rescorla,E., “The Transport Layer Security (TLS) Protocol Version 1.2.” https://tools.ietf.org/html/rfc5246/, 2008.

[46] Tom Gaffney, “Is Firmware Kryptonite for Routers and the IoT?.” https://www.cedmagazine.com/article/2016/02/firmware-kryptonite-routers-and-iot/, 2016.

[47] J. Wallen, “Five Nightmarish Attacks That Show the Risks of IoT Security.” http://www.zdnet.com/article/5-nightmarish-attacks-that-show-the-risks-of-iot-security/, 2017.

[48] Trusted Computing Group, “Trusted Platform Module (TPM) Summary.” https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/, 2008.

[49] B. Parno, J. McCune, and A. Perrig, "Bootstrapping Trust in Modern Computers." https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/BootstrappingTrustBook.pdf, 2011.

[50] Trusted Computing Group, "TCG Infrastructure Working Group Architecture Part II: Integrity Management." https://trustedcomputinggroup.org/wp-content/uploads/IWG_ArchitecturePartII_v1.0.pdf, 2006.

[51] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *Proceedings of DARPA Information Survivability Conference and Exposition*, vol. 2, pp. 119–129, IEEE, 2000.

[52] J. C. Foster, V. Osipov, N. Bhalla, N. Heinen, and D. Aitel, "Chapter 1 - Buffer Overflows: The Essentials," in *Buffer Overflow Attacks*, pp. 3 – 23, Syngress, 2005.

[53] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 559–572, ACM, 2010.

[54] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, 2012.

[55] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, ACM, 2015.

[56] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pp. 298–307, ACM, 2004.

[57] H. M. Gisbert and I. Ripoll, "On the Effectiveness of NX, SSP, RenewSSP, and ASLR Against Stack Buffer Overflows," in *Proceedings of the 13th IEEE International Symposium on Network Computing and Applications*, pp. 145–152, IEEE, 2014.

[58] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 367–382, USENIX, 2016.

[59] CRIU Community, "CRIU: A Project to Implement Checkpoint/Restore Functionality for Linux." http://criu.org, 2021.

[60] Y. Diogenes, "Internet of Things Security Architecture." Microsoft Azure documentation. https://azure.microsoft.com/en-us/documentation/articles/iot-security-architecture/, August 2, 2016.

[61] SmartThings, "SmartofThings Developer Documentation." http://docs.smartthings.com/en/latest/device-type-developers-guide/overview.html/, 2021.

[62] Wink Labs Inc., "Wink Hub." https://www.wink.com/products/wink-hub/, 2020.

[63] The Chamberlain Group Inc., "Chamberlain Internet Gateway User's Guide." https://www.chamberlain.com/CatalogResourcesV3/en-us/shared/files/tucmanuals/114A4608.pdf, 2013.

[64] Astrium DigitalGlobe, "Android Auto." https://www.android.com/auto/, 2021.

[65] Logitech, "Harmony Hub." https://www.logitech.com/en-us/product/harmony-hub, 2021.

[66] Congatec, "Spec Sheet: Conga IA4." http://www.congatec.com/us/products/mini-itx/conga-ia4.html, 2016.

[67] TQ Group GmbH, "COM Express Mini Module (Type 10) Key Functionalities." https://www.tq-group.com/en/products/product-details/prod/embedded-modul-tqmxe38m/extb/Main/productdetail/, 2021.

[68] Infineon Technologies AG, "Protecting Integrity and Authenticity of Embedded Devices and Systems." https://www.infineon.com/cms/en/product/security-and-smart-card-solutions/optiga-embedded-security-solutions/optiga-tpm/channel.html?channel=5546d462503812bb015066de24291768&redirId=27019, 2019.

[69] STMicroelectronics, "Trusted Platform Module with LPC Interface Based on 32-bit ARM SecurCore SC300 CPU." http://www.st.com/en/secure-mcus/st33tpm12lpc.html, 2021.

[70] MDSec, "Building an IoT Botnet: BSides Manchester 2016." https://www.mdsec.co.uk/2016/10/building-an-iot-botnet-bsides-manchester-2016/, 2016.

[71] Ken Munro, "What Did Mirai Miss? Making a Better, Bigger Botnet." https://www.pentestpartners.com/security-blog/what-did-mirai-miss-making-a-better-bigger-botnet/, 2021.

[72] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 227–242, IEEE, 2014.

[73] Senrio, "400,000 Publicly Available IoT Devices Vulnerable to Single Flaw." http://blog.senr.io/blog/400000-publicly-available-iot-devices-vulnerable-to-single-flaw/, 2016.

[74] VN Security, “Analysis of Nginx 1.3.9/1.4.0 Stack Buffer Overflow and x64 Exploitation .” http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html, 2013.

[75] OpenCV team, “Multiple Vulnerabilities in OpenCV.” https://github.com/opencv/opencv/issues/4550, 2015.

[76] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352, USENIX, 2013.

[77] MSDN Microsoft, “Control Flow Guard.” https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2018.

[78] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: a Sandbox for Portable, Untrusted x86 Native Code,” in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 79–93, IEEE, 2009.

[79] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-FLAT: Control-Flow Attestation for Embedded Systems Software,” in *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, pp. 743–754, ACM, 2016.

[80] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. A. Gunter, X. Zhou, and M. Grace, “Guardian of the HAN: Thwarting Mobile Attacks on Smart-Home Devices Using OS-level Situation Awareness,” *CoRR*, vol. abs/1703.01537, 2017.

[81] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, “FlowFence: Practical Data Protection for Emerging IoT Application Frameworks,” in *Proceedings of the 25th USENIX Security Symposium*, pp. 531–548, USENIX, 2016.

[82] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, “An Operating System for the Home,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pp. 25–25, USENIX, 2012.

[83] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, “Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pp. 5:1–5:7, ACM, 2015.

[84] Trusted Computing Group, “TCG PC Client Platform Firmware Profile Specification.” https://www.trustedcomputinggroup.org/wp-content/uploads/PC-ClientSpecific_Platform_Profile_for_TPM_2p0_Systems_v21_Public-Review.pdf, 2015.

[85] F. Wang, Y. Joung, and J. Mickens, “Cobweb: Practical Remote Attestation Using Contextual Graphs,” in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, p. 1–7, ACM, 2017.

[86] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX, 2006.

[87] The Netfilter's webmasters, "The netfilter.org Project." https://www.netfilter.org/, 2021.

[88] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *Proceedings of the 22Nd ACM Conference on Computer and Communications Security*, pp. 941–951, ACM, 2015.

[89] XEN project, "XEN - Unstable Development Version." http://xenbits.xensource.com/xen-unstable.hg, 2021.

[90] MinnowBoard.org Foundation, "Minnowboard." https://minnowboard.org/, 2021.

[91] J. Malinen, "HostAP." https://w1.fi/, 2013.

[92] J. Malinen, "Linux WPA/WPA2/IEEE 802.1X Supplicant." https://w1.fi/wpa_supplicant/, 2013.

[93] The FreeRADIUS Server Project, "FreeRADIUS Server." https://freeradius.org/, 2018.

[94] K. Goldman, "IBM's TPM 2.0 TSS." https://sourceforge.net/projects/ibmtpm20tss/, 2021.

[95] K. Goldman, "IBM ACS Library." https://sourceforge.net/projects/ibmtpm20acs/, 2021.

[96] S. Kelley, "DNSMasq: a Lightweight, Caching DNS Proxy with Integrated DHCP Server." http://www.thekelleys.org.uk/dnsmasq/, 2021.

[97] M. Garret, "GRUB2 with TPM2 Support." https://github.com/mjg59/grub, 2016.

[98] Linux, "Integrity Measurement Architecture." https://sourceforge.net/p/linux-ima/wiki/Home/, 2017.

[99] A. Kopytov, "sysbench: A Modular, Cross-platform and Multi-threaded Benchmark Tool." https://man.cx/sysbench(1), 2021.

[100] The Apache Foundation, "Apache Benchmark." https://httpd.apache.org/docs/2.4/programs/ab.html, 2021.

[101] F. Ludovici and H. P. Pfeifer, "NetEm - Network Emulator." http://man7.org/linux/man-pages/man8/tc-netem.8.html, 2011.

[102] Standard Performance Evaluation Corporation, "SPEC CPU 2006." https://www.spec.org/cpu2006/, 2018.

[103] Shodan, “Shodan IoT Search Engine.” `https://www.shodan.io/`, 2021.

[104] D. Hedin and A. Sabelfeld, “A Perspective on Information-Flow Control,” in *Proceedings of the Marktoberdorf Summer School*, 2011.

[105] P. Li, Y. Mao, and S. Zdancewic, “Information Integrity Policies,” in *Proceedings of the 7th International Workshop on Formal Aspects in Security and Trust*, Springer, 2003.

[106] A. Ronacher, “Minitwit.” `https://github.com/martyanov/minitwit`, 2012.

[107] F. Primerano, “Ionic Backup.” `https://github.com/Max00355/IonicBackup`, 2013.

[108] A. Lindsay, “Thrifty P2P.” `https://github.com/atl/thrifty-p2p`, 2009.

[109] S. Sen, S. Guha, A. Datta, S. Rajamani, J. Tsai, and J. Wing, “Bootstrapping Privacy Compliance in Big Data Systems,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 327–342, IEEE, 2014.

[110] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley, “Towards Practical Taint Tracking,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.

[111] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask),” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 317–331, IEEE, 2010.

[112] A. Slowinska and H. Bos, “Pointless Tainting? Evaluating the Practicality of Pointer Tainting,” in *Proceedings of the 4th ACM European Conference on Computer Systems*, pp. 61–74, ACM, 2009.

[113] T. Austin, J. Yang, and C. F. A. Solar-Lezama, “Faceted Execution of Policy-agnostic Programs,” in *Proceedings of the 2013 SIGPLAN Workshop on Programming Languages and Analysis for Security*, pp. 15–26, ACM, 2013.

[114] J. Yang, K. Yessenov, and A. Solar-Lezama, “A Language for Automatically Enforcing Privacy Policies,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 47, pp. 85–96, ACM, 2012.

[115] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pp. 1–6, USENIX, 2010.

[116] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, “Ironclad Apps: End-to-End Security via Automated Full-System Verification,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pp. 165–181, OSDI, 2014.

[117] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, p. 348–370, Springer-Verlag, 2010.

[118] Lyon Brothers Enterprises, "Turtl: Find Your Private Space." `https://turtlapp.com/`, 2021.

[119] Apache Software Foundation, "Apache SpamAssassin: Open-source Spam Filter." `http://spamassassin.apache.org/`, 2021.

[120] Google, "What is Safe Browsing?." `https://developers.google.com/safe-browsing/`, 2021.

[121] J. Corpuz, "Best Ad Blockers and Privacy Extensions: Chrome, Safari, Firefox, and IE." `https://www.tomsguide.com/us/pictures-story/565-best-adblockers-privacy-extensions.html`, 2021.

[122] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Proceedings of the 15th Conference on USENIX Security Symposium*, pp. 305–320, 2006.

[123] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, and B. Sniffen, "Principles of Remote Attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[124] Docker, "Docker Home Page." `https://docker.com`, 2021.

[125] Docker Docs, "Using the OverlayFS Storage Driver." `https://docs.docker.com/storage/storagedriver/overlayfs-driver/`, 2021.

[126] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking Information Flow in JavaScript and Its APIs," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1663–1671, ACM, 2014.

[127] B. Livshits, "Dynamic Taint Tracking in Managed Runtimes," Tech. Rep. MSR-TR-2012-114, 2012.

[128] D. Chandra and M. Franz, "Fine-grained Information Flow Analysis and Enforcement in a Java Virtual Machine," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, pp. 463–475, IEEE, 2007.

[129] PyPy, "PyPy Home Page." `https://pypy.org/`, 2021.

[130] J. Layton, "Extended File Attribute Rock!." `http://www.linux-mag.com/id/8741/`, 2011.

[131] Oracle Corporation, "Java Native Interface." `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/`, 2021.

[132] C. Qian, X. Luo, Y. Shao, and A. Chan, "On Tracking Information Flows Through JNI in Android Applications," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 180–191, IEEE, 2014.

[133] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART," in *Proceedings of the 26th USENIX Conference on Security Symposium*, pp. 289–306, USENIX, 2017.

[134] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical Taint-Based Protection Using Demand Emulation," in *Proceedings of the 1st ACM European Conference on Computer Systems*, pp. 29–41, ACM, 2006.

[135] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: a Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 135–148, IEEE, 2006.

[136] A. Razeen, A. Lebeck, D. Liu, A. Meijer, V. Pistol, and L. Cox, "SandTrap: Tracking Information Flows on Demand with Parallel Permissions," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, p. 230–242, ACM, 2018.

[137] S. Gurajala, J. White, B. Hudson, and J. Matthews, "Fake Twitter Accounts: Profile Characteristics Obtained Using An Activity-Based Pattern Detection Approach," in *Proceedings of the 2015 International Conference on Social Media & Society*, p. 1–7, ACM, 2015.

[138] E. van der Walt and J. Eloff, "Using Machine Learning to Detect Fake Identities: Bots vs Humans," *IEEE Access*, vol. 6, pp. 6540–6549, 2018.

[139] C. Xiao, D. Freeman, and T. Hwa, "Detecting Clusters of Fake Accounts in Online Social Networks," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pp. 91–101, 2015.

[140] GoDaddy, "What is a Wildcard SSL Certificate?." `https://www.godaddy.com/help/what-is-a-wildcard-ssl-certificate-567`, 2021.

[141] K. Reitz, "Requests: HTTP for Humans." `http://docs.python-requests.org/en/master/`, 2021.

[142] Docker, "Docker SDK for Python." `https://docker-py.readthedocs.io/en/stable/`, 2021.

[143] Trusted Computing Group, "TPM 2.0 Library Specification." `https://trustedcomputinggroup.org/tpm-library-specification/`, 2021.

[144] CoreOS, "GRand Unified Bootloader 2.0." `https://github.com/coreos/grub`.

[145] A. Ronacher, "Flask." `http://flask.pocoo.org/`, 2010.

[146] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-peer Systems," in *Proceedings of the 2001 IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 329–350, ACM, 2001.

[147] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, vol. 31, pp. 149–160, ACM, 2001.

[148] Apache Software Foundation, "Apache Thrift." `https://thrift.apache.org/`, 2021.

[149] D. Schoepe, D. Hedin, and A. Sabelfeld, "SeLINQ: Tracking Information Across Application-Database Boundaries," in *Proceedings of the 19th ACM International Conference on Functional Programming*, pp. 25–38, ACM, 2014.

[150] D. Schultz and B. Liskov, "IFDB: Decentralized Information Flow Control for Databases," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 43–56, ACM, 2013.

[151] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, Dynamic Information Flow for Database-Backed Applications," in *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation*, pp. 631–647, ACM, 2016.

[152] PyPy, "PyPy Benchmarks." `https://bitbucket.org/pypy/benchmarks`, 2018.

[153] Docker, "Docker PyPy Images." `https://hub.docker.com/_/pypy/`, 2021.

[154] Busbey, Sean, "Yahoo! Cloud Serving Benchmark." `https://github.com/brianfrankcooper/YCSB/wiki`, 2019.

[155] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *Proceedings of the 12th ACM International Conference on Measurement and Modeling of Computer Systems*, pp. 53–64, ACM, 2012.

[156] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *International Journal of Computer and Telecommunications Networking*, vol. 53, no. 11, pp. 1830–1845, 2009.

[157] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pp. 153–167, ACM, 2017.

[158] The Linux Foundation, "Linux Network Bridge." `https://wiki.linuxfoundation.org/networking/bridge`, 2021.

[159] Michael Kerrisk, “Namespaces in Operation, Part 1: Namespaces Overview.” https://lwn.net/Articles/531114/, 2013.

[160] N. Brown, “Control groups.” https://lwn.net/Articles/604609/, 2014.

[161] Broadband Search, “Mobile vs. Desktop Usage (Latest 2020 Data).” https://www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics, 2020.

[162] F. Rizzato and I. Fogg, “How AT&T, Sprint, T-Mobile and Verizon Differ in their Early 5G Approach.” https://www.opensignal.com/2020/02/20/how-att-sprint-t-mobile-and-verizon-differ-in-their-early-5g-approach, 2020.

[163] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, “Polaris: Faster Page Loads Using Fine-grained Dependency Tracking,” in *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, p. 123–136, USENIX, 2016.

[164] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha, “Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, p. 390–403, 2017.

[165] A. Sivakumar, C. Jiang, Y. S. Nam, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. G. Rao, S. Sen, M. Thottethodi, and T. N. Vijaykumar, “NutShell: Scalable Whittled Proxy Execution for Low-Latency Web Over Cellular Networks,” in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, p. 448–461, ACM, 2017.

[166] Google, “Google Transparency Report: HTTPS Encryption on the Web.” https://transparencyreport.google.com/https/overview?hl=en, 2020.

[167] F. Tip, “A Survey of Program Slicing Techniques,” *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[168] R. Netravali and J. Mickens, “Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs,” in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, p. 249–266, USENIX, 2018.

[169] X. S. Wang, A. Krishnamurthy, and D. Wetherall, “Speeding Up Web Page Loads with Shandian,” in *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, p. 109–122, USENIX, 2016.

[170] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, “Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices,” in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, p. 439–453, USENIX, 2015.

[171] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, “BlindBox: Deep Packet Inspection over Encrypted Traffic,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, p. 213–226, ACM, 2015.

[172] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "How Much Can We Micro-Cache Web Pages?," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, p. 249–256, ACM, 2014.

[173] S. Mardani, M. Singh, and R. Netravali, "Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating ," in *Proceedingds of the 17th USENIX Symposium on Networked Systems Design and Implementation*, pp. 879–894, USENIX, 2020.

[174] H. Wang, M. Liu, Y. Guo, and X. Chen, "Similarity-based Web Browser Optimization," in *Proceedings of the 23rd International Conference on World Wide Web*, pp. 575–584, ACM, 2014.

[175] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart Caching for Web Browsers," in *Proceedings of the 19th International Conference on World Wide Web*, p. 491–500, ACM, 2010.

[176] M. Hoque, "Poster: Extremely Parallel Resource Pre-Fetching for Energy Optimized Mobile Web Browsing," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, p. 236–238, ACM, 2015.

[177] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who Killed My Battery? Analyzing Mobile Browser Energy Consumption," in *Proceedings of the 21st International Conference on World Wide Web*, p. 41–50, ACM, 2012.

[178] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao, "Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, p. 14–26, ACM, 2015.

[179] B. Zhao, W. Hu, Q. Zheng, and G. Cao, "Energy-aware web browsing on smartphones," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 761–774, 2015.

[180] M. Belshe, BitGo, R. Peon, Google, M. Thomson, and Mozilla, "Hypertext Transfer Protocol Version 2 (HTTP/2)." `https://tools.ietf.org/html/rfc7540`, 2015.

[181] World Wide Web Consortium (W3C), "Resource Hints." `https://www.w3.org/TR/resource-hints`, 2019.

[182] OpenJS Foundation, "Node.js Homepage." `https://nodejs.org/en/`, 2020.

[183] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM International Symposium on Foundations of Software Engineering*, p. 263–272, ACM, 2005.

[184] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation*, p. 213–223, ACM, 2005.

[185] L. de Moura and N. Bjorner, “Z3: An Efficient SMT Solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, p. 337–340, Springer, 2008.

[186] V. Ganesh and D. L. Dill, “A Decision Procedure for Bit-Vectors and Arrays,” in *Proceedings of the 19th International Conference on Computer Aided Verification*, p. 519–531, Springer, 2007.

[187] B. Loring, D. Mitchell, and J. Kinder, “ExpoSE: Practical Symbolic Execution of Standalone JavaScript,” in *Proceedings of the 24th ACM International SPIN Symposium on Model Checking of Software*, p. 196–199, ACM, 2017.

[188] J. Mickens, J. Elson, and J. Howell, “Mugshot: Deterministic Capture and Replay for JavaScript Applications,” in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, p. 11, USENIX, 2010.

[189] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay,” in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, p. 211–224, USENIX, 2002.

[190] M. Eernisse, “EJS: Embedded JavaScript Templating.” `https://ejs.co/`, 2020.

[191] Electron Community, “Electron Documentation.” `https://www.electronjs.org/docs/development/v8-development`, 2020.

[192] T. Tsujikawa, “Nghttp2 Proxy.” `https://nghttp2.org`, 2020.

[193] A. Lexborisov, “C/C++ HTML 5 Parser Using Threads.” `https://github.com/lexborisov/myhtml`, 2020.

[194] Samsung, “Web Development on a Phone. Updated for Linux on DeX..” `https://webview.linuxondex.com/`, 2020.

[195] Sitespeed.io, “Browsertime: Your Browser, Your Page, Your Scripts.” `https://github.com/sitespeedio/browsertime`, 2020.

[196] World Wide Web Consortium (W3C), “WebDriver.” `https://www.w3.org/TR/webdriver/`, 2020.

[197] Software Freedom Conservancy, “SeleniumHQ: Browser Automation.” `https://www.selenium.dev/`, 2020.

[198] J. Ribeiro, “Chrome Headless,” 2020. Docker Hub. `https://hub.docker.com/r/justinribeiro/chrome-headless`.

[199] QFish, “A CSS Parsing Library in Pure C99.” `https://github.com/hackers-painters/katana-parser`, 2020.

[200] Majestic, “The Majestic Million: the Million Domains We Find with the Most Referring Subnets.” `https://majestic.com/reports/majestic-million`, 2020.

[201] WebPageTest.org, "Documentation: Speed Index." https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2020.

[202] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez, "Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation," in *Proceedings of the 18th International Static Analysis Symposium*, p. 112–128, Springer, 2011.

[203] R. Villalobos, "Building a Website with Node.js and Express.js." https://github.com/planetoftheweb/expressjs, 2018.