EFFICIENT DATA PROTECTION BY NOISING, MASKING, AND METERING

Qiuyu Xiao

A Dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2019

ABSTRACT

Qiuyu Xiao: Efficient Data Protection by Noising, Masking, and Metering
(Under the direction of Michael K. Reiter)

Protecting data secrecy is an important design goal of computing systems. Conventional techniques like access control mechanisms and cryptography are widely deployed, and yet security breaches and data leakages still occur. There are several challenges. First, sensitivity of the system data is not always easy to decide. Second, trustworthiness is not a constant property of the system components and users. Third, a system's functional requirements can be at odds with its data protection requirements. In this dissertation, we show that efficient data protection can be achieved by noising, masking, or metering sensitive data. Specifically, three practical problems are addressed in the dissertation—storage side-channel attacks in Linux, server anonymity violations in web sessions, and data theft by malicious insiders. To mitigate storage side-channel attacks, we introduce a differentially private system, `dpprocfs`, which injects noise into side-channel vectors and also reestablishes invariants on the noised outputs. Our evaluations show that `dpprocfs` mitigates known storage side channels while preserving the utility of the proc filesystem for monitoring and diagnosis. To enforce server anonymity, we introduce a cloud service, `PoPSiCl`, which masks server identifiers, including DNS names and IP addresses, with personalized pseudonyms. `PoPSiCl` can defend against both passive and active network attackers with minimal impact to web-browsing performance. To prevent data theft from insiders, we introduce a system, `Snowman`, which restricts the user to access data only remotely and accurately meters the sensitive data output to the user by conducting taint analysis in a replica of the application execution without slowing the interactive user session.

TABLE OF CONTENTS

[1]This chapter is excerpted from previously published work [161]

[2]This chapter is excerpted from previously published work [162]

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CA	Certificate Authority
CDN	Content Distribution Network
GUI	Graphical User Interface
IaaS	Infrastructure-as-a-Service
IRQ	Interrupt Request
ISP	Internet Service Provider
NAT	Network Address Translator
OS	Operating System
PaaS	Platform-as-a-Service
PKI	Public Key Infrastructure
RCB	Retired Conditional Branch
SACK	Selective Acknowledgment
SDN	Software-defined Networking
SIMD	Single Instruction Multiple Data
SNI	Server Name Indication
SVM	Support Vector Machine
TAD	Traffic Analysis Defense
URL	Uniform Resource Locator
VM	Virtual Machine
VPN	Virtual Private Network

CHAPTER 1: INTRODUCTION

Protecting data secrecy is an important design goal of computing systems. A common approach is data isolation. In the classic multi-level security model [16], system data and system components are categorized into various security levels. This model ensures that system components can only read data in the same or lower security levels, so that information can only flow from a lower level to a higher level but not vice versa. Modern computing systems adopt similar concepts to protect data secrecy. In Linux, the memory data of the system kernel cannot be accessed by user-space processes. Similarly in Xen, the memory data of the hypervisor cannot be accessed by virtual machine kernels and user-space processes.

Simple data isolation rules often cannot meet the data protection requirements of complicated systems, especially in the multi-user environment. Many security mechanisms were proposed to control the propagation of secret data. Access control mechanisms [135, 136, 151, 81, 138] ensure that secret data can only be accessed by the users who have the required permissions. Information-flow control mechanisms [134, 95, 52, 76, 24, 110] prevent secret data propagating to public data during the computation of the program. Many other security mechanisms restrict the propagation of the secret data by hiding the contents of secret data. Cryptography protocols [17, 131, 130, 168] encrypt secret data to ciphertext that can be only decrypted with the private key. Anonymization schemes [148, 55, 69, 28, 78] scrub personal identifiers before releasing the data to prevent re-identification.

Despite these efforts, data leakage incidents still frequently occur in computing systems [12, 49, 61]. There are several challenges. First, sensitivity of the system data is not always easy to decide. We need to know what data to protect before deploying security mechanisms to protect it. It is very obvious that some data is sensitive, e.g., private keys. However, some seemingly non-sensitive data is statistically correlated with sensitive data. This is a fundamental problem in side-channel attacks where an attacker can conduct statistical analysis on non-sensitive data (e.g., cache timing information) to recover sensitive data (e.g., private key). Second, trustworthiness is not a constant property of system components and users. The system kernel is trusted but an attacker can control the kernel by exploiting vulnerabilities. Employees are trusted to access confidential files but they might turn rogue and steal the confidential data. Third, system's functional requirements can be at odds with its data protection requirements. Sometimes sensitive data is chosen to be exposed to untrusted system components in favor of meeting the system's functional requirements. For example, IP addresses in the network packets of a user are exposed to the Internet service provider (ISP) because routers rely on the IP information to make routing decisions. However, the ISP can also build a web browsing profile of the user based on the IP information, which violates the user's privacy.

In this dissertation, we propose novel data protection mechanisms to address unique challenges in three practical problems—storage side-channel attacks in Linux, server anonymity violations in web sessions, and data theft by malicious insiders. Based on sensitivity of the system data, trustworthiness of the system components, and functional requirements of the system, we choose to either noise, mask, or meter sensitive data with the goal of protecting data secrecy while maintaining normal system functionality.

1.1 Noising

A *noising* mechanism obfuscates sensitive data by adding random noise or modifying the data value before releasing the sensitive data to untrusted parties. The goal of the noising mechanism is maintaining the utility of sensitive data while reducing the risk of security and privacy violation.

In previous work, noising mechanisms have been applied to solve various problems. Privacy-preserving data mining techniques (e.g., [7, 6, 152, 156]) aim to develop accurate data mining models without access to precise information in individual data records. They use randomizing algorithms to obfuscate individual data values and simultaneously preserve underlying distribution properties at a macroscopic level. Privacy-control methods for statistical databases (e.g., [5, 56, 23, 62]) allow untrusted parties to query aggregate statistics of subsets of entities in the database while protecting privacy of any individual entity represented in the database by adding noise to the query results. Privacy-enhancing recommendation systems (e.g., [140, 141, 99]) obfuscate user data before releasing the data to recommendation servers with the goal of preserving recommendation quality. k-anonymity and the derivative theorems [148, 92, 98] define rules to obfuscate sensitive attributes of individuals before publishing data records so that an individual cannot be easily identified from the published records. Adding noise is also used in defending against side-channel attacks by reducing the precision of side-channel vectors [170, 154].

In our work, we propose a novel noising mechanism to defend against storage side-channel attacks. A storage side channel occurs when an attacker accesses data objects influenced by another victim computation and infers information about the victim that it is not permitted to learn directly. In Linux, side-channel attackers exploit the runtime metadata, such as memory and CPU usage of processes, from the proc filesystem (`procfs`) to recover secrets. In Chapter 2, we introduce a differentially private system, `dpprocfs`, which adds noise to side-channel vectors to provably mitigate storage side-channel attacks and simultaneously preserve the utility of `procfs` for monitoring and diagnosis.

1.2 Masking

A *masking* mechanism transforms sensitive data to a form that hides its original contents from the untrusted parties in the masking phase. The original data can be recovered from the transformed data by the trusted parties in the unmasking phase. Masking mechanisms are very useful in scenarios where sensitive data has to be stored in or transmitted by untrusted parties.

Masking mechanisms are mostly enabled by cryptographic algorithms, where the trusted parties use cryptographic keys to encrypt and decrypt sensitive data. Secure communication across an untrusted network is supported by cryptographic protocols (e.g., [130, 168]) which first authenticate communication endpoints and then mask and unmask sensitive data by encryption and decryption. Secure cloud storage systems (e.g., [18, 166]) only store encrypted data in the cloud storage and explore novel solutions to implement useful services on top of the encrypted data, such as deduplication and auditing. Secure processor extensions (e.g., [101, 150]) encrypt memory data of the trusted programs which cannot be undetectably altered by even the most privileged system components, like operating system kernel or hypervisor.

Anonymous communication tools (e.g., [55, 69, 28, 78]) focus on protecting the identity of the communication endpoints by masking and unmasking common identifiers, including DNS names and IP addresses, besides protecting the communication contents by using encryption. Server anonymity, which ensures that the server's identity is hidden from network attackers and eavesdroppers, is one of the anonymous properties guaranteed by these tools. In Chapter 3, we introduce a cloud service, `PoPSiCl`, which enforces server anonymity for tenant servers running in the cloud by masking server identifiers with personalized pseudonyms. Unlike most existing anonymous tools, `PoPSiCl` requires no changes to client-side software and works with all major web browsers. Our evaluations show that `PoPSiCl` only introduces modest overhead to server access latency and throughput, and is capable of scaling to large numbers of users.

1.3 Metering

A *metering* mechanism monitors the amount of sensitive data leaked to the untrusted parties and takes extra steps to protect data secrecy if the data access pattern deviates from normal behaviors (e.g., the amount of leakage exceeds a certain limit).

Quantitative information flow analysis (e.g., [42, 43, 44, 96, 172]) quantifies the amount of sensitive data leakage of a program by leveraging program analysis and formal modeling. With the quantification of sensitive data leakage, flexible data protection policies can be enforced (e.g., allowing "small leakage" but preventing "large leakage"). In differential privacy [62], there is a privacy budget which defines the maximum private information a user is allowed to get when using the system. Every time a user makes a query, the system counts the amount of private data leakage and deducts the amount of leakage from the privacy budget. When the privacy budget is used up, the user is not allowed to make future queries. This privacy budget based protection is basically a metering mechanism, and is deployed in practical systems to protect privacy for making SQL-like queries [103], doing MapReduce computations [132], using location-based services [9, 25], and getting personalized recommendations [140, 102].

In our work, a novel metering mechanism is proposed to defend against data theft by insiders, which involves misuse of permissions that the insider presumably must be given to perform his/her duties in the organization. In Chapter 4, we introduce a system, `Snowman`, which restricts the user to access data only remotely and accurately meters the sensitive data output to the user through the graphical user interfaces. Under the protection of `Snowman`, malicious insiders exfiltrating large volumes of sensitive data in a short time span can be distinguished from normal users. Also, `Snowman` conducts metering without slowing the interactive user session, by concurrently tracking leakage in a replica of the application execution.

CHAPTER 2: DPPROCFS: NOISING SIDE-CHANNEL VECTORS TO MITIGATE STORAGE SIDE CHANNELS[1]

Side-channel attacks aim at disclosing data in computer systems by exfiltrating sensitive information through interfaces that are not designed for this purpose. In recent years, the scope of side-channel attacks has been extended beyond their traditional use to attack cryptographic keys, and techniques utilized in side-channel analysis have also increased in variety and sophistication.

In this chapter, we examine one particular type of side-channel attack vector, which we call *storage side channels*. Storage side channels occur when an adversary accesses data objects associated with a victim computation and makes inferences about the victim based on the contents of the data objects themselves or their metadata. As we use the term here, storage side channels form a subclass of storage *covert* channels [114] that gleans information from an unwitting victim, versus receiving information inconspicuously from an accomplice. Storage side (and covert) channels differ from legitimate communication channels since the data value or the metadata exploited by the side channel is not considered sensitive by itself; yet, it still leaks information that may be exploited to infer victim secrets.

A generic approach to mitigate storage side (and covert) channels is to reduce the accuracy of the data or its metadata being reported by adding random noise to disturb side-channel observations [114]. A challenge in this approach is to develop principled mechanisms to perturb the side channels with provable security guarantees, and to do so while preserving the utility of the data and metadata in the system.

In this chapter, we present a novel approach to doing so by leveraging privacy concepts in storage side-channel defense. By limiting data reporting to conform to *differential privacy* and generalizations thereof, we show how to introduce noise into the data reporting so as to bound

[1]This chapter is excerpted from previously published work [161]

information leakage mathematically. The difficulties in doing so, however, stem from the challenges in (i) modeling these storage channels as statistical databases, where differential privacy was previously applied; (ii) designing privacy mechanisms to add noise so that side channels are provably mitigated; and (iii) designing these mechanisms so as to minimize the loss of utility of the released data. We will discuss methods to address these challenges in the remaining sections. In theory, these methods can be applied to mitigating a variety of storage side channels. However, in this chaper we illustrate the idea by focusing only on storage channels based on `procfs`, a file-system interface for reporting resource usage information on Linux and Android systems.

Toward this end, we propose a modified `procfs`, dubbed `dpprocfs`, that provides guarantees about the inferences possible from values reported through the `procfs` interfaces. In doing so, `dpprocfs` defends against a variety of storage side channels recently exploited in `procfs` on both Linux and Android (see Section 2.1.1 for a summary of these attacks). Our work builds on the works of Dwork et al. [63, 64] and Chan et al. [33], which consider differential privacy under continuous observations, but we are forced to extend from this starting point in multiple ways. First, differential privacy itself is not a good match for side-channel mitigation in the `procfs` context; rather, we turn to a recent generalization called d-privacy [35] that is parameterized by a distance metric d. By defining a suitable distance metric d and expressing side-channel mitigation goals in terms of the distance between two series of `procfs` observations, we prove that the differentially private mechanism of Chan et al. [33] generalizes to mitigate storage side channels. Second, however, the naive application of this mechanism to noise `procfs` outputs would risk correctness of applications that depend on invariants that `procfs` outputs satisfy in practice. To retain the utility of `procfs`, `dpprocfs` therefore extracts and reestablishes invariants on the noised outputs so as to assist applications that depend on them.

We implemented `dpprocfs` for Linux as a suite that consists of an extension of the Linux kernel, a userspace daemon process, and a software tool that is used for generating invariants on the values of kernel data structures offline. The kernel extension alters the functionality of `procfs` to enforce d-privacy on the exported data values while preserving the standard `procfs` interfaces. The userspace daemon interacts with the kernel extension to reestablish the invariants `procfs` satisfies. We will elaborate on our implementation choices in later sections.

We evaluate our prototype for both its security and utility. For security, we demonstrate configurations that effectively mitigate existing `procfs` side-channel attacks from the literature. We specifically demonstrate preventing two attacks, one that uses `procfs` data to measure keystroke behavior as a means to recover a typed input, and another that monitors the resource usage of a browser process to determine the website it is accessing [82]. We evaluate the utility of `dpprocfs` by measuring the relative error of protected fields and the similarity of the resource-use rankings of processes by the popular `top` utility to those rankings without noise.

In summary, our contributions are as follows:

- We bring advances in privacy for statistical databases to bear on storage side-channel defense. Specifically, we show that an existing mechanism due to Chan et al. [33] for enforcing differential privacy under continuous *binary* data release extends to implement d-privacy for a distance metric d^* that can quantify storage side channels in `procfs`. We define this distance metric d^*, argue its utility for capturing storage side channels, and prove that the Chan et al. mechanism implements d^*-privacy.
- We identify a challenge in inserting noise into `procfs` outputs, namely the violation of invariants that `procfs` clients (and `procfs` code itself) might depend. Drawing from previous research in invariant identification, we develop a tool for extracting invariants and imposing them upon noised values prior to returning `procfs` outputs. In doing so, we ensure that `procfs` outputs are consistent, even while being noised to interfere with side channels.
- We develop a working implementation of `dpprocfs`, our variant of `procfs` that implements storage side-channel defense, and evaluate both the protection it offers against previously published attacks and the utility it offers for monitoring and diagnosis. Our results illustrate that side-channel defense can be accomplished while still maintaining the utility of `procfs` for its intended purposes.

The remainder of this chapter is organized as follows. Section 2.1 provides an overview of storage side channel attacks via `procfs`, the background of differential privacy, and the theoretical basis of d-privacy. Section 2.2 presents our design of `dpprocfs`, which is followed by details of its implementation in Section 2.3. We evaluate both the security and utility of `dpprocfs` in Section 2.4 and discuss remaining challenges in Section 2.5. We summarize this chapter in Section 2.6.

2.1 Background

2.1.1 Side Channel Attacks via PROCFS

`procfs` is a pseudo file system implemented in Linux, Android, and a few other UNIX-like operating systems to facilitate userspace applications' accesses to kernel-space information. Two types of information are typically shared through `procfs`: per-process information and system-wide information. Per-process information reveals configuration and state information about a process, including path of the executable, environment variables, size of virtual and physical memory, CPU and network usage, and so on. While some of the information should only be consumed by the process itself, other information, especially statistics about resource usage, is required for performance monitoring and diagnosis. For instance, in Linux, `top`, `ps`, `iostat`, `netstat`, `pidstat`, and others rely on `procfs` to function. In Android, `procfs` is used for apps to monitor the resource usage, e.g., transferred network data, of other apps.

This useful facility has been exploited to conduct side-channel attacks by several prior works. Particularly of interest in our work are the attacks exploiting publicly available per-process information to infer secrets of the targeted process; see Table 2.1 for examples. The techniques underlying these attacks are similar. Jana et al. [82] introduced an attack that, by reading from a file in `procfs`, `/proc/<pid>/statm`, and learning the data resident size (drs) of a Chrome browser, enables a malicious co-located application to infer the website it is visiting. The feature used to differentiate multiple websites being browsed is the snapshot of the application's memory

Reference	Description	procfs files used	Kernel fields
Jana et al. [82]	Memory footprint and context switches of a browser process leak website it visits	/proc/<pid>/statm /proc/<pid>/status /proc/<pid>/schedstat	mm_struct.total_vm mm_struct.shared_vm task_struct.nvcsw task_struct.nivcsw
Zhou et al. [171]	Sizes of network packets to/from Android app leaks its activity	/proc/uid_stat/<uid>/tcp_rcv /proc/uid_stat/<uid>/tcp_snd	uid_stat.tcp_rcv uid_stat.tcp_snd
Chen et al. [40]	Android foreground activity identified using shared memory, CPU utilization time and network activity	/proc/<pid>/statm /proc/<pid>/stat /proc/uid_stat/<uid>/tcp_rcv /proc/uid_stat/<uid>/tcp_snd	mm_struct.shared_vm mm_struct.rss_stat .count[MM_FILEPAGES] mm_struct.rss_stat .count[MM_ANONPAGES] uid_stat.tcp_rcv uid_stat.tcp_snd task_struct.utime
Lin et al. [94]	Use of software keyboard detected using CPU utilization time	/proc/<pid>/stat	task_struct.utime

Table 2.1: Selected attacks leveraging storage side channels in the procfs file system

footprint. Zhou et al. [171] explored ways in Android to infer a victim app's activity by monitoring its network communications. Specifically, by sampling the files /proc/uid_stat/<uid>/tcp_rcv and /proc/uid_stat/<uid>/tcp_snd, an adversary is able to learn the packet sizes sent and received by the victim app with high accuracy. Chen et al. [40] extracted the victim app's CPU utilization time, memory usage, and network usage from various procfs files to classify the application's behaviors. Lin et al. [94] also used utime to recognize a user's operation of the software keyboard on Android.

2.1.2 Differential Privacy

Privacy concerns arise when a database client learns information about individuals represented in the database through one or multiple queries to the database [5]. Differential privacy puts constraints on publishing aggregate information from a statistical database which limits the disclosure of private information of an individual in the database [62]. Prior to our work, differential privacy has been implemented in practical systems, e.g., to support privacy for data accessed through SQL-like queries [103] or MapReduce computations [132]. The security scenarios we consider, however, differ from the statistical database privacy model in two dimensions: First, storage side channels revolve around information leakage due to an attacker continuously monitoring the same data as it changes over time. Statistical databases are typically static, however. Second, database indistinguishability is not well defined under our security model, and hence we need to adapt the definition of differential privacy for our intended purposes.

We build our work upon two lines of research in the literature. The first line is concerned with differential privacy with continuous data release [63, 64, 33]. In these works, the continuous data release takes the form of a sequence of binary values, and only sequences that differ in a single binary value are rendered indistinguishable to the attacker. In the model we consider, in contrast, the continuous data release can be characterized as a sequence of integers, and even sequences that differ in multiple values might need to be rendered indistinguishable. The second line of research generalizes the definition of differential privacy for statistical databases. In particular, Chatzikokolakis et al. [35] broadened the definition of differential privacy by parameterizing the definition with a distance metric d, and requiring that the degree of indistinguishability of two databases be a function of their distance. (The original definition of differential privacy can be viewed as a special case for Hamming distance [35].) We build from this approach, defining a metric d that applies to storage side channels and implementing this defense in a working system.

2.1.3 d-Privacy

In our work we leverage a generalization of differential privacy due to Chatzikokolakis et al. [35] called d-privacy, which we summarize here briefly. (Our summary is not of the most general form of d-privacy, however.) A *metric* d on a set $\mathcal{X}$ is a function $d : \mathcal{X}^2 \to [0, \infty)$ satisfying $d(x, x) = 0$, $d(x, x') = d(x', x)$, and $d(x, x'') \leq d(x, x') + d(x', x'')$ for all $x, x', x'' \in \mathcal{X}$. A randomized algorithm $A : \mathcal{X} \to \mathcal{Z}$ satisfies (d, ϵ)-privacy if

$$\mathbb{P}\left(A(x) \in Z\right) \leq \exp(\epsilon \times d(x, x')) \times \mathbb{P}\left(A(x') \in Z\right)$$

for all $Z \subseteq \mathcal{Z}$.

We leverage the following composition property of d-privacy:

Proposition 1. *If $A : \mathcal{X} \to \mathcal{Z}$ is (d, ϵ)-private and $A' : \mathcal{X} \to \mathcal{Z}'$ is (d, ϵ')-private, then $A'' : \mathcal{X}^2 \to \mathcal{Z} \times \mathcal{Z}'$ defined by $A''(x, x') = (A(x), A'(x'))$ satisfies*

$$\begin{aligned}\mathbb{P}\left(A''(x, x') \in Z \times Z'\right) \leq \exp(\epsilon \times d(x, x'') + \epsilon' \times d(x', x'''))\\ \times \mathbb{P}\left(A''(x'', x''') \in Z \times Z'\right)\end{aligned}$$

for any $Z \subseteq \mathcal{Z}$, any $Z' \subseteq \mathcal{Z}'$, and any $x, x', x'', x''' \in \mathcal{X}$.

Proof of Prop. 1.

$$\begin{aligned}&\mathbb{P}\left(A''(x, x') \in Z \times Z'\right)\\ &= \mathbb{P}\left(A(x) \in Z\right) \times \mathbb{P}\left(A'(x') \in Z'\right)\\ &\leq \exp(\epsilon \times d(x, x'')) \times \mathbb{P}\left(A(x'') \in Z\right)\\ &\quad \times \exp(\epsilon' \times d(x', x''')) \times \mathbb{P}\left(A'(x''') \in Z'\right)\\ &= \exp(\epsilon \times d(x, x'') + \epsilon' \times d(x', x'''))\\ &\quad \times \mathbb{P}\left(A''(x'', x''') \in Z \times Z'\right)\end{aligned}$$

□

Let $\mathbb{Z}$ and $\mathbb{R}$ denote the integers and reals, respectively. In the case $\mathcal{X} = \mathbb{Z}^n$, a metric that will be of interest for our purposes is L1 distance, defined by

$$d_{\mathrm{L1}}(x, x') = \sum_{i=1}^{n} |x[i] - x'[i]\,|$$

where $x = \langle x[1], \ldots, x[n] \rangle$.

Proposition 2. *Let $A : \mathbb{Z}^n \to \mathbb{R}^n$ be the algorithm that returns $A(x) = \langle x[1] + r_1, \ldots, x[n] + r_n \rangle$, where each $r_i \overset{\$}{\leftarrow} \mathsf{Lap}\left(\frac{1}{\epsilon}\right)$. Then, for any $x, x' \in \mathbb{Z}^n$ and $Z \subseteq \mathbb{R}^n$,*

$$\mathbb{P}\left(A(x) \in Z\right) \leq \exp(\epsilon \times d_{\mathrm{L1}}(x, x')) \times \mathbb{P}\left(A(x') \in Z\right)$$

Proof of Prop. 2. First note that for any i and any $z \in \mathcal{Z}$,

$$\begin{aligned}
\frac{\mathbb{P}\left(x[i] + r_i = z\right)}{\mathbb{P}\left(x'[i] + r_i = z\right)} &= \frac{\exp(-\epsilon \times |x[i] - z|)}{\exp(-\epsilon \times |x'[i] - z|)} \\
&= \exp(-\epsilon \times (|x[i] - z| - |x'[i] - z|)) \\
&= \exp(\epsilon \times (|x'[i] - z| - |x[i] - z|)) \\
&\leq \exp(\epsilon \times (|x[i] - x'[i]\,|))
\end{aligned}$$

The result then follows from Prop. 1. $\square$

2.2 Design of a d-Private Procfs

In an effort to suppress information leakages in `procfs` such as those described in Section 2.1.1, we devise a new `procfs`-like file system, called `dpprocfs`, that leverages differential privacy principles. In this section, we describe how we apply these principles in the design of `dpprocfs`.

2.2.1 Threat Model

This work considers side-channel attacks exploiting statistics values exported by `procfs` from co-located applications running within the same OS. In particular, we consider the default settings of `procfs`, which do not restrict accesses to a process' private directories in `procfs` by other processes from different users. Such settings are very typical in traditional desktop environments or shared server hosting environments running all kinds of Linux distributions, and mobile devices running Android. We assume the OS kernel and the root user of the system are not compromised. Accordingly, security attacks due to software vulnerabilities are beyond the scope of consideration.

2.2.2 Design Overview

When a `procfs` file is open and read, the data read are created on-the-fly by the Linux kernel. To create the file data, the kernel draws information from several data structures. Examples include the `task_struct` structure that describes a process or task in the system, and the `mm_struct` structure that describes the virtual memory of a process.

One option to interfere with adversary inferences about victim processes using values obtained from `procfs` would be to add noise to those values directly, just before outputting them. Unfortunately, there are numerous outputs from `procfs` with complex relationships among them, and so we determined that adding noise to the underlying kernel data-structure field values used to calculate `procfs` outputs would be a more manageable design choice. In particular, there are fewer such fields, and while there remain relationships among them (more on that below), they are reduced in number and complexity.

So, in the design of `dpprocfs`, we treat updates to the relevant per-process kernel data structures as constituting a "database" x that represents the evolution of the process since its inception. That is, consider a conceptual database x to which a record is added each time one or more of a process' kernel data-structure fields changes. The columns of x correspond to the numeric fields of the per-process kernel data structures consulted by `procfs`. So, for example, the `mm_struct.total_vm` field, which indicates the total number of virtual memory pages of a process, is represented by a column in x. As the process executes, a new record is appended to x anytime the value in one of these fields changes. Each time a `procfs` file is read, the values returned are assembled from what is, in effect, the most recently added row of the database x. We stress, however, that this database is conceptual only, and does not actually exist in `dpprocfs`.

We design an algorithm to implement d-privacy per column of x (i.e., per data-structure field), relying on Prop. 1 to bound the information leaked from multiple columns simultaneously.

Since each column of the database x corresponds to a specific field in a kernel data structure, our mechanism is applied each time a field in a protected data structure is read by `procfs` code. For the remainder of this chapter, we adjust our notation so that the database x represents a single column corresponding to that data-structure field. We refer to $x[i]$ as the value of the last element of that column (i.e., the field in the kernel data structure corresponding to the column) when the i-th access occurs (i.e., $i = 1$ is the first access to the data-structure field).

Even to limit leakage from a single column, it is necessary to decide on a distance metric d for which to implement d-privacy. While we might not know exactly how the adversary uses the `procfs` outputs to infer information about a victim process, we can glean guidance from known attacks. For example, Zhou et al. [171] discuss how they used `procfs` output based on the `uid_stat.tcp_snd` field to infer when a victim sent a tweet (a la Twitter) as follows: "a tweet is considered to be sent when the increment sequence is either (420—150, 314, 580–720) or (420—150, 894–1034)." [171, Section 3.2] That is, their attack works by reading from `procfs` four times in a short interval to obtain values $x[1]$, $x[2]$, $x[3]$, $x[4]$ where x denotes the `uid_stat.tcp_snd` field, and deciding that a tweet was sent if either $x[2] - x[1] \in \{150, 420\}$, $x[3] - x[2] = 314$, and $x[4] - x[3] \in \{580, \ldots, 720\}$ or $x[2] - x[1] \in \{150, 420\}$ and $x[3] - x[2] \in \{894, \ldots, 1034\}$. So, to interfere with this attack, it is necessary to render these readings from the "database" x indistinguishable from readings from an alternative "database" x' that reflects a run in which no tweet was sent. This insight led us to choose the following metric d^* for enforcing privacy:

$$d^*(x, x') = \sum_{i \geq 1} |(x[i] - x[i-1]) - (x'[i] - x'[i-1])|$$

Proposition 3. *d^* is a metric.*

Proof of Prop. 3. For any $x, x' \in \mathbb{Z}^n$, the properties $d^*(x,x) = 0$ and $d^*(x,x') = d^*(x',x)$ are evident. The triangle property results as follows, for $x, x', x'' \in \mathbb{Z}^n$:

$$\begin{aligned}
d^*(x,x'') &= \sum_{i=1}^{n} |(x[i] - x[i-1]) - (x''[i] - x''[i-1])| \\
&= \sum_{i=1}^{n} \left| \begin{array}{l} (x[i] - x[i-1]) - (x'[i] - x'[i-1]) \\ +(x'[i] - x'[i-1]) - (x''[i] - x''[i-1]) \end{array} \right| \\
&\leq \sum_{i=1}^{n} |(x[i] - x[i-1]) - (x'[i] - x'[i-1])| \\
&\qquad + \sum_{i=1}^{n} |(x'[i] - x'[i-1]) - (x''[i] - x''[i-1])| \\
&= d^*(x,x') + d^*(x',x'')
\end{aligned}$$

□

The distance d^* captures the distinguishability of consecutive pairs of observations of a data-structure field via `procfs`, and so by defining d^* in this way (and choosing ϵ appropriately), we ensure that a (d^*, ϵ)-private mechanism can hide the differences between x and x' that, e.g., enabled Zhou et al. to identify a tweet being sent in their attack.

Moreover, adopting d^* is plausibly of use in defending against a much broader range of attacks, since d^*-privacy implies d_{L1}-privacy:

Proposition 4. *If A is (d^*, ϵ)-private, then A is $(d_{\mathrm{L1}}, 2\epsilon)$-private.*

Proof of Prop. 4. First note that

$$\begin{aligned}
d^*(x,x') &= \sum_{i=1}^{n} |(x[i] - x[i-1]) - (x'[i] - x'[i-1])| \\
&\leq \sum_{i=1}^{n} |x[i] - x'[i]\,| + \sum_{i=1}^{n} |x[i-1] - x'[i-1]\,| \\
&\leq 2 \times d_{\mathrm{L1}}(x,x')
\end{aligned}$$

Therefore, if $A : \mathcal{X} \rightarrow \mathcal{Z}$ is (d^*, ϵ)-private, then we have that for any $x, x' \in \mathcal{X}$ and any $Z \subseteq \mathcal{Z}$,

$$\begin{aligned}\frac{\mathbb{P}\left(A(x) \in Z\right)}{\mathbb{P}\left(A(x') \in Z\right)} &\leq \exp(\epsilon \times d^*(x, x')) \\ &\leq \exp(\epsilon \times 2d_{\mathrm{L1}}(x, x')) \\ &= \exp(2\epsilon \times d_{\mathrm{L1}}(x, x'))\end{aligned}$$

□

Since *any* p-point metric space can be embedded in L1 distance with $O(\log p)$ distortion [4], making it difficult to distinguish x and x' with low d^* (and hence L1) distance should make it more difficult to distinguish them via other distance metrics, too.

One challenge of using d-privacy to protect information from kernel data structures used in responding to `procfs` reads is that the information obtained through `procfs` might become inconsistent. That is, our mechanism might break data-structure invariants on which the `procfs` code or the clients of `procfs` rely. `dpprocfs` therefore reestablishes these invariants on the d-private values prior to providing them to `procfs` code. So, for example, since enforcing d-privacy adds noise to the `mm_struct.total_vm` and `mm_struct.shared_vm` values, the resulting values might fail to satisfy the invariant `mm_struct.total_vm` $\geq$ `mm_struct.shared_vm`. `dpprocfs` thus adjusts `mm_struct.total_vm` and `mm_struct.shared_vm` to reestablish this invariant before permitting them to be used by the `procfs` code. In Section 2.2.4, we describe how we generate the invariants for these kernel data structures and how we reestablish those invariants on d-private values. Note that these invariants are public information: they can be extracted statically or dynamically via the same methods we obtain them, and post-processing d-private values to reestablish these invariants does not impinge on their d-privacy (cf., [74]).

2.2.3 d^*-Private Mechanism Design

In this section we describe the mechanism we use to implement d^*-privacy for the conceptual single-column database x described above. This mechanism is due to Chan et al. [33], though they considered only the case where $x[i+1]-x[i] \in \{0,1\}$ and, moreover, differential privacy (so that $x[i+1]-x[i] \neq x'[i+1]-x'[i]$ for only one i), rather than d^*-privacy as we do here. As such, our primary contribution is in proving that this mechanism generalizes to implement d^*-privacy and does so for vectors over the natural numbers.

Let $\mathbb{N}$ denote the natural numbers and $D(i) \in \mathbb{N}$ denote the largest power of two that divides i; i.e., $D(i) = 2^j$ if and only if $2^j | i$ and $2^{j+1} \not| i$. Note that $i = D(i)$ if and only if i is a power of two. The mechanism A computes a value $\tilde{x}[i]$ that is used in place of $x[i]$ in the `procfs` code using the recurrence

$$\tilde{x}[i] = \tilde{x}[G(i)] + (x[i] - x[G(i)]) + r_i \tag{2.1}$$

where $x[0] = \tilde{x}[0] = 0$, $\mathsf{Lap}(b)$ denotes the Laplace distribution with scale b and location $\mu = 0$, and

$$G(i) = \begin{cases} 0 & \text{if } i = 1 \\ i/2 & \text{if } i = D(i) \geq 2 \\ i - D(i) & \text{if } i > D(i) \end{cases} \tag{2.2}$$

$$r_i \sim \begin{cases} \mathsf{Lap}\left(\frac{1}{\epsilon}\right) & \text{if } i = D(i) \\ \mathsf{Lap}\left(\frac{\lfloor \log_2 i \rfloor}{\epsilon}\right) & \text{otherwise} \end{cases} \tag{2.3}$$

So, for example, the first eight queries to x result in the following return values, where $r_i \overset{\$}{\leftarrow} \mathsf{Lap}\,(b)$ denotes sampling randomly according to the distribution $\mathsf{Lap}\,(b)$.

$$
\begin{aligned}
&\tilde{x}[1] \leftarrow x[1] + r_1 && \text{where } r_1 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{1}{\epsilon}\right)\\
&\tilde{x}[2] \leftarrow \tilde{x}[1] + (x[2] - x[1]) + r_2 && \text{where } r_2 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{1}{\epsilon}\right)\\
&\tilde{x}[3] \leftarrow \tilde{x}[2] + (x[3] - x[2]) + r_3 && \text{where } r_3 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{1}{\epsilon}\right)\\
&\tilde{x}[4] \leftarrow \tilde{x}[2] + (x[4] - x[2]) + r_4 && \text{where } r_4 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{1}{\epsilon}\right)\\
&\tilde{x}[5] \leftarrow \tilde{x}[4] + (x[5] - x[4]) + r_5 && \text{where } r_5 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{2}{\epsilon}\right)\\
&\tilde{x}[6] \leftarrow \tilde{x}[4] + (x[6] - x[4]) + r_6 && \text{where } r_6 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{2}{\epsilon}\right)\\
&\tilde{x}[7] \leftarrow \tilde{x}[6] + (x[7] - x[6]) + r_7 && \text{where } r_7 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{2}{\epsilon}\right)\\
&\tilde{x}[8] \leftarrow \tilde{x}[4] + (x[8] - x[4]) + r_8 && \text{where } r_8 \overset{\$}{\leftarrow} \mathsf{Lap}\left(\tfrac{1}{\epsilon}\right)
\end{aligned}
$$

Chan et al. characterize the amount of noise introduced by the mechanism described above, which grows only logarithmically in i, specifically:

Proposition 5 ([33]). *With probability at least* $1-\delta$, $|\tilde{x}[i]-x[i]| = O\left((\log \frac{1}{\delta}) \times (\lfloor \log i \rfloor)^{3/2} \times \epsilon^{-1}\right)$.

Our main contribution as it relates to this mechanism design lies in showing the following result:

Proposition 6. *The algorithm in Eqns. 2.1–2.3 is* $(d^*, 2\epsilon)$*-private.*

Proof of Prop. 6. For any i such that $i = D(i) \geq 2$ and any z_i,

$$
\begin{aligned}
&\frac{\mathbb{P}\left((x[i] - x\left[\frac{i}{2}\right]) + r_i = z_i\right)}{\mathbb{P}\left((x'[i] - x'\left[\frac{i}{2}\right]) + r_i = z_i\right)}\\
&\leq \exp\left(\epsilon \times \left|\left(x[i] - x\left[\frac{i}{2}\right]\right) - \left(x'[i] - x'\left[\frac{i}{2}\right]\right)\right|\right)
\end{aligned}
$$

by Prop. 2, and so

$$\prod_{i:i=D(i)\geq 2} \frac{\mathbb{P}\left((x[i]-x\left[\frac{i}{2}\right])+r_i=z_i\right)}{\mathbb{P}\left((x'[i]-x'\left[\frac{i}{2}\right])+r_i=z_i\right)}$$
$$\leq \exp\left(\epsilon \times \sum_{i:i=D(i)\geq 2} \left|\left(x[i]-x\left[\frac{i}{2}\right]\right)-\left(x'[i]-x'\left[\frac{i}{2}\right]\right)\right|\right)$$
$$\leq \exp\left(\epsilon \times \sum_{i\geq 2} |(x[i]-x[i-1])-(x'[i]-x'[i-1])|\right) \tag{2.4}$$

Similarly, for any $i > D(i)$ and any z_i,

$$\frac{\mathbb{P}\left((x[i]-x[i-D(i)])+r_i=z_i\right)}{\mathbb{P}\left((x'[i]-x'[i-D(i)])+r_i=z_i\right)}$$
$$\leq \exp\left(\frac{\epsilon}{k} \times |(x[i]-x[i-D(i)])-(x'[i]-x'[i-D(i)])|\right)$$

where $k = \lfloor \log_2 i \rfloor$. For any fixed j and k,

$$\prod_{\substack{i\,\in\,(2^k,2^{k+1})\\ :\,D(i)\,=\,2^j}} \frac{\mathbb{P}\left((x[i]-x[i-D(i)])+r_i=z_i\right)}{\mathbb{P}\left((x'[i]-x'[i-D(i)])+r_i=z_i\right)}$$
$$\leq \exp\left(\frac{\epsilon}{k} \times \sum_{\substack{i\,\in\,(2^k,2^{k+1})\\ :\,D(i)\,=\,2^j}} \left|\begin{array}{l}(x[i]-x[i-D(i)])\\ -(x'[i]-x'[i-D(i)])\end{array}\right|\right)$$
$$\leq \exp\left(\frac{\epsilon}{k} \times \sum_{i\in(2^k,2^{k+1})} \left|\begin{array}{l}(x[i]-x[i-1])\\ -(x'[i]-x'[i-1])\end{array}\right|\right)$$

And so,

$$
\begin{aligned}
&\prod_{i:i>D(i)} \frac{\mathbb{P}\left((x[i]-x[i-D(i)])+r_i=z_i\right)}{\mathbb{P}\left((x'[i]-x'[i-D(i)])+r_i=z_i\right)} \\
&= \prod_{k>0}\prod_{j\in[0,k)} \prod_{\substack{i\,\in\,(2^k,2^{k+1})\\ :\,D(i)\,=\,2^j}} \frac{\mathbb{P}\left((x[i]-x[i-D(i)])+r_i=z_i\right)}{\mathbb{P}\left((x'[i]-x'[i-D(i)])+r_i=z_i\right)} \\
&\le \prod_{k>0}\prod_{j\in[0,k)} \exp\left(\frac{\epsilon}{k}\times\sum_{i\in(2^k,2^{k+1})}\left|\begin{array}{l}(x[i]-x[i-1])\\ -(x'[i]-x'[i-1])\end{array}\right|\right) \\
&= \prod_{k>0}\exp\left(\epsilon\times\sum_{i\in(2^k,2^{k+1})}\left|\begin{array}{l}(x[i]-x[i-1])\\ -(x'[i]-x'[i-1])\end{array}\right|\right) \\
&\le \exp\left(\epsilon\times\sum_{i\ge 2}\left|(x[i]-x[i-1])-(x'[i]-x'[i-1])\right|\right) \qquad (2.5)
\end{aligned}
$$

Finally, note that

$$
\begin{aligned}
&\frac{\mathbb{P}\left((x[1]-x[0])+r_i=z_i\right)}{\mathbb{P}\left((x'[1]-x'[0])+r_i=z_i\right)} \\
&\quad\le \exp\left(\epsilon\times\left|(x[1]-x[0])-(x'[1]-x'[0])\right|\right) \qquad (2.6)
\end{aligned}
$$

by Prop. 2. So, for any $x, x' \in \mathcal{X}^n$ and any $\langle z_1,\ldots,z_n\rangle$,

$$
\begin{aligned}
&\frac{\mathbb{P}\left(A(x)=\langle z_1,\ldots,z_n\rangle\right)}{\mathbb{P}\left(A(x')=\langle z_1,\ldots,z_n\rangle\right)} \\
&= \prod_{i=1}^{n} \frac{\mathbb{P}\left((x[i]-x[G(i)])+r_i=z_i-z_{G(i)}\right)}{\mathbb{P}\left((x'[i]-x'[G(i)])+r_i=z_i-z_{G(i)}\right)} \\
&\le \exp\left(2\epsilon\times\sum_{i\ge 1}\left|(x[i]-x[i-1])-(x'[i]-x'[i-1])\right|\right)
\end{aligned}
$$

where the last step follows by multiplying Eqn. 2.6, Eqn. 2.4, and Eqn. 2.5. □

2.2.4 Consistency Enforcement

The values provided to `procfs` code, once rendered d^*-private by the mechanism described in Section 2.2.3, are processed as usual by the `procfs` code to produce the values served as the contents of the queried `procfs` files. By adding noise to these values, however, it is possible that we cause them to violate invariants on which the `procfs` code or the reader of the `procfs` files depends. As such, prior to providing the d^*-private values to the `procfs` code, we process these values to re-establish invariants on which this code might depend.

Specifically, the invariants we reestablish are of two types, namely *one-field* or *multiple-field*. A *one-field* invariant holds between the values of *the same* data-structure field when queried at two different times. For example, the fact that the `task_struct.utime` field is monotonically nondecreasing is a one-field invariant. In contrast, a *multiple-field* invariant holds among the values of two or more data-structure fields accessed at *the same* time, e.g., `mm_struct .hiwater_rss` $<$ `mm_struct.shared_vm`. There could also be invariants that hold among the values of two or more data-structure fields accessed at different times, though we do not consider such invariants here.

Techniques for invariant identification range from static (e.g., [173]) to dynamic (e.g., [67]) and combinations thereof (e.g., [51]). While `dpprocfs` is agnostic to the method of invariant generation, the type we explored for our prototype is dynamic. Intuitively, in this approach we execute the system under a variety of workloads, taking snapshots of the relevant kernel data structures after they are updated. We then post-process these snapshots to identify properties that held consistently in all executions. Obviously we cannot detect all such properties (there are infinitely many that could be inferred from finitely many traces), nor is identifying all of them strictly necessary. (We return to this issue in Section 2.5.) In Section 2.3.2, we detail the invariants that `dpprocfs` enforces in our current implementation, though we stress that these invariants can be generated through a combination of techniques—including manually.

Enforcing these invariants involves processing the data-structure field values output by the d^*-private mechanism described in Section 2.2.3 to satisfy these invariants. More specifically, any attempt to read from a procfs file will cause an access to certain data-structure fields. The values in these fields and in any other fields related to them by multi-field invariants (even transitively) are each subjected to the d^*-private mechanism of Section 2.2.3, producing a noised value $\tilde{x}[i]$ to replace the actual value $x[i]$ in this, the i-th, access to this field. These outputs are then altered to satisfy relevant single-field and multiple-field invariants, resulting in a final output $\hat{x}[i]$ for further processing by the kernel routine that produces the contents of the accessed procfs file.

In Section 2.3.3, we explore two ways of manipulating these outputs to satisfy invariants. In the first, to which we refer as computing a *heuristic* solution to the invariants, dpprocfs leverages a hand-implemented algorithm to deterministically modify the outputs to conform. This method is very efficient, but might alter the outputs more than other ways of satisfying the invariants might. In the second approach, to which we refer as computing the *nearest* solution to the invariants, we generate an integer programming problem with the invariants as constraints and an objective of minimizing the total magnitude of the changes to the d^*-private outputs to conform to the invariants. We then feed this integer program to a commercial solver (in our current implementation, CPLEX[2]) to compute an optimal solution. We stress that both the heuristic and nearest solutions are computed using invariants that an adversary can compute himself (i.e., are public), and so this post-processing does not erode the d^*-privacy of these outputs.

2.3 Implementation

We implemented dpprocfs as a suite of software tools in Ubuntu Linux LTS 14.04 with kernel version 3.13.11. dpprocfs consists of three components: a kernel extension, which we call privfs, that enhances the procfs with d^*-private mechanisms (as discussed in Section 2.2.3) without altering its existing program interfaces; a software tool, invgen, that automatically searches for invariants in kernel data structures for maintaining procfs value consistency (as

[2]http://www.ibm.com/software/commerce/optimization/cplex-optimizer/

discussed in Section 2.2.4); and a userspace daemon, `privfsd`, that interacts with the kernel extension and facilitates consistency enforcement in real time.

2.3.1 d^*-Private Mechanism Implementation

When a file in `procfs` is read by a userspace process, a kernel function is invoked to serve the request, and the return values are sent to the process as if it is reading a file. The values reported by `procfs` are computed from fields in certain kernel data structures. To generate d^*-private outputs, a kernel extension `privfs` computes noised versions of those protected fields for use by the kernel function computing the `procfs` output.

Specifically, `privfs` introduces a kernel data structure of type `privfs_struct` per kernel data-structure field x that is protected (rendered d^*-private) by `dpprocfs`. This structure includes two arrays of floating-point values. After access i to the data-structure field x to which the `privfs_struct` structure is associated, position $\log_2 D(i)$ in these arrays are updated to hold $x[i] - x[G(i)]$ and r_i, respectively. Together with $x\left[2^{\lfloor \log_2 i \rfloor}\right]$ and $\tilde{x}\left[2^{\lfloor \log_2 i \rfloor}\right]$, which the structure also stores, these arrays permit the efficient computation of $\tilde{x}[i+1]$. Also to speed up this computation, the `privfs_struct` structure maintains a buffer of 32B to store precomputed random values $r_{i+1}, r_{i+2}, \ldots$ following the specified Laplace distributions. Buffer refilling is implemented as a tasklet, a type of software IRQ in Linux kernels.

The arrays in `privfs_struct` in our present implementation are of fixed length, specifically 32 floating-point values, which limits the number of queries to the protected data-structure field to $2^{32} - 1$. These arrays might instead be made arbitrarily extensible so as to allow an unlimited number of queries. That said, as the query count i grows, the accuracy of the returned $\tilde{x}[i]$ value decays. As such, alternative designs might limit (or rate-limit) the number of queries to any protected data-structure field by each userspace process or its associated user. Another implementation choice might be to maintain separate arrays for each user of the system, so that queries from one user would not decrease the utility of queries from other users.

privfs does not return $\tilde{x}[i]$ directly for use in computing the procfs output. Instead, it sends this value to privfsd for enforcing invariants across all noised values. privfsd will be discussed in Section 2.3.3, after we discuss how data-structure invariants are identified in Section 2.3.2.

2.3.2 Invariant Generation

Kernel data-structure invariants are generated by a component called invgen. invgen generates two types of invariants, namely one-field and multiple-field invariants as discussed in Section 2.2.4. One-field invariants are relationships between a field's current and previous values. Multiple-field invariants are relationships between different variables when accessed at the same time.

As discussed in Section 2.2.4, our system generates invariants from traces of data-structure values captured during execution. Specifically, invgen does so by collecting execution traces of all numerical data-structure fields that are relevant to procfs outputs. To do so, we patch an OS kernel by adding one more file in the procfs to directly export all numeric kernel data-structure fields of interest. invgen then repeatedly reads the extended procfs file, sampling the values of these fields frequently and writing them into trace files. For this work, traces were collected by monitoring the data-structure fields during the execution of a variety of software programs, including Google Chrome and a set of benchmark applications from Phoronix Test Suite[3]. By executing each benchmark application three times, we collected 22.6MB of trace files.

We then used Daikon [67] to extract invariants from these trace files. To use Daikon, we first configured it with invariant templates, or *filters*, that the tool uses to search for invariants. For one-field invariants, Daikon was configured with filters to locate fields that do not change, that are monotonically nonincreasing, or that are monotonically nondecreasing. For multiple-field invariants, we implemented a filter that Daikon uses to search for linear invariants among a set $\mathcal{X}$ of fields, i.e., a property of the form $\sum_{x \in \mathcal{X}} c_x \times x[i] \geq 0$ that holds for all i, for some constant

[3]http://www.phoronix-test-suite.com

$c_x \in \{-1, 0, 1\}$. We ran Daikon with this filter for two sets $\mathcal{X}$, one for memory-related fields and one for scheduler-related fields. After using Daikon to extract likely invariants in this way, we manually inspected the outputs and discarded those that were either implied by others or that we believed to be spurious. For example, if we get invariants $a > b$, $b > c$, and $a > c$, then $a > c$ can be omitted, because it is implied by the previous two invariants. Other similar implication relationships among invariants can be processed in the same way.

The invariants produced in this way are shown in Table 2.2. (We also include invariants that all fields are integral, but we do not show those, for brevity.) The right half of the table shows the invariants expressed using the labels for kernel data-structure fields indicated in the left half of the table. The fields marked "Protected" in the left half of the table are those that `dpprocfs` renders d^*-private in our present implementation. Those fields marked with a "✠" were selected based on their use in existing attacks (see Section 2.1.1), and those marked with a "checkmark" were selected for protection because they are included in invariants with such fields. One field, namely uptime, is not protected in our present implementation despite being included in invariants, simply because the information it carries (the time since the machine was booted) seems unlikely to carry information useful to a side-channel attack. That said, it could also be protected with minimal additional cost.

The upper right corner of the right half of Table 2.2 lists one-field invariants, e.g., that `task_struct.utime`$[i]$ (the i-th access to `task_struct.utime`) is at least as large as `task_struct.utime`$[i-1]$. That is, `task_struct.utime`$[i]$ is nondecreasing. The other invariants hold for all simultaneous accesses to the indicated fields.

Our chosen method of invariant generation is admittedly limited, in that like any method of invariant generation based on an incomplete set of recorded traces, it allows for false positives and false negatives. False positives—i.e., found "invariants" that are not actually invariants—will presumably not cause difficulties for the `procfs` code or applications when `dpprocfs` enforces them, since even if not invariant, the identified behavior is evidently common. False negatives (i.e., missed invariants) might cause such problems, however, and so it would be prudent to

Data-structure field	Protected	Label
mm_struct.total_vm	✠	totalVM
mm_struct.shared_vm	✠	sharedVM
mm_struct.stack_vm	✓	stackVM
mm_struct.exec_vm	✓	execVM
mm_struct.rss_stat .count[MM_FILEPAGES]	✠	filePages
mm_struct.rss_stat .count[MM_ANONPAGES]	✠	anonPages
mm_struct.rss_stat .count[MM_SWAPENTS]	✓	swapEnts
mm_struct.hiwater_rss	✓	hiwaterRSS
mm_struct.hiwater_vm	✓	hiwaterVM
task_struct.utime	✠	utime
task_struct.stime	✓	stime
task_struct.gtime	✓	gtime
task_struct.signal->cstime	✓	cstime
task_struct.signal->cutime	✓	cutime
task_struct.real_start_time	✓	starttime
task_struct.nvcsw	✠	nvcsw
task_struct.nivcsw	✠	nivcsw
get_monotonic_boottime()		uptime

Invariants			
$\mathsf{totalVM} \geq 0$	$\mathsf{swapEnts} \geq 0$	$\mathsf{cstime} \geq 0$	$\mathsf{utime}[i] \geq \mathsf{utime}[i-1]$
$\mathsf{sharedVM} \geq 0$	$\mathsf{hiwaterRSS} \geq 0$	$\mathsf{cutime} \geq 0$	$\mathsf{stime}[i] \geq \mathsf{stime}[i-1]$
$\mathsf{stackVM} \geq 0$	$\mathsf{hiwaterVM} \geq 0$	$\mathsf{nvcsw} \geq 0$	$\mathsf{gtime}[i] \geq \mathsf{gtime}[i-1]$
$\mathsf{execVM} \geq 0$	$\mathsf{utime} \geq 0$	$\mathsf{nivcsw} \geq 0$	$\mathsf{cstime}[i] \geq \mathsf{cstime}[i-1]$
$\mathsf{filePages} \geq 0$	$\mathsf{stime} \geq 0$		$\mathsf{cutime}[i] \geq \mathsf{cutime}[i-1]$
$\mathsf{anonPages} \geq 0$	$\mathsf{gtime} \geq 0$		$\mathsf{nvcsw}[i] \geq \mathsf{nvcsw}[i-1]$
			$\mathsf{nivcsw}[i] \geq \mathsf{nivcsw}[i-1]$
$\mathsf{hiwaterRSS} < \mathsf{sharedVM}$			$\mathsf{starttime}[i] = \mathsf{starttime}[i-1]$
$\mathsf{hiwaterVM} \geq \mathsf{filePages}$			
$\mathsf{execVM} \geq \mathsf{filePages} + \mathsf{swapEnts}$			
$\mathsf{sharedVM} + \mathsf{filePages} \geq \mathsf{anonPages} + \mathsf{swapEnts}$			
$\mathsf{sharedVM} + \mathsf{execVM} \geq \mathsf{filePages} + \mathsf{anonPages} + \mathsf{swapEnts}$			
$\mathsf{sharedVM} \geq \mathsf{execVM} + \mathsf{filePages} + \mathsf{swapEnts}$			
$\mathsf{totalVM} \geq \mathsf{execVM} + \mathsf{stackVM} + \mathsf{filePages} + \mathsf{anonPages} + \mathsf{swapEnts}$			
$\mathsf{totalVM} \geq \mathsf{sharedVM} + \mathsf{stackVM} + \mathsf{swapEnts}$			
$\mathsf{totalVM} + \mathsf{filePages} \geq \mathsf{sharedVM} + \mathsf{anonPages} + \mathsf{swapEnts}$			
$\mathsf{totalVM} + \mathsf{execVM} \geq \mathsf{sharedVM} + \mathsf{stackVM} + \mathsf{filePages} + \mathsf{anonPages} + \mathsf{swapEnts}$			
$\mathsf{uptime} \geq \mathsf{starttime} + \mathsf{utime} + \mathsf{stime} + \mathsf{gtime} + \mathsf{cutime} + \mathsf{cstime}$			

Table 2.2: Selected kernel data-structure fields (Linux kernel 3.13) and generated invariants (Section 2.3.2) that reference them. “Protected” fields are rendered d^*-private as described in Section 2.3.1, either because they have been utilized in published side-channel attacks (✠) or because they are involved in invariants that include such fields (✓).

augment our dynamic approach with static analysis (e.g., [51, 173]) and additional manual inspection. That said, we have not identified applications (or kernel routines that respond to `procfs` reads) that appear to depend on behaviors other than those identified in Table 2.2.

2.3.3 Reestablishing Invariants

Upon producing $\tilde{x}[i]$ for each protected field x needed by a kernel routine to respond to a `procfs` query,[4] `privfs` needs to reestablish the invariants among those field values before submitting them to the kernel routine. For our prototype, we implemented this step in a userspace daemon process, which we call `privfsd`, that receives requests from the `privfs` via Netlink sockets. This implementation choice allows us to sidestep the need to port more complex operations (e.g., floating-point operations, constraint-solving algorithms) to run in the kernel.

[4]We are abusing notation here slightly, in that the access index i might be different per field x.

`privfs` produces inputs for `privfsd` by first identifying the set $\mathcal{X}$ of protected fields to be accessed by the kernel routine serving the `procfs` query (i.e., from those fields marked "protected" in Table 2.2). `privfs` forms the set of relevant invariants from Table 2.2, namely $I(\mathcal{X}) = \bigcup_{x \in \mathcal{X}} I(x)$ where $I(x)$ for any x is defined using the following inductive definition: (i) $I(x)$ is initialized to include any constraint in Table 2.2 that includes field x; and (ii) if any protected field x' from Table 2.2 is named in an invariant already in $I(x)$, then $I(x')$ is added to $I(x)$. `privfs` instantiates each protected field x named in $I(\mathcal{X})$ with a variable $\hat{x}[i]$ and, if $\mathsf{uptime} \in I(\mathcal{X})$, instantiates uptime with its current value. `privfs` then produces the relevant value $\tilde{x}[i]$ for each field $x \in \mathcal{X}$ and sends $I(\mathcal{X})$ and the noised values $\{\tilde{x}[i]\}_{x \in \mathcal{X}}$ to `privfsd`.

`privfsd` operates in one of two modes, computing either a *nearest* compliant assignment to each $\hat{x}[i]$ or a *heuristic* assignment to each $\hat{x}[i]$. The nearest assignment is calculated by taking the instantiated invariants $I(\mathcal{X})$ as constraints in an integer programming (IP) problem, with variables $\{\hat{x}[i]\}_{x \in \mathcal{X}}$ and objective being to minimize the cumulative relative error, i.e., to minimize $\sum_{x \in \mathcal{X}} |\tilde{x}[i] - \hat{x}[i]\,|/|\tilde{x}[i]\,|$. Our current implementation invokes CPLEX to solve this IP problem. In contrast, the heuristic approach simply calculates *any* values for $\{\hat{x}[i]\}_{x \in \mathcal{X}}$ that satisfy $I(\mathcal{X})$ using manually coded heuristics to adjust the $\{\tilde{x}[i]\}_{x \in \mathcal{X}}$ values. Basically, the heuristic approach simply increases or decreases each $\hat{x}[i]$ to satisfy the equality or inequality relation in the invariant. In Section 2.4, we will evaluate both modes of operation. Regardless of its mode of operation, `privfsd` returns the computed values $\{\hat{x}[i]\}_{x \in \mathcal{X}}$ to `privfs` to pass along to the kernel routine for preparing the `procfs` output to the waiting client.

2.4 Evaluation

In this section we evaluate the efficacy of `dpprocfs` design. While our design is provably d^*-private (and hence d_{L1}-private by Prop. 4), we perform an empirical security evaluation of our design in Section 2.4.1 to better illustrate settings of ϵ that suffice to interfere with known attacks. With greater clarity as to reasonable settings of ϵ, we then evaluate the utility of `procfs` for these ϵ values in Section 2.4.2.

2.4.1 Security Evaluation

In this section, we evaluate the capability of `dpprocfs` to defend against side-channel attacks discussed in Section 2.1.1. Specifically, we measure the extent to which the `procfs` features used by the attacker in selected attacks are still effective attack features in `dpprocfs`. Rather than trying to replicate each attack from previous work exactly, we adopt a more general framework for evaluation in which the attacker's task is detecting one of m classes of activities.

We perform this measurement of the attacker's likely success by building a multiclass classifier for classifying `procfs` features (which are attack-dependent) into one of m classes. We use the `scikit-learn`[5] support-vector-machine (SVM) implementation to build the multiclass classifier. We then report the *accuracy* of the classifier in a testing phase, namely the fraction of test instances that it classifies correctly.

2.4.1.1 Defending Against Keystroke Timing Attacks

A user's interactions with the hardware keyboard can trigger context switches from the user-space application to the operating system kernel to handle keystrokes. The voluntary context switch counter (nvcsw in Table 2.2) can be exploited to identify a user's keystroke actions and hence the timing characteristics of those keystrokes. These timing characteristics can then leak information about what those keystrokes were (e.g., [146]). To approximate the defense that `dpprocfs` offers against this attack, we consider an adversary that consecutively reads the nvcsw field from `procfs` six times, and then the adversary classifies this vector of readings to determine when the keystroke occurred. (We only inject one keystroke during these six readings.) As such, we model the attacker as a multi-class classifier, which classifies the vector of six readings (i.e., a vector in $\mathbb{N}^6$) into $m = 5$ classes; classifying a vector as class i indicates that the keystroke occurred between reads i and $i + 1$.

To perform these experiments, we used a tool called `xdotool` to simulate the keystroke actions at a specified time. During each experiment run, we started a `bash` terminal and injected

[5] `http://scikit-learn.org/dev/index.html`

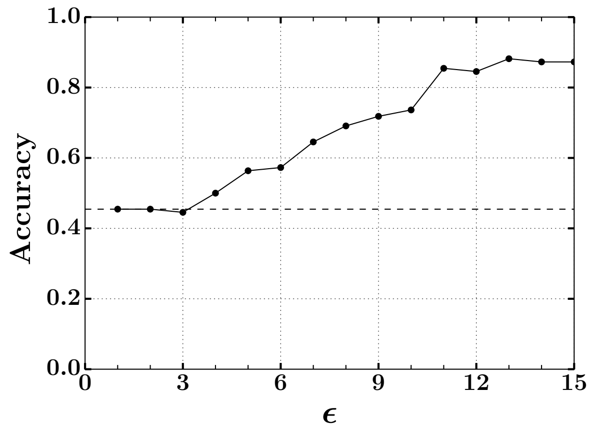
one keystroke, at a time distributed normally with mean 2.5s and standard deviation 0.83s (i.e., 0.0s is three standard deviations from the mean). Beginning with the launch of the `bash` process, the attacker process read the `/proc/<pid>/status` file of the `bash` process every second[6] to obtain the voluntary context switch counter nvcsw, yielding six readings (a vector in $\mathbb{N}^6$). Invariant enforcement (Section 2.3.3) provided the *nearest* solution to the needed invariants. To allow for a powerful attacker, we provided to it the underlying normal distribution imposed on the keystroke timing. The attacker used this distribution to estimate the true (unnoised) nvcsw value corresponding to each vector element (adapting [111, Eqn. 10]), yielding an estimated true vector per collected vector. We repeated this experiment 440 times to get 440 estimated true vectors. When training and testing the SVM classifier, we used 75% of the vectors from each class for training and 25% for testing.

The accuracy of the resulting classifier on the testing examples is shown in Figure 2.1a. The horizontal axis shows various values of ϵ; the vertical axis shows classifier accuracy. Because of the form of the distribution imposed on keystroke timings, the most likely class occurred roughly 44% of the time, and so this baseline (shown by the horizontal dashed line) is the accuracy that the adversary could achieve simply by blindly guessing based on that distribution. As shown in the graph, setting $\epsilon \in [1, 3]$ suffices to reduce the classifier to this baseline accuracy. By comparison, the classifier was perfect (an accuracy of 1.0) when no noise was added.

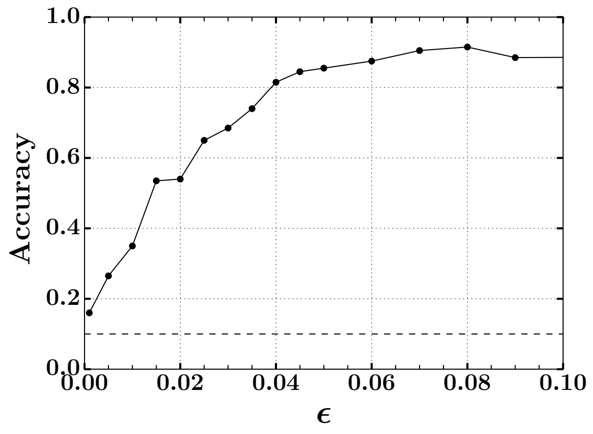
2.4.1.2 Mitigating Website Inference

The memory footprint of a browser can leak the website it visits (as discussed in Section 2.1.1). In this experiment, we instrumented the Google Chrome browser with a script to visit a target website, chosen uniformly from the Alexa top-10 websites. While this occurred, an attacker process repeatedly sampled the data resident size field drs, calculated as $\mathsf{totalVM} - \mathsf{sharedVM}$ (using

[6]This interval is much longer than in the demonstrated attack of Jana et al. [82], but we lengthened this interval to minimize ambiguity regarding the class $i \in \{1 \ldots 5\}$ to which each vector should be assigned for training. By increasing this interval, we believe we produced classification results that are conservative (i.e., advantageous for the attacker).



(a) Keystroke timing attack



(b) Website inference attack

Figure 2.1: Multi-class classifier accuracy under different ϵ settings; dashed horizontal lines show accuracies of blind guesses based only on knowledge of the likelihood of each class

the labels defined in Table 2.2), by reading the `/proc/<pid>/statm` of the browser process every 500μs. To support this rate of sampling, `dpprocfs` employed the *heuristic* method of invariant reestablishment (Section 2.3.3), which returned results in roughly 50μs (in comparison to 8ms for the nearest solution). The sampling period lasted for 3s, during which the attack process recorded all the drs field values read. As in Section 2.4.1.1, the attacker estimated the true (unnoised) drs value corresponding to the j-th read value in each 3s interval, using an empirical distribution observed for these j-th values gathered by accessing each of these 10 websites an equal number of times. The attacker then constructed a histogram of these estimated drs values binned into seven equal-width bins, and the vector of bin counts (in $\mathbb{N}^7$) was used as a feature vector for classification. Each of the Alexa top-10 websites were visited 100 times; when used to train and test the SVM classifier (with $m = 10$ classes), 70% were used for training and 30% were used for testing.

The resulting accuracy of the classifier is shown in Figure 2.1b. The most important distinction from the graph in Figure 2.1a is that the values of ϵ needed to interfere with the website inference attack are much smaller, meaning that the noise added was greater. This is primarily a function of the size differences between drs readings from the m classes, which were generally much greater than the differences between the readings of the voluntary context switch counter nvcsw with and without a keystroke. In terms of d^*, the distances between the classes in the website inference attack were much greater than the distances between classes in the keystroke attack. This is noteworthy because it implies that the settings of ϵ needed for privacy will differ per-field and per-application and, to some extent, will need to be informed by known attacks. Still, however, several values tested for ϵ decayed classification accuracy to a significant extent; with no noise added, the classifier reached 0.915 accuracy.

2.4.2 Utility Evaluation

We evaluate the utility of `dpprocfs` in two ways. First, we measure the relative error of selected `procfs` outputs that are calculated using fields protected by `dpprocfs`, under the two methods discussed in Section 2.3.3 for enforcing invariants, namely producing a heuristic solution and a nearest solution to the invariants. Second, we report the impact of `dpprocfs` to the ranking of processes according to certain features by `top`, a common utility for monitoring and diagnosis. Here we focus on `dpprocfs` outputs such as memory and CPU usage, as these are generally useful systems diagnostics.

2.4.2.1 Relative Error

We begin our utility evaluation by measuring the relative error of the drs field, the same field exploited by website inference attackers (see Section 2.4.1.2). To calculate the relative error of this field under `dpprocfs`, we preserved access to an unprotected version of `procfs` alongside the protected version. Then, we extended our setup described in Section 2.4.1.2 to simultaneously query both the protected and unprotected versions of the drs field while the browser process was running. During the evaluation, the browser was instrumented to repeatedly visit `https://www.youtube.com`, and the drs field was queried every 50ms for a total of 500 queries. We repeated this experiment 200 times.

Figure 2.2 shows the distribution of relative error for both the nearest and heuristic solutions for invariant reestablishment, computed on the same noised values $\tilde{x}$ produced by `privfs`, for a parameter setting ($\epsilon = 0.005$) that provided good security for the side-channel attack tested in Section 2.4.1 (see Figure 2.1b). Each query range on the horizontal axis has two box-and-whiskers plots, one for nearest and one for heuristic. The three horizontal lines forming each box indicate the first, second (median), and third quartiles, and the whiskers extend to cover all points within $1.5\times$ the interquartile range. Outliers are indicated using plus ("+") symbols. A different box-and-whiskers plot is shown per 100-query block across the 200 runs (i.e., each boxplot represents 20,000 points) because the noise increases as the number of queries grows. The differences between the nearest and heuristic distributions are nearly imperceptible, and this trend holds for other parameter and `procfs` fields we have explored, as well. That said, the heuristic solution relies on hand-tuned algorithms and by default provides no guarantees, and so in cases where the speed of computing the nearest solution is acceptable—the nearest solution took an average of 8ms to return, whereas our heuristic approach completed in an average of $50\mu\text{s}$—it might be preferable.

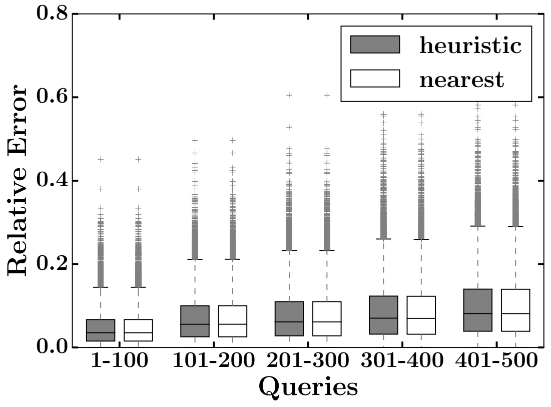


Figure 2.2: Comparison between nearest and heuristic invariant reestablishment for drs field; $\epsilon = 0.005$

Figure 2.3 and Figure 2.4 represent the relative error in readings of the drs field and of the utime field from the `/proc/<pid>/stat` file, respectively, for various values of ϵ. The values of ϵ in Figure 2.3 were chosen to overlap those used in the security evaluation depicted in Figure 2.1b. The ϵ values tested in Figure 2.4 were chosen based on our simulation of the software-keyboard side-channel attack of Lin et al. [94], which we conducted on a Nexus 4 smartphone running Android 5.1 with kernel 3.4.0; based on this simulation, we estimated that ranging ϵ over $1/2 \leq \epsilon \leq 5$ would result in curve similar to or better (with lower accuracy) than that

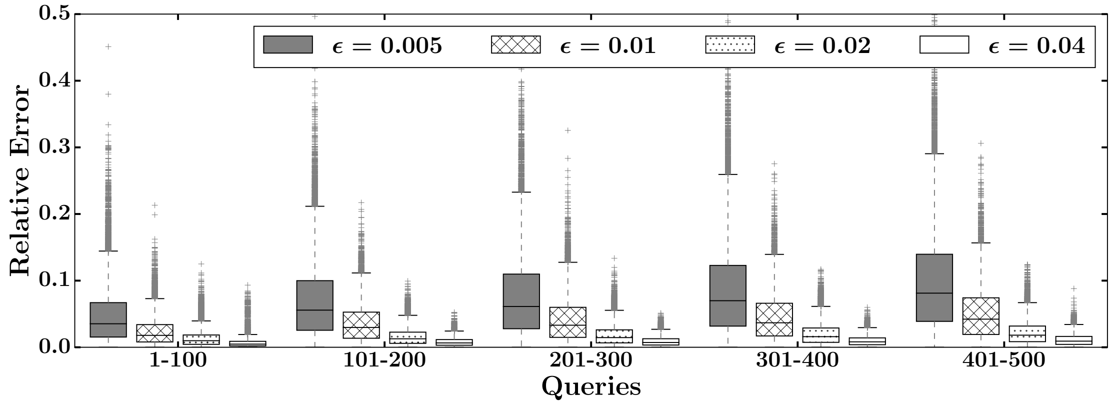


Figure 2.3: Relative error for drs field under nearest invariant reestablishment

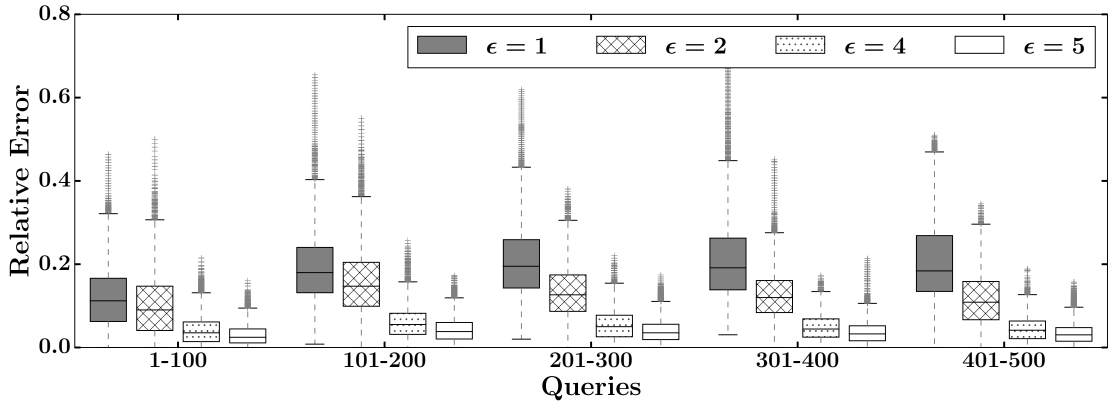


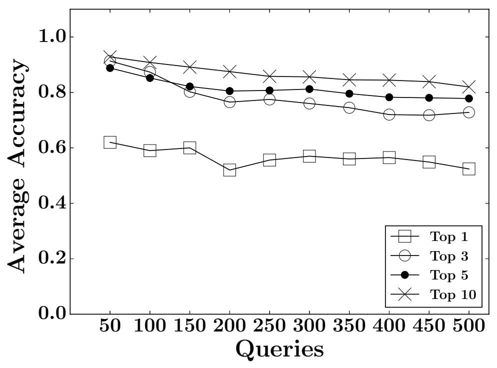
Figure 2.4: Relative error for utime field under nearest invariant reestablishment

in Figure 2.1a.[7] In the tests in Figure 2.4, the utime field was queried every 50ms while a video game was running. These graphs suggest that the relative error is typically modest, e.g., with a third quartile of $< 15\%$ in Figure 2.3 and $< 30\%$ in Figure 2.4, though outliers can be large.

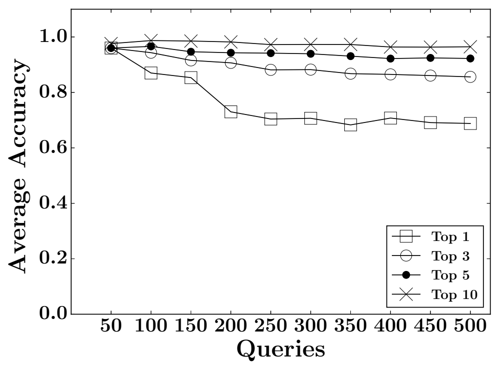
2.4.2.2 Rank Accuracy of `top`

The utility `top` is used by Linux administrators for performance monitoring and diagnosis. By reading `procfs`, `top` displays system information like memory and CPU usage of running processes. The processes are ranked by `top` according to a chosen field. In this section, we evalu-

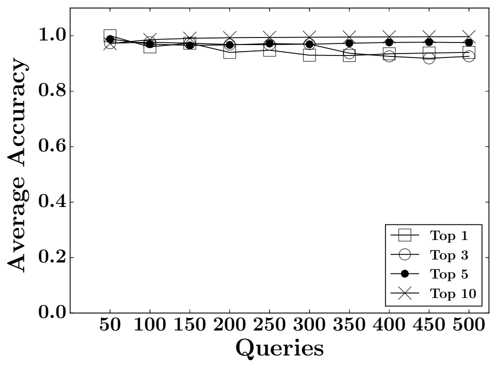
[7] Lin et al. reported querying the utime field of the software keyboard process every 100ms to detect its increase. With very rapid typing, the utime field in our tests increased less than 3 (jiffies) per 100ms interval, on average.



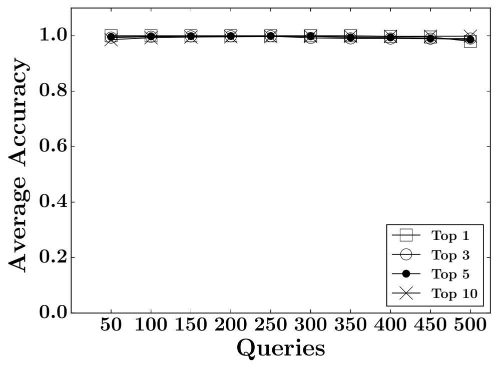
(a) $\epsilon = 0.005$



(b) $\epsilon = 0.01$



(c) $\epsilon = 0.02$

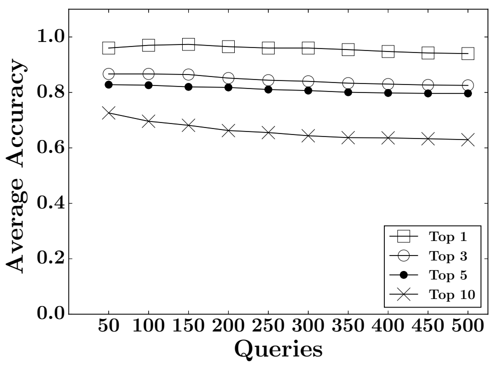


(d) $\epsilon = 0.04$

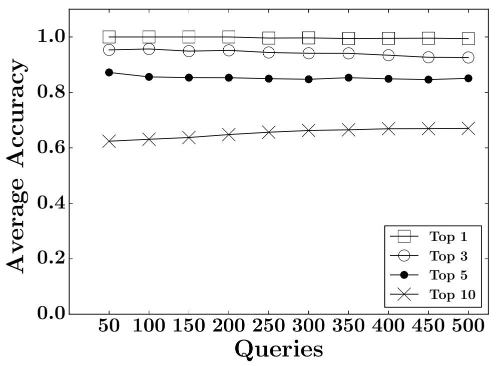
Figure 2.5: Average rank accuracy based on `RES` field

ate the utility of `dpprocfs` by measuring the rank accuracy of `top` when run using `dpprocfs` in place of the original `procfs`.

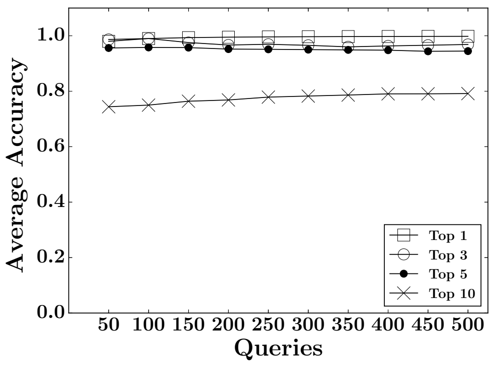
To measure the rank accuracy, we ran two `top` processes on one computer. These two `top` processes were started at the same time and updated information with the same frequency (every two seconds in our tests). The only difference was that one `top` process read from `dpprocfs` (with heuristic invariant reestablishment), and the other read from `procfs` in its original form. To control the test workload in each experiment, we ran a set of ten processes doing floating-point computations continually during each test. The number of memory pages allocated by each process to store its array of floats was scaled linearly across the ten processes: the first process allocated an 80MB array, the next process allocated a 95MB array, and so on up to the tenth process, which allocated a 215MB array. Similarly, the processes were configured with linearly scaled `nice` values ranging from -19 (highest priority) through -1 (lowest).



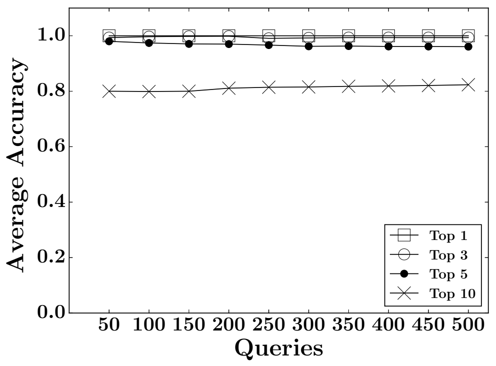
(a) $\epsilon = 1$



(b) $\epsilon = 2$



(c) $\epsilon = 4$



(d) $\epsilon = 5$

Figure 2.6: Average rank accuracy based on `%CPU` field

Let $\mathsf{R}(k)$ and $\mathsf{R}'(k)$ be the set of top k processes displayed by the two `top` programs. The *top-k accuracy* is defined as $\frac{1}{k}|\mathsf{R}(k) \cap \mathsf{R}'(k)|$. Figure 2.5 shows the average rank accuracy for various values of k when processes were ranked by the `RES` field. The `RES` field is read from `/proc/<pid>/statm`, calculated as $\mathsf{filePages}+\mathsf{anonPages}$, and represents the physical memory usage of the process. Figure 2.6 shows the average rank accuracy when processes were ranked by the `%CPU` field, calculated as $(\mathsf{utime}[i] - \mathsf{utime}[i-1])/(\mathsf{uptime}[i] - \mathsf{uptime}[i-1])$.

Several observations from Figure 2.5 and Figure 2.6 are worth noting. First, `top` retains much of its ability to rank processes by these measures; e.g., even for the lowest values of ϵ tested (Figure 2.5a and Figure 2.6a), the top-5 ranks remained roughly 80% correct on average through the tests. Second, whereas the top-10 rank is generally more accurate than the top-1 rank in Figure 2.5, the reverse is true in Figure 2.6. This occurs because while the memory usage of the ten test processes was scaled linearly, our linear scaling of `nice` values caused the actual

`%CPU` to drop off super-linearly. So, for example, the average difference in `%CPU` values for the processes with `nice` values -19 and -18 was much larger than the average `%CPU` difference between processes with `nice` values -3 and -1.

2.5 Discussion

Security limitations. Since we do not noise every kernel data-structure field that is used to serve `procfs` queries, there remains the possibility that such fields might reveal information about the true values of noised fields. This could occur either because the unprotected fields are related to those noised fields by invariants that our techniques did not find (see Section 2.3.2) or because those relationships are only statistical (but not invariant). It will therefore be necessary to extend the scope of our protections to other fields as new `procfs` storage side-channel attacks are discovered or, in the limit, that all kernel data-structures used to generate `procfs` contents be protected. As additional fields are brought under the protections of `dpprocfs`, the invariants that are reestablished on those values will need to be expanded appropriately.

Similarly, the value of ϵ used to protect a field might need to be updated as new attacks involving that field are discovered. As shown in Section 2.4.1, the value of ϵ may need to differ from one field to another. The magnitude of ϵ needed for a field will be correlated with the variation of that field and the number of queries over which protection needs to be provided, since as the number of queries grow, presumably so might d^* (between the actual field values and another from which it should remain indistinguishable).

Utility limitations. As the number of `procfs` queries grows, the amount of noise added to the kernel data-structure fields used to generate the `procfs` outputs grows (see Section 2.2.3 and Section 2.3.1). Although the amount of noise only grows logarithmically as the number of queries grows, the malicious user can still make a lot of queries to decay the utility of `procfs` outputs. To slow this decay, it may be necessary to rate-limit the queries that involve each field or to limit the number of such queries from any one user.

It is possible to provide a separate protection domain for each user. Under this design, a separate `privfs_struct` is maintained per querying user for each `procfs` field (as suggested in Section 2.3.1). Each user can have her own view of the noised `procfs`, and the decay of her `procfs` outputs won't affect other users' usage of those outputs. However, colluding users might be able to weaken the protection of `dpprocfs`.

It may eventually be necessary to "reset" the d^*-private mechanism associated with a field, particularly for a field associated with a long-running process. If the "reset" window is long enough to cover the duration of the sensitive event, this sensitive event can still be protected by our d^*-private mechanism. Typical durations of sensitive events range from milliseconds to seconds. For example, the duration of a keyboard typing event is less than a second and the duration of a browser page-loading event is several seconds. The system administrator can choose a proper "reset" window size according to the type of attacks faced by the system. The "reset" action can also be triggered by restarting the process itself, since restarting a process also refreshes its associated kernel data structures. (This is also beneficial for performance and reliability [47].)

Alternative solutions to `procfs` side channels. An alternative to adding noise in `procfs` outputs is to isolate mutually distrusting processes into different namespaces so that they cannot read each others' private `procfs` files. For example, Linux containers[8] isolate multiple applications from each other using `PID` namespaces in the kernel. While useful in hosting services such as modern PaaS clouds, Linux containers are less suitable in personal computing environments (e.g., Android devices and desktop computers) since sharing between different software applications are necessary in these single-user settings. Without a shared `procfs`, applications that need accesses to these system statistics—e.g., most traffic- and system-monitoring apps on Google Play, as well as the sysstat utilities[9]—will no longer work.

Another approach to defend against `procfs` side channels is to detect suspicious behaviors based on the frequency of `procfs` queries. Attackers usually make `procfs` queries in a high

[8]`https://linuxcontainers.org/`

[9]`http://sebastien.godard.pagesperso-orange.fr/`

frequency. Otherwise, they might miss the opportunity of capturing sensitive events. For example, Memento [82] reads the data resident size of the browser process from `procfs` in a loop to detect changes of that data field. This reading frequency is easily more than one thousand times per second. In contrast, the `top` command updates once every three seconds in the default setting. It is possible to set a threshold of `procfs` reading rate to differentiate malicious behaviors from normal ones. Some previous work [169] implements such detection method to defend against side channel attacks. Our d^*-private mechanism can be augmented with this frequency-based detection method. The ϵ value can be reduced as the `procfs` reading rate increases so that it is harder for the attackers to get useful information.

Extensions to other storage side channels. We believe our proposed method can be extended to other storage side channels such as those associated with mobile sensors (e.g., [30, 100, 165, 31, 11, 104, 142]). However, it is unclear how adding noise, e.g., to smartphone gyroscopes, will affect the usability of the apps that rely on their readings.

2.6 Summary

In this chapter we have reported on the design, implementation, and evaluation of `dpprocfs`, a modification to the `procfs` pseudo file system that suppresses storage side channels. The innovations that are central to our design include: (i) framing the side-channel problem as one of achieving d-privacy for continual data release, and defining an appropriate distance d^* for instantiating d-privacy for this scenario; (ii) generalizing a differentially private mechanism for the continuous release of binary values to the d^*-privacy goal we set forth; (iii) recognition of the systems difficulties that can arise when adding noise to `procfs` outputs, and an invariant reestablishment framework to address those difficulties; and (iv) a working implementation of `dpprocfs`, coupled with an evaluation that shows it can simultaneously defend against known storage side-channel attacks while retaining the utility of `procfs` for monitoring and diagnosis.

CHAPTER 3: POPSICL: MASKING SERVER IDENTIFIERS TO ENFORCE SERVER ANONYMITY[1]

Monitoring of online activities is a fact of life for many users, be it by, e.g., an employer to detect activity that is inconsistent with corporate policy or a government to monitor sites accessed by its citizens. While encryption is a first line of defense against monitoring, many identifiers (including the DNS names and IP addresses) still reveal the identity of the servers accessed by users. Numerous techniques have thus been developed to support *server-anonymous* access, i.e., access to a server in a way that hides the server identity from a monitor. The socalled “hidden service”, supported by Tor [55], is an example of a technology that enables server-anonymous access.

The growth of large hosting infrastructures such as compute clouds and content distribution networks (CDNs) has opened new opportunities for deploying server-anonymous systems (see Section 3.1 for a discussion). Because these hosting platforms serve content from many tenant servers, intermingling controversial server content or anonymizing proxies among them (i.e., as other tenants) can make it more difficult or expensive for a monitor to disambiguate which tenant server a client is accessing. The vast resources and connectivity available via these infrastructures can also expand the capacity of server-anonymous systems. However, prior attempts (of which we are aware) to leverage these infrastructures to support server anonymity have been designed for an oblivious cloud operator that (at best) provides no specific support for server anonymity (e.g., [28, 69]). Changes are thus typically hoisted onto the client users of these systems, who often lack the permissions, trust, or know-how to do so.

In this chapter, we instead propose a design for server-anonymous communication to tenant servers of a cloud that leverages support and cooperation of the cloud provider and, in doing so,

[1]This chapter is excerpted from previously published work [162]

avoids requiring any changes to client software. Our central innovation to enable this capability is a PoPSiCl (pronounced "popsicle"), a Personalized Pseudonym for a Server in the Cloud. Specifically, a PoPSiCl is a domain name with the following properties: First, it is *personalized*: A PoPSiCl can be used to access the tenant server only by the client for which the cloud generated it. Second, it is a *pseudonym*: A PoPSiCl is a persistent identifier that the client to whom it is issued can use to access the server over time (e.g., by bookmarking it). Moreover, the cloud protects the identity of the tenant server accessed using this PoPSiCl from an attacker who can both observe the client's communication with the cloud and probe the cloud as another client or tenant server itself. Though the cloud is trusted in our design, note that today the cloud is already typically trusted with knowing which users frequent a tenant server. Even if the user connects to a tenant server using an anonymizing service such as Tor, the cloud can access any identifying information the user provides to the tenant server, either intentionally (e.g., an email address) or not (e.g., HTTP cookies or browser fingerprints [116, 118, 32][2]). In such cases, our trust in the cloud does not substantially increase the trusted computing base for user privacy.

A design goal for PoPSiCls is that they can be implemented by the cloud operator in a way that is unobtrusive to their tenants or their tenants' clients. Specifically, we demonstrate an implementation of PoPSiCls to support private TLS accesses to web servers in the cloud that has the following features:

- Our implementation requires no changes to client-side software and works with all major web browsers. This stands in contrast to most work on anonymous access to servers (see Section 3.1) that requires the installation of proxies on client computers or the installation of a custom browser (e.g., Tor). Requiring no changes to the client is important for users who lack either the permissions needed to modify their client platforms (as an employee using a company-owned computer might) or the willingness to do so (e.g., since even security software is often riddled with vulnerabilities [129, 46, 139]).

[2]The Tor Browser tries to mitigate browser fingerprinting by restricting browser features, but this requires reacting to new attacks as they are discovered [118, 32] and has an inevitable impact on usability.

- The only changes in user experience for supporting use of PoPSiCls is the use of client-side TLS certificates to support TLS connections, and a visit to a cloud-operated *PoPSiCl store* to obtain a PoPSiCl and the client-side certificate for a tenant server prior to her first (server-anonymous) access to that server.
- A tenant server requires some changes to its OS and, if the server is a web server, minimal other changes that can be hidden within high-level web programming frameworks like Ruby on Rails. So, these changes can be packaged either in a platform-as-a-service (PaaS) cloud offering or a virtual machine (VM) image for deployment to an infrastructure-as-a-service (IaaS) cloud, without imposing on web-content developers.

Supporting PoPSiCls does impose more substantially on cloud infrastructure, notably through the establishment of the PoPSiCl store; in dynamic generation of switching rules to configure software-defined networking (SDN) switches in the cloud infrastructure; and, as mentioned above, in tenant server operating systems. We detail the changes needed to OpenStack and Linux to implement the needed functionality. Our implementation therefore most directly reflects how an IaaS cloud operator could deploy and support PoPSiCls, with OS modifications provided through PoPSiCl-enabled virtual-machine images. We envision that a cloud operator might be motivated to support PoPSiCls as one component of a larger "security as a service" offering, charging tenant servers for PoPSiCl use, perhaps per PoPSiCl or even per PoPSiCl-based connection.

We have used CloudLab (`https://www.cloudlab.us/`) to characterize the performance impact of PoPSiCl usage, versus regular (non-server-anonymous) web browsing over TLS. Our results show that our design introduces modest overhead to server access latency and throughput, and is capable of scaling to large numbers of users, should PoPSiCl use catch on. We also show that the access latency of our implementation is considerably better than proxy-based systems such as Tor, which also enhance privacy for server access, as discussed above. (We caution the reader, however, that the threat model and protections offered by PoPSiCls are different than those for which systems like Tor were designed, as we will discuss in Section 3.1 and especially Section 3.7.)

The rest of this chapter is structured as follows. We provide background in Section 3.1, and outline the principles behind our design in Section 3.2. Section 3.3 contains our high-level system design, and Section 3.4 describes our current implementation. We evaluate that implementation in Section 3.5. In Section 3.6, we extend our design to address some forms of traffic analysis. We discuss the limitations of our design in Section 3.7. Finally, we summarize this chapter in Section 3.8.

3.1 Background

The goal of our design of PoPSiCls is to provide *server anonymity* (elsewhere called *recipient anonymity* [126] or *recipient untraceability* [36]) against network attackers. That is, a network attacker can observe that a client is initiating communication with a server in the cloud, but the attacker is unable to determine the specific server with which the client is communicating. In this context, a PoPSiCl is an *implicit address* [126] for a tenant server; moreover, it is *visible* in that its reuse to reconnect to the server is evident—both to the cloud, which can use the PoPSiCl to route the client to the physical machine currently hosting the tenant server, and to the attacker. The servers that appear to the adversary to be the possible targets of the client (i.e., the server's *anonymity set* [36]) is the set of all tenant servers in the same cloud datacenter as the target.

The most widely used methods to achieve server anonymity today are based on proxying (e.g., Tor [55], Psiphon (`https://psiphon.ca/`), and early versions of the Anonymizer [27]) or VPNs (such as the current Anonymizer, `https://www.anonymizer.com/`). Unlike these systems, PoPSiCls are not supported in our design through proxying or VPN tunneling. In particular, communication to a PoPSiCl is encrypted by the client and decrypted only by the tenant server in the cloud (vs. at a proxy or tunnel endpoint), leaving few opportunities for accidental leakage. Moreover, as we will show, proxies can become performance bottlenecks, and so our design scales better to heavy usage.

Variations on the goal of server anonymity have been studied in several forms, often under the rubric of *censorship resistance*. Like our design, several in this space leverage infrastructure providers explicitly (clouds, CDNs, or ISPs) to hide the server with which a client is trying to interact.

- In domain fronting [69], a client connects to a CDN edge server or reflector web application run in the cloud via a *front domain* other than the *hidden domain* of actual interest. The edge server or reflector then inspects the plaintext payload (e.g., the HTTP `Host` header) to discover the hidden domain and retrieves it for the client. A PoPSiCl can be viewed as a front domain, though the mapping to its hidden domain is maintained by the cloud operator and managed without inspecting the client's payload or, more to the point, without decrypting it, which is better for client/tenant security.
- CacheBrowser [78] enables a website's content to be retrieved from any CDN edge server without a DNS resolution, leveraging the assumption that it is untenable for censors to block IP addresses of CDN edge servers due to the collateral damage it would cause. However, this system still exposes the true server domain in the SNI field (see Section 3.2.1) and so is not truly server-anonymous. Recent improvements [174] rectify this concern, but do so in a way that is incompatible with some CDNs. In either case, these solutions work only for cacheable content.
- CloudTransport [28] repurposes cloud *storage* to implement interactive communication to a server in a way that will evade common censorship techniques.
- Telex [160] enables friendly on-path ISPs to recognize "tagged" traffic addressed to uncensored websites and divert it to the censored websites for which it is really intended. Tagging is implemented in the SSL handshake protocols, by embedding a tag into the random value field in the `ClientHello` message.
- LAP [80], Dovetail [137], HORNET [37], and PHI [38] are network-layer protocols that aim to provide low-latency and high-throughput anonymous communication. In these pro-

tocols, the source and destination addresses are encrypted so that the intermediate routing node only knows its adjacent nodes in the path (similar to Tor).

- There have been several proposals to host Tor relays in clouds [85, 109]. Moreover, systems like Tor have tended to be vulnerable to censors because users are connected to a small set of entry points that can be blocked. So, prior works have proposed to reduce the disclosure of IP addresses of Tor entry points through Tor bridges (a variation of keyspace hopping [68]; see `https://www.torproject.org/docs/bridges`) and, through the deployment of the Tor Cloud project (`https://cloud.torproject.org/`), to run Tor bridges inside clouds.

As they relate to our work, all of the above approaches require modifying client-side software. In contrast, our design requires no client-side software changes at all (albeit while requiring changes to infrastructure, as many of the above designs also require). That said, we stress that in contrast to some of the works above, our goal here is not censorship resistance, per se, but rather server anonymity, as the assumption of a trustworthy and cooperative cloud is somewhat at odds with the former. We discuss this issue further in Section 3.7.

3.2 Design Principles

In this section we detail the security (Section 3.2.1) and usability (Section 3.2.2) goals of our system.

3.2.1 Security

Threat model. We begin our discussion of the security principles of our design by recalling our threat model. A cloud hosts tenant servers, to which clients can connect (e.g., using TLS). As is the case today, the cloud operator is trusted by both tenant servers and their clients. We are concerned with enabling a client to connect to a tenant server without divulging to an attacker the tenant server to which it is connecting. The client machine is trusted, as is the tenant server to which it connects. Other clients and other tenant servers are not trusted in the context of this client-server interaction; i.e., the identity of the server to which the client connects should remain hidden despite the efforts of other clients and other tenant servers. We also allow the attacker to capture and manipulate all traffic outside the cloud premises, including traffic to or from the client, but traffic within the cloud is invisible to the attacker (except if the attacker controls the source or destination of the traffic).

Our threat model gives the attacker many opportunities to observe the identity of the tenant server to which a client connects in today's clouds (see Figure 3.1). First, the DNS resolution of the server domain name can reveal that domain name to the attacker. Second, the IP address to which the client connects will be visible to the attacker; the attacker can then connect to this IP address itself to see what the server provides, or simply use this IP address to determine the server's identity from a preassembled database (like a reverse DNS lookup). The connection process itself can offer additional opportunities for the attacker to identify the server; in particular, a TLS connection exposes the server domain name in the Server Name Identification (SNI) field that the client sends, and in the certificate that the server provides to the client. The certificate also exposes the server's public key, which can be matched against the public keys in certificates obtained by other clients.

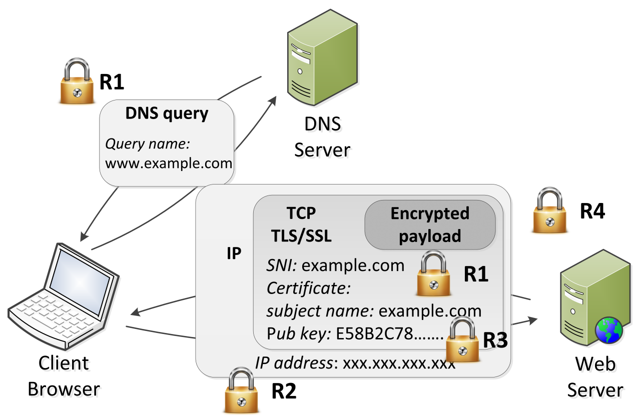


Figure 3.1: In our threat model, the server identity can leak via the client's DNS query, the SNI field of the client-to-server TLS connection, the server IP address, or the server public key

We leave several types of attack outside our scope, relying on orthogonal defenses to address them. For example, we assume the security of TLS and that the attacker can impersonate neither any cloud-provided service or tenant server that it does not control (by virtue of not having the needed server private key), nor any client that it does not control (by virtue of not having the client private key).

Also outside our scope are connection-level features that can divulge indications of the server involved in the connection, such as have been used in TCP fingerprinting (e.g., [70]), TLS fingerprinting (e.g., [108]), or website fingerprinting (e.g., [65, 158, 122]). Such features include the number of servers to which the client connects, the timing connections to relative to one another, connection volume patterns, the direction of the connections, etc. That said, we have made initial progress toward a framework for traffic-analysis defense, as we will discuss in Section 3.6.

Security principles. To achieve server anonymity in the threat model described above, several steps are necessary. The first is to replace the server's domain name with a different domain name—the PoPSiCl—everywhere it is visible to the attacker. So, it will be necessary to cause the PoPSiCl to be used in the client's DNS lookup, the TLS SNI field, and the certificate that the tenant server sends to the client. In our system, the PoPSiCl takes the form *str*`.popsicls.com` where `popsicls.com` is the domain name of the cloud and *str* is a string that represents the PoPSiCl prefix. So, for example, `1f5qz7nfhj1uworr7laduh9fen.popsicls.com` might be a PoPSiCl. Of course, the PoPSiCl prefix *str* must be generated for this client in a way that prevents the attacker from correlating it with PoPSiCls generated for other clients to access the same tenant server.

R1: *The PoPSiCl for a client to access a tenant server is independent of the PoPSiCl generated for other clients, and the PoPSiCl is used in place of the tenant server's domain name everywhere that domain name appears in client communication.*

The PoPSiCl is intended to be a long-lived identifier that the client can use to access the server. To that end, the client uses the PoPSiCl just like any other domain name—by performing a DNS lookup on it to obtain an IP address to which to address network packets. To prevent the attacker from using this IP address to identify the server, however, the DNS resolution must produce a *pseudo-address*, which is a different IP address that the one the server actually uses. More specifically, a pseudo-address is a publicly routable IP address that is part of the IP address block allocated to the cloud, so that a packet addressed to the pseudo-address will eventually reach a switch in the cloud datacenter. However, the pseudo-address should be otherwise unrelated to the actual IP address of the tenant server.

R2: *The pseudo-address to which a client addresses packets for the tenant server (and from which return packets arrive to this client) is independent from the actual network endpoint (i.e., IP address) of the tenant server in the cloud.*

A pseudo-address can be used in our system to establish a TLS connection to the tenant server associated with the PoPSiCl. TLS connection establishment introduces other potential identifiers that might be used to deanonymize the tenant server, particularly the public-key certificate for the server. As such, the tenant server should use a different public key per client.

R3: *In a TLS connection setup with a client that is accessing the server using a PoPSiCl, the tenant-server public key used was generated independently of the server public keys used in its TLS connections with other clients (regardless of whether those clients use PoPSiCls to access the server).*

There remains the risk that the attacker who observes the pseudo-address could simply connect to that pseudo-address itself and identify the server based on the content returned. To prevent this possibility, the client for which the PoPSiCl was created should be the only one that can complete a secure connection using it.

R4: *A tenant server completes a TLS connection setup with a client using a PoPSiCl only if that PoPSiCl was registered for use by that client, with this tenant server.*

3.2.2 Usability

Here, *usability* refers to the operational impact of PoPSiCls on all actors in the cloud ecosystem—the cloud operator, cloud tenants, and the tenant's clients. The approach we take in our design of a system to support PoPSiCls is to place a larger usability burden of deploying PoPSiCls on those groups of actors with greater technical capabilities. Major cloud operators arguably offer the highest concentration of technical capability, as their datacenter functioning is integral to all tenants' availability and security. As such, they will bear the greatest burden in supporting our design. Tenants who deploy servers to the cloud typically require at least a knowledge of how to populate a server with content, and so we will limit our design to small modifications to that process (at least in the case of web servers). Finally, we presume the tenants' clients might be driven by wholly nontechnical users (e.g., via web browsers) who might not have the permissions needed to install software on their computers (e.g., as a user of a corporate-controlled computer might not) or a willingness to do so (e.g., due to the vulnerabilities that such software can introduce [129, 46, 139]). So, we place a priority on minimizing client-side changes.

Treating these groups in reverse order, then, our first requirement is that changes to clients be very limited.

R5: *PoPSiCls are usable with no changes to client-side software, including no browser extensions or add-ons, in the case of web clients. While PoPSiCls are visible to clients and may require some adaptation of client user procedures (in the case of typical web browsing, for example), these adaptations are already supported by the dominant software clients in the market.*

The primary operational adaptations required by our design for a web user, for example, are the following. First, our design involves the use of client authentication via a client-side certificate in TLS. While not without its issues [123], support for client authentication is already in all major browsers and is in use by large communities (e.g., in Estonia, due to its national PKI initiative [123], and by MIT faculty, staff, and students to access some web services[3]). Second, a user must take an additional, online step to obtain ("register") a PoPSiCl for future accesses to a website. We will describe PoPSiCl registration in Section 3.3.1.

For tenant servers, who might range from large, well-staffed organizations to small online vendors, we allow changes to the software they use but require that those changes can be made largely "invisible" to them, if they so choose.

R6: *Changes to tenant servers can be hidden so that they do not impose on server content creation. For example, changes involving the tenant-server operating system (OS) or content-*

[3]`http://ist.mit.edu/certificates/guide`

programming frameworks can be packaged within a virtual-machine image that respects existing application programming interfaces (APIs). As such, a tenant-server creator should be able to "port" his content to this VM image with minimal effort.

Our design intrudes on tenant servers primarily by requiring specific OS-level changes, discussed in Section 3.3.2. These can be packaged within a VM for deployment to Infrastructure-as-a-Service (IaaS) clouds. Alternatively, in a Platform-as-a-Service (PaaS) cloud, the OS is managed by the cloud operator, and so these changes would be invisible to the tenant server. In addition, some defenses specific to HTTP servers described in Section 3.3.3 and an optional extension described in Appendix 3.6 induce very minor additional changes to modern web programming frameworks (such as Ruby on Rails).

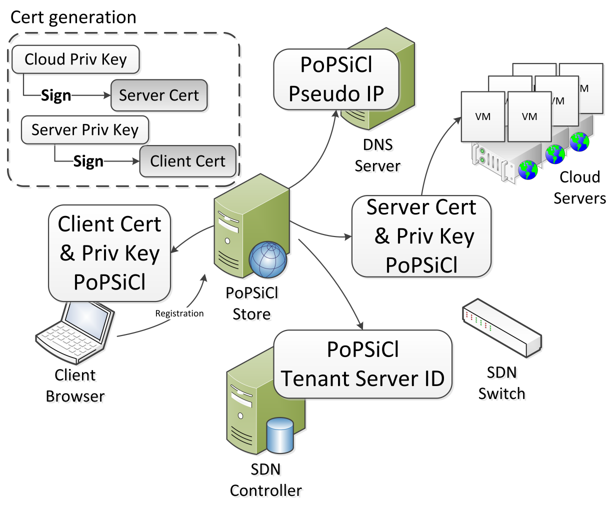
We allow for our design to impact cloud operators more directly. Again, though, cloud operators are the most technically savvy and so presumably the most capable of accommodating such changes.

3.3 Design

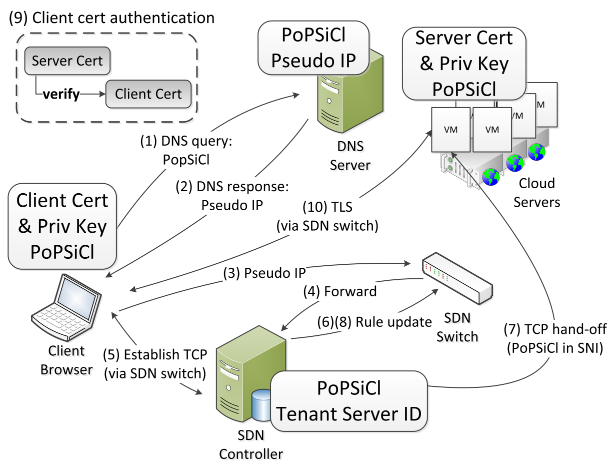
In this section we describe the design of a system to enable a cloud operator to implement PoPSiCls for its tenants and their clients. In order to use a PoPSiCl to access a tenant server, a client must first *register* the PoPSiCl, a process described in Section 3.3.1. The mechanisms supporting the use of a PoPSiCl to connect to a tenant server are described in Section 3.3.2. Adaptations specific to supporting HTTP clients using PoPSiCls are described in Section 3.3.3. Finally, how our design achieves the requirements laid out in Section 3.2 is the topic of Section 3.3.4.

3.3.1 Registering a PoPSiCl

The registration of a PoPSiCl is a user-initiated process, involving connecting to a particular cloud-operated service, the PoPSiCl store, using a web browser. The connection should employ TLS, though need not require a password login or any other form of client authentication. Rather, TLS is employed here simply to protect the privacy of the user. Upon accessing the PoPSiCl



(a) Registration, described in Section 3.3.1


(9) Client cert authentication
Server Cert
verify
Client Cert
PoPSiCl Pseudo IP
DNS Servers
Server Cert & Priv Key PoPSiCl
VM
Cloud Servers
(1) DNS query: PopSiCl
(2) DNS response: Pseudo IP
(10) TLS (via SDN switch)
Client Cert & Priv Key PoPSiCl
(3) Pseudo IP
(4) Forward
(6)(8) Rule update
SDN Switch
(7) TCP hand-off (PoPSiCl in SNI)
Client Browser
(5) Establish TCP (via SDN switch)
PoPSiCl Tenant Server ID
SDN Controller

(b) Access, described in Section 3.3.2

Figure 3.2: Steps for registering a PoPSiCl (Figure 3.2a) and then using it (Figure 3.2b)

store, the user is presented with a web form to indicate the domain name, say `tenantA.com`, for which she wishes to register a PoPSiCl. Since we are trusting the cloud operator, we assume that if `tenantA.com` is not, in fact, hosted by the cloud, then it will decline the registration.

If `tenantA.com` is one of its tenants, then the PoPSiCl store takes the following actions (see Figure 3.2a).

(i) The PoPSiCl store first creates a new PoPSiCl for `tenantA.com`, of the form *str*`.popsicls.com`, where *str* denotes a string of characters allowed in domain names. It then creates and exports a DNS record for *str*`.popsicls.com` that maps this PoPSiCl to one or more publicly routable IP addresses in the address ranges allocated to `popsicls.com`, to which we refer as pseudo-addresses. As we will see in Section 3.3.2, these pseudo-addresses are addresses of SDN controllers in the same datacenter (region) as `tenantA.com`. The PoPSiCl store also informs these SDN controllers that this PoPSiCl corresponds to `tenantA.com`.

(ii) The PoPSiCl store creates a new public/private keypair for use by `tenantA.com` and binds the public key to *str*`.popsicls.com` in a TLS server certificate signed by the PoPSiCl store. The PoPSiCl store delivers this private key and server certificate to the tenant server. It also delivers to the tenant server a root certificate for authenticating client certificates in TLS connection attempts to *str*.`popsicls.com`. For our purposes, it suffices for the newly generated server certificate to be used as this root certificate, as well. (Alternatively, a different root certificate generated for this specific purpose could be used.) This step assumes a tenant server capable of receiving this information. As described in Section 3.4, the Nginx web server already supports this capability, for example.

(iii) The PoPSiCl store generates a public/private keypair for the client to use to authenticate itself when connecting to *str*.`popsicls.com`; creates a certificate for this public key that can be verified using the root certificate of Step (ii); and returns the private key, public-key certificate, and PoPSiCl to the user. The user then saves this information and takes whatever steps are necessary to permit its TLS client to make use of this key when connecting to *str*.`popsicls.com`. For example, if the client is a web browser, then the client might bookmark the PoPSiCl and import this key pair and certificate into the browser. This step is already supported by major browsers.

Prior to connecting to *str*.`popsicls.com`, the client must also be configured with the PoPSiCl store as a certificate authority (CA) for TLS server certificates. In this way, the client will accept the tenant server's certificate (see Step (ii)) during TLS connection setup.

A user's first (or any) connection to the PoPSiCl store could be used to obtain a PoPSiCl for the PoPSiCl store, so that subsequent accesses to the PoPSiCl store can be hidden from an attacker.

3.3.2 Connection Establishment

Via the steps described in Section 3.3.1, the client is in possession of a PoPSiCl for the server to which it wants to connect. To connect to this server, the client performs a DNS lookup on the PoPSiCl, as it would any other server domain name (steps 1–2 in Figure 3.2b). Because the PoPSiCl is of the form *str*.`popsicls.com` where `popsicls.com` is the cloud domain name, the cloud ultimately provides the IP address returned to this DNS query. The IP address provided by the cloud is not the actual IP address of the machine hosting the tenant server (this would violate **R2**), but rather must be independent of it. One option would be to return the IP address of a (reverse) proxy in the cloud that relays client requests to the tenant server associated with the PoPSiCl (and responses back to the client); this design has similar security properties to ours, but as we will see in Section 3.5.2, this proxy will become a bottleneck. Rather, in our design, the DNS query is resolved to a publicly routable IP address of an SDN controller in the same datacenter (region) as the tenant server corresponding to *str*`.popsicls.com`; we refer to this IP address as a pseudo-address for the tenant server.

Let the pseudo-address be denoted by *pseudo-IP*, and let *client-IP* and *client-port* denote the source IP address and source port of the first packet that the client sends to *pseudo-IP* (a TCP SYN), when it arrives at the cloud switch (step 3 in Figure 3.2b). *client-IP* and *client-port* need not be the actual IP address and port of the client; rather, these could instead be the address and port of a network-address translator (NAT) or proxy between the client and the cloud. Unless it has a higher-priority rule (see below) that matches specifically this source IP address (*client-IP*), source port (*client-port*), destination address (*pseudo-IP*), and destination port (denoted *server-port*), the switch forwards the packet using the default routing rule for this destination (*pseudo-IP*). Since in our design, the *pseudo-IP* is set to an IP address of the SDN switch controller in the cloud datacenter, this TCP SYN packet is forwarded to the controller (step 4 in Figure 3.2b). In this case, the controller responds with a TCP SYN-ACK and completes the TCP connection with the client (step 5 in Figure 3.2b).

After completing the TCP connection, the client then launches the TLS handshake. At this point, the controller learns *str*`.popsicls.com` from the `ClientHello` SNI field, with which it can look up the tenant server to which this PoPSiCl corresponds, say with IP address *tenant-IP*. The controller then takes the following steps (without responding with a `ServerHello` message), in order:

(i) The controller installs two new rules in the switch (step 6 in Figure 3.2b). One matches packets with source address *client-IP*, source port *client-port*, destination address *pseudo-IP*, and destination port *server-port*; this rule simply drops any such packet silently. The second matches packets with source address *tenant-IP*, source port *server-port*, destination address *client-IP*, and destination port *client-port*; changes the source address to *pseudo-IP*; and forwards the packet toward *client-IP*.[4] These rules have higher priority than any other rules that apply to the same packets.

(ii) The controller then transfers the TCP connection state to the tenant server OS (step 7 in Figure 3.2b), including the buffer containing the `ClientHello`, and so the tenant server picks up the TCP session where the controller left off (by responding with a `ServerHello`). Our implementation of TCP connection transfer uses the technique in the `tcpcp` tool [8].

(iii) The controller then replaces the drop rule installed in Step (i) above (i.e., matching packets with source address *client-IP*, source port *client-port*, destination address *pseudo-IP*, and destination port *server-port*) to instead change the destination address of any matching packet to *tenant-IP* and then to forward the packet toward *tenant-IP* (step 8 in Figure 3.2b).

The TLS connection establishment that the tenant server continues with the client requires the client to present a client certificate that can be verified by the server certificate for this PoPSiCl (step 9 in Figure 3.2b). The tenant server received the server certificate for this PoPSiCl, and the client received this matching client certificate, in the PoPSiCl registration process (Section 3.3.1). If this client certificate is not sent, then the TLS connection fails; otherwise, the TLS connection can be established and communication proceeds as normal (step 10 in Figure 3.2b).

The above ordering of steps is chosen purposely. The drop rule is installed in Step (i) to drop any inbound messages from the client during the TCP state transfer, which could confuse the transfer process. The other rule in Step (i) is installed to ensure that the `ServerHello` and the following messages sent by the tenant server in Step (ii), when the TCP state transfer completes, are forwarded to the client with the *pseudo-IP* as their source addresses. The controller replaces the drop rule from Step (i) as described in Step (iii) to permit the connection between the client and tenant server to continue. Of course, it is possible that the controller does not finish Step (iii)

[4]The installation of this second rule assumes that return packets traverse this same switch. If they do not, then this second rule would need to be inserted into another switch that they will traverse.

before the client sends another message, in which case the drop rule from Step (i) will drop it. We leverage TCP's retransmission capabilities to overcome any such drops that occur.

A possible denial-of-service attack against our architecture is to overwhelm the SDN controller(s), which will handle all TCP connections established using PoPSiCls until they are handed off to their tenant servers. For this reason, the controllers must be defended using state-of-the-art denial-of-service defenses. That said, note that our design allows for multiple SDN controllers and switches, and load-balancing among them. We could allocate a pool of servers hosting SDN controllers and increase the number of serving controllers as the load increases.

3.3.3 HTTP-specific Mechanisms

PoPSiCls can be used to support any type of server accessed using TLS. Overwhelmingly, however, the most common example today is HTTP, and so in this section we address several issues specific to their use to support access to HTTP servers.

Same-origin policy and cookies. Our architecture for supporting PoPSiCls permits the browser to accurately track origins, i.e., to support its same origin policy [133], since the browser is provided a unique domain name (the PoPSiCl) per tenant domain. This same property also enables the browser to send cookies to (only) the right domains—avoiding a pitfall of some previous anonymous communication systems (e.g., early versions of the Anonymizer [27]). To prohibit tenants from setting cookies for the cloud domain (e.g., `popsicls.com`), the cloud operator should add `popsicls.com` to the public suffix list[5], just as is, e.g., `amazonaws.com` today.

Object hyperlinking. When a tenant web server accessed using a PoPSiCl serves hyperlinks to its own objects, their URLs should leverage the PoPSiCl as their domain name. Otherwise, the browser would retrieve these objects using a URL with a different domain name, causing them to be viewed by the browser as coming from a different origin. This could cause the web page to malfunction, or it could result in disclosure of the true domain name to an attacker. Fortunately, this is achieved easily by hyperlinking to relative URLs, or by authoring web content with the

[5]`https://publicsuffix.org/`

domain name inserted by a macro that is resolved to the PoPSiCl with which the current client is accessing the server.

Hyperlinking between servers requires additional attention, since one server should not learn the PoPSiCl that a client uses to access another server. First, to ensure that a client browser does not disclose the PoPSiCl that it uses for accessing a tenant server, say `tenantA.com`, to another server to which `tenantA.com` refers the client (i.e., in the HTTP `Referer` field), `tenantA.com` should set the referrer policy of its referring page to `no-referrer` or `same-origin`.[6] Second, to allow referrals to a tenant server, say `tenantB.com`, without disclosing the server's identity to our attacker, the cloud operator `popsicls.com` can support hyperlinking to it using a URL such as `https://linker.popsicls.com?tenantB.com/...`, where `linker.popsicls.com` is a cloud-operated server. Upon receiving the TLS connection from the client, `linker.popsicls.com` can look up the PoPSiCl that this client uses to access `tenantB.com` (authenticating the client using its client certificate) and then redirect the client browser to that PoPSiCl. (For reasons discussed below, however, `linker.popsicls.com` will have to apply additional policy before doing so.) If no such PoPSiCl exists, then `linker.popsicls.com` can simply redirect the client to `tenantB.com`. To support hyperlinking in this fashion, the PoPSiCl store should provide a client-side certificate and accompanying private key for the client to use to connect to `linker.popsicls.com`, during the client's first PoPSiCl registration (for a web server) at the PoPSiCl store.

Cross-origin attacks in browsers. A tenant server accessed using a PoPSiCl does not complete a TLS connection setup with a client other than the one that registered that PoPSiCl. An attacker who obtains a PoPSiCl in use by a client, say `1f5...fen.popsicls.com`, is thus unable to connect to it directly in an effort to retrieve content from it (and thereby deanonymize it). Through cross-origin side channels, however, the attacker could potentially infer the true server identity behind a PoPSiCl. For example, the attacker could set up a web server (not necessarily in the

[6]See `https://www.w3.org/TR/referrer-policy/`. As of this writing, the latest versions of Edge and Safari support an older draft of the referrer-policy specification, for which the referring policy should be set to `never`.

cloud) and, if it could convince the client browser to visit its server, could serve back to the client a hyperlink, say `https://1f5...fen.popsicls.com/`*path*, that uses the PoPSiCl as its domain name. An attacker script could then test if the browser successfully retrieved the object at this URL,[7] thereby inferring whether *path* is a valid path at the tenant server. Since this might be a distinctive pathname, the attacker could deanonymize the server this way.

To prevent such cross-origin attacks, the tenant server refuses requests for URLs containing the PoPSiCl `1f5...fen.popsicls.com` except from its own pages or via redirections from `linker.popsicls.com`. The tenant server enforces this property by requiring any URL utilizing `1f5...fen.popsicls.com` to be appended with a *capability*, specific to this PoPSiCl, that only itself and `linker.popsicls.com` can obtain. This capability is implemented as a random, unguessable string that must be encoded as a query string in any URL using `1f5...fen.popsicls.com`, so that it is always transmitted under TLS protection. It is created by the PoPSiCl store when `1f5...fen.popsicls.com` is first registered and is returned in the URL that the user is invited to bookmark. Both the tenant server and `linker.popsicls.com` are then permitted to retrieve the capability (from a cloud-operated database) when needed, for the purposes of producing URLs containing that PoPSiCl.

There remains a cross-origin attack that a web server with which the client is interacting can mount, to infer the PoPSiCl the client uses to contact a tenant server, say `tenantA.com`, provided that the adversary controlling the web server can simultaneously monitor the traffic from the client to the cloud. If the web server directs the client to retrieve `https://linker.popsicls.com?tenantA.com/...` and then monitors traffic for an interaction with `linker.popsicls.com` and then a connection using a PoPSiCl in close temporal proximity, then the attacker can infer that the client uses this PoPSiCl for `tenantA.com`. Fortunately, the intervening interaction with the trusted `linker.popsicls.com` provides an opportunity to mitigate this attack. For example, a reasonable policy might be for `linker.popsicls.com` to redirect the request to `https://linker.popsicls.com?tenantA.com/...`

[7]There are many ways to perform this test, e.g., by hyperlinking to an image and then testing the height of the image, which would typically differ depending on whether the image retrieval succeeded or failed.

to one that uses the client's PoPSiCl for `tenantA.com` only if the referrer site is trusted by `tenantA.com` and the client is also accessing the referrer site using a PoPSiCl. (Otherwise, `linker.popsicls.com` redirects the client to `tenantA.com`, sans PoPSiCl.) The referrer site can indicate compliance with this last condition to `linker.popsicls.com` by, say, appending the client's capability for the referring site to the referral `https://linker.popsicls.com?tenantA.com/...`, which `linker.popsicls.com` can check by looking up the capability and corresponding referrer in a database.

3.3.4 Design Principles, Revisited

In this section we revisit the principles outlined in Section 3.2 to describe how our design achieves them.

Security. Requirement **R1** is met by having the PoPSiCl store generate each PoPSiCl (specifically, the *str* part of *str*`.popsicls.com`) pseudorandomly. The tenant server to which this PoPSiCl corresponds is protected during the registration process by TLS (Section 3.3.1) and thereafter is accessible only to the cloud's SDN controller(s) (Section 3.3.2) in order to route connection traffic appropriately (and the tenant server itself, of course). The PoPSiCl is used by the client as any other domain name would be, and so it appears everywhere that the domain name would in the normal course of client communication—notably in DNS queries, the TLS SNI field, and the server TLS certificate.

Requirement **R2** is met by having the cloud's DNS server resolve the PoPSiCl to an IP address of an SDN controller in the datacenter hosting the corresponding tenant server. This reveals the datacenter (region) in which the tenant resides, but nothing else.

Requirement **R3** is met in a manner similar to that for **R1**, i.e., by the PoPSiCl store generating a new public key (and public-key certificate) for the tenant server per client who registers a PoPSiCl for that server. This certificate is then provided to the tenant server for use in TLS connection setups when accessed using the corresponding PoPSiCl.

Finally, requirement **R4** is met because the tenant server will accept a TLS connection to a PoPSiCl only from the client that registered it. Additionally, in the case of HTTP traffic, cross-origin attacks to indirectly query a PoPSiCl are prevented through refusing requests for URLs containing the PoPSiCl unless those URLs are appended with the PoPSiCl-specific capability that only the tenant server or `linker.popsicls.com` can obtain (Section 3.3.3).

Usability. Registration to obtain a PoPSiCl for a server is the only per-server procedure that a user must perform. In this step, the user visits a cloud-run website via HTTPS and enters the web server domain of interest. In return, it receives a PoPSiCl and a file containing a client public-key certificate and corresponding private key for use in TLS connections using this PoPSiCl. How the user employs this data is client-specific, but for a modern web browser, it might involve creating a bookmark using the PoPSiCl and importing the client certificate and private key into the browser. As such, we believe that our design meets requirement **R5**. It should also be noted that to register a PoPSiCl for a server, a user needs to learn that the server is hosted in the cloud. This point is discussed further in Section 3.7.

Changes to tenant servers are as follows. A tenant server OS must be modified to support the receipt of TCP connection states from the SDN controllers in the cloud (Section 3.3.2), and a tenant server also needs to be modified to refuse any connection using a PoPSiCl except from the client who registered it. A tenant web server must also check any URL using a PoPSiCl for the corresponding capability, as discussed in Section 3.3.3, to prevent cross-origin request forgeries that might deanonymize the PoPSiCl. Finally, the hyperlinks in the server's content must be changed to use relative URLs (for content at the same site) or the cloud-operated linker service (for content at another site), and the tenant server must set its referrer policy appropriately (Section 3.3.3). As discussed below, these changes can be introduced within VMs (or by a PaaS cloud operator) in such a way that content programming APIs need not be altered. We thus argue that requirement **R6** is also met.

3.4 Implementation

We realized our design in an OpenStack[8]-based IaaS cloud on top of the CloudLab[9] testbed. Each cloud computing node supports one or more tenant virtual machines (VMs) using the KVM hypervisor[10]. All tenant VMs are connected to the cloud network via Open vSwitch [125]. Open vSwitch is a software switch that runs in each hypervisor and bridges the virtual network interfaces of VMs on multiple computing nodes to a single layer-two network. Besides normal layer-two switching, Open vSwitch can be integrated with SDN controllers to support dynamic rule deployment and packet rewriting. Although our system is built upon OpenStack, KVM and Open vSwitch, we believe that our design could also integrate easily with other cloud implementations, such as Amazon EC2.

Below we detail our implementation of the three major components in our design: the PoPSiCl store, the SDN controller, and tenant web servers. In addition, a video demonstration of the user experience for our prototype can be found at `http://www.cs.unc.edu/~qiuyu/popsicl/`.

3.4.1 PoPSiCl Store

PoPSiCl store is implemented on one of the web servers that are controlled by the cloud operator. Its frontend is a regular HTTPS web server offering a web interface for browsers, which accepts registration requests from any client browser. The backend of PoPSiCl store is implemented as a native component (400 lines of C++ code) that interacts with the frontend using a FastCGI protocol. Upon receiving a registration request, the frontend passes the request to the backend to complete the registration process.

Generating PoPSiCl and pseudo-address. The PoPSiCl store backend generates a PoPSiCl for the client. The newly created PoPSiCl is prefixed by a pseudorandom string *str* that meets the

[8]`https://www.openstack.org/`

[9]`https://www.cloudlab.us/`

[10]`http://www.linux-kvm.org/`

domain-name format requirements [107], and so the PoPSiCl takes the form *str*`.popsicls.com` where `popsicls.com` is the domain name of the cloud. The PoPSiCl store also chooses a pseudo-address for the PoPSiCl uniformly at random from a block of addresses for cloud SDN controller(s). Uniqueness is not required for the pseudo-address; different PoPSiCls may be associated with the same pseudo-address.

Generating certificates. The PoPSiCl store generates an X.509 server certificate `SvrCert` for the tenant server for which the client is registering a PoPSiCl, and an X.509 client certificate `ClntCert` for the client. In `SvrCert`, the issuer is the cloud operator, `popsicls.com`; the subject name is the generated PoPSiCl; and the public key comes from a key pair (2048-bit RSA) that is newly generated by the PoPSiCl store using OpenSSL[11]. `SvrCert` is signed by the cloud operator's private key, and so a chain of trust can be established when the cloud's certificate is trusted. (To do so, the cloud operator must first obtain a CA certificate that authorizes its private key to sign new certificates.) The issuer of `ClntCert` is the PoPSiCl and the subject name is a unique string that is derived from the PoPSiCl. The PoPSiCl store generates another RSA key pair for the `ClntCert` and signs the `ClntCert` using the private key that was created for the tenant server, so that trust in the `SvrCert` can be extended to `ClntCert`. The PoPSiCl store bundles each certificate and its corresponding private key into a single PKCS#12 format file.

Distributing registration data. The PoPSiCl store distributes registration data to parties as follows: It sends the PoPSiCl and the pseudo-address to the cloud DNS server (which stores them as a DNS record); the PoPSiCl and the domain name of the tenant server to the SDN controller; `SvrCert` and the corresponding private key to the tenant server; and the PoPSiCl (or to support HTTP access, a URL containing the PoPSiCl and a capability for it, embedded as a query string; see Section 3.3.3), `ClntCert`, and its corresponding private key to the client through the frontend interface.

[11] https://www.openssl.org/

3.4.2 Cloud SDN Controller

In our implementation, all Open vSwitch instances are managed by the same SDN controller, a vSwitch controller we implemented in about 600 lines of C code. The SDN controller uses ovs-ofctl[12], a Linux command-line tool, to install and remove rules in each Open vSwitch.

As discussed in Section 3.3.2, every new TCP connection request using a PoPSiCl will be directed to the SDN controller, which completes the TCP handshake and then, after receiving a `ClientHello` message, hands off the TCP connection to the tenant server. To seamlessly transfer the TCP state from the SDN controller to the tenant server, our system uses a custom kernel extension (i.e., a kernel driver) to the Linux kernel (v4.2.0) to create a new user-kernel interface on both the SDN controller and each tenant VM. Userspace programs can exploit this interface (through `ioctl` system calls) to query or make changes to the internal TCP states. To facilitate TCP state migration, we also developed a userspace library that enables the SDN controller to obtain a copy of the TCP state information (sequence number, acknowledgment number, etc.) for a specific Linux socket descriptor. The state information is then sent through a long-lived TCP connection to the tenant server, which uses our helper library to create a new TCP socket with the specified TCP state and then resume the TCP session.

3.4.3 Tenant HTTP Server

Key to our tenant web-server implementation is the support of virtual hosts—or "server blocks" in Nginx, on which we base our implementation. A virtual host is a website implemented by a single web server; importantly, one web server can implement multiple virtual hosts. In our implementation, each virtual host corresponds to a PoPSiCl and thus a client of the website.

Upon PoPSiCl registration, the tenant web server receives the registration data for the client (i.e., the PoPSiCl, the server certificate, and the corresponding private key) from the PoPSiCl store. The configuration file of the Nginx web server is updated automatically to reflect these registration data: a server block is added to the configuration file, with its `server_name` direc-

[12]`http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt`

tive set to be the PoPSiCl and the `server_certificate` and `server_certificate_key` directives set as the file-system paths of the server certificate and private key, respectively. The `server_client_certificate` is set to be the path of the server certificate, as well, so that the virtual server to be created accepts only connections from clients who possess a certificate signed by the server's private key (see Section 3.4.1). Nginx supports server reconfiguration on-the-fly, and so a new virtual server for the PoPSiCl is created with the updated configuration file without any server down time.

We also adapted the Ruby on Rails web-content development framework (v2.2.2) to defend the cross-origin attack (see Section 3.3.3). Several macros, including `stylesheet_link_tag`, `javascript_include_tag`, and `link_to`, were modified to append the capability query string to same-origin URLs constructed via these macros. For each incoming HTTP request, the Ruby on Rails framework first checks for the capability query string. If the validation fails, a 404 error is returned.

3.5 Evaluation

In this section, we evaluate the impact of our design on performance of server interactions. More specifically, the goals of our evaluation are to demonstrate the impact of PoPSiCls on server-access latencies and throughputs, as well as the scalability of our design.

Our PoPSiCl-enabled OpenStack cloud was deployed in the CloudLab Wisconsin data center. For most experiments, we configured our cloud with three physical nodes: one for running OpenStack services (including DNS); one for running the PoPSiCl store and SDN controller; and another for running a tenant web server in a virtual machine. All nodes ran an Open vSwitch, though all rule installations described in Section 3.3.2 occurred on the controller machine. Each physical node was equipped with two Intel E5-2630 v3 8-core 2.40GHz CPUs, 128GiB of memory, and a Dual-port Intel X520-DA2 10Gb NIC. The web client was running on a desktop located in the UNC-Chapel Hill network. One experiment that compares our design with a proxy-based design has different settings, as will be discussed later.

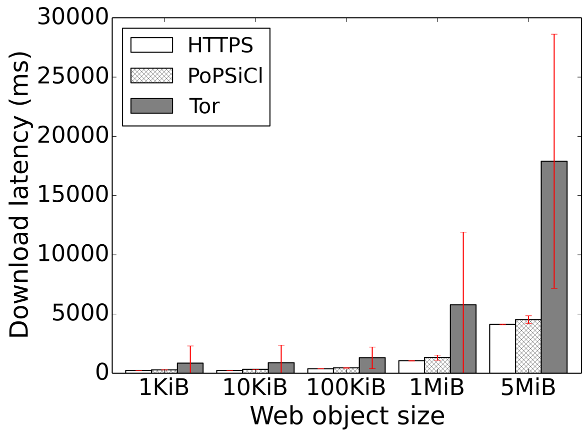
3.5.1 Performance

In this section, we discuss the performance of our PoPSiCl implementation. We primarily compare to the performance of HTTPS alone (with no PoPSiCl and no client authentication) and, in one experiment, the performance of Tor. We caution the reader that our comparison with Tor is only somewhat fair: While Tor also provides server anonymity (a socalled "hidden service"), it does so against stronger adversaries than our design does; e.g., our design reveals the cloud datacenter within which a tenant server resides (Tor would hide this information) and additionally provides client-server unlinkability [126], to an extent. Still, we compare to Tor because it is the most widely used anonymous communication system today.

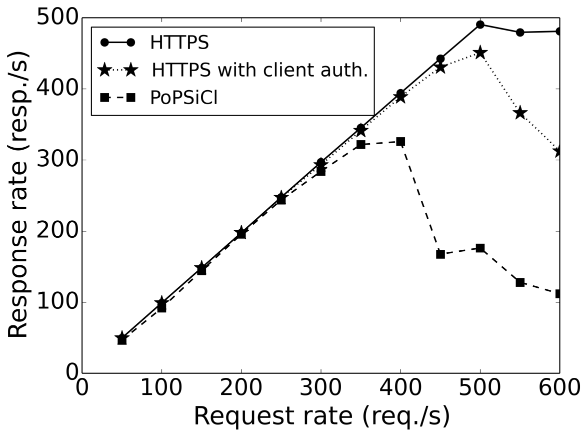
Web object download latency. In our first experiment, we measured the latency of downloading web objects. We chose Firefox as the web client for evaluating HTTPS and PoPSiCls, and the latest Tor browser (v5.5.5) for evaluating Tor. To fairly measure the extra overhead introduced by PoPSiCls, we restarted the web browser for each test, and so every web access was made through a new TCP and TLS connection, including the overheads of TLS client certificate authentication and TCP session hand-off from the SDN controller to the tenant server. We also restarted the browser between accesses when testing HTTPS. For the Tor browser, we measured the access latency after the Tor circuit had been built. We repeated the experiment 50 times per web-object size, which varied from $1\mathrm{KiB}$ to $5\mathrm{MiB}$.

Average access latencies per object size are shown in Figure 3.3a. As is clear from Figure 3.3a, access using PoPSiCls is minimally more expensive than using HTTPS with no client authentication, and is roughly 2.5–$4\times$ more efficient than accessing content using Tor. Moreover, the standard deviation of download latency for Tor is very considerable, indicating that for some Tor downloads, latencies were even worse than $4\times$ more expensive.

The latency of web-object retrieval using PoPSiCls is robust as the request rate grows. For example, the average latency for retrieving a $10\mathrm{KiB}$ object increases to $404\mathrm{ms}$ when it is requested at a rate of 200 requests per second from distinct clients (not shown), an increase of less than 20% over the corresponding latency in Figure 3.3a. Latency is also relatively unaf-



(a) Latency



(b) Throughput

Figure 3.3: Performance of PoPSiCl access

fected by cross-site hyperlinking (see Section 3.3.3), especially for larger objects: accessing `linker.popsicls.com` involves another HTTPS connection but none of the mechanism in Section 3.3.2, and is unaffected by the retrieved objects' size. So, for example, the latency of retrieving a $1\mathrm{MiB}$ object via `linker.popsicls.com` (not shown) is less than 11% larger than when retrieving it using a PoPSiCl directly.

Web access throughput. In this experiment, we measured the throughput of the web server. We used httperf[13], a popular web server benchmark tool, to measure the throughput. httperf can be used to dictate the rate of TCP requests and the number of HTTP requests per TCP connection, and it then will report the corresponding HTTP response rate. In our experiment, we measured and compared the server throughput when the server provides web access through HTTPS, HTTPS with client authentication, or a PoPSiCl. We scaled the request rate from 50 requests/s to 600 requests/s, with one HTTP request/response pair per TCP connection. The size of the requested web object was $1\mathrm{KiB}$. For each request rate and each condition, we took 10 samples of the response rate and calculated their mean.

As can be seen from Figure 3.3b, before the web server reached its sustainable throughput, its response rate kept pace with the request rate. After the web server reached its limit, the re-

[13]`http://www.labs.hpe.com/research/linux/httperf/`

sponse rate dropped as the request rate increased further. The maximum throughputs of HTTPS, HTTPS with client authentication, and PoPSiCl were 490.5, 450.9, and 325.8 responses/s, respectively. Compared with HTTPS, PoPSiCl induced a 33.5% throughput decrease. The throughput bottleneck in these tests was the switch rule installation procedure (see Section 3.3.2), which increasingly encountered failures when the request rate grew.

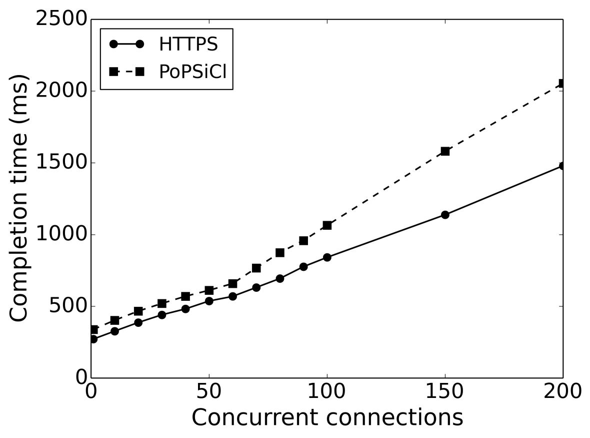
3.5.2 Scalability

Our design poses several potential scalability pitfalls. In this section we evaluate these elements of our design.

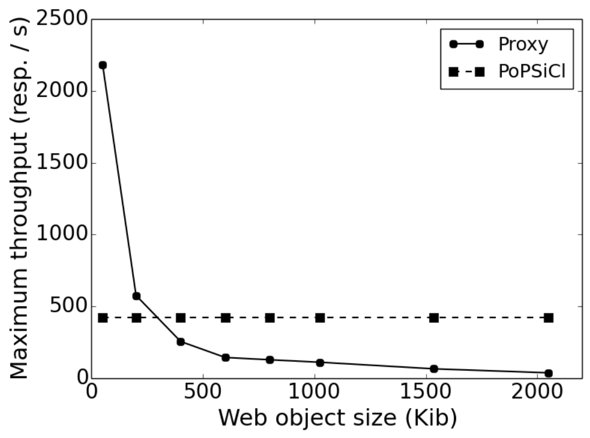
SDN rule installation and TCP handoff. As discussed in Section 3.3.2, our design results in the installation of two rules per PoPSiCl-based TCP connection through a switch (or one rule into each of two switches), followed by the transfer of TCP state from the SDN controller to the tenant server and then the adjustment of one of the rules previously installed for this connection. These steps slow the connection setup to a tenant server, and so in our first experiment we evaluated the impact of these overheads, as the number of concurrent connection setups grows.

We used cURL (`https://curl.haxx.se/`) as the web client for both HTTPS and PoPSiCl access. In each test, we launched concurrent cURL processes; each process opened a connection (HTTPS in one type of test, or using a PoPSiCl in the other type), retrieved a web object from the tenant server, and then terminated its connection. We measured the completion time of *all* connections, and plotted this completion time as a function of the number of processes (and connections) launched. The size of the web object in this experiment was 10KiB.

Figure 3.4a shows the result of these experiments, where each point is the average of the results from ten runs. As can be seen there, the completion time for the PoPSiCl-based connections was at most $1.4\times$ the completion time for the same number of concurrent HTTPS connections. Our SDN controller and tenant web server implementations already perform the steps for each connection concurrently, though otherwise the implementations are relatively unoptimized.



(a) Concurrent connections



(b) Per-connection volume

Figure 3.4: Scalability of PoPSiCl access along several dimensions

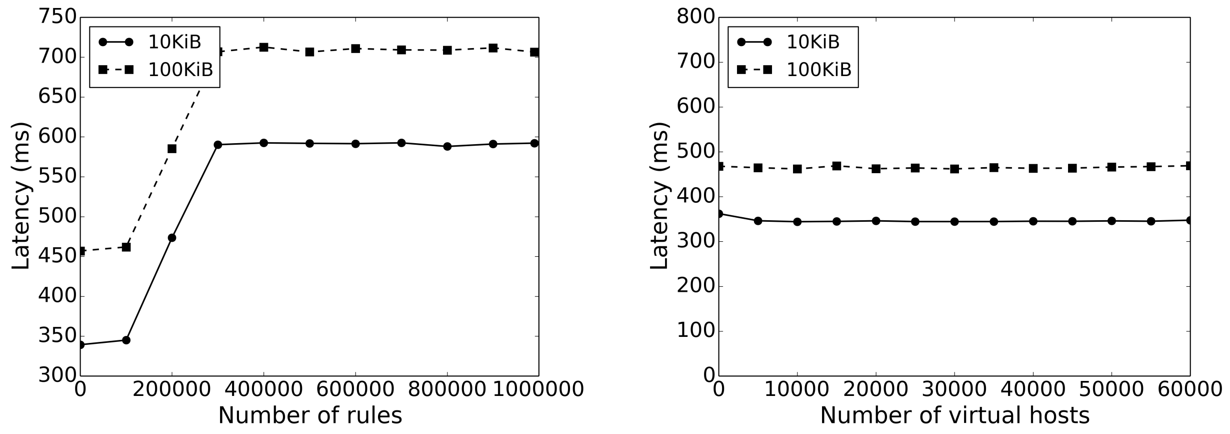
The need for TCP handoff. The motivation for SDN rule installation and TCP handoff steps detailed in Section 3.3.2 and evaluated above becomes evident when comparing our design to a proxy-based alternative. In this alternative, each PoPSiCl is resolved to a *pseudo-IP* that is the address of a proxy that completes the TCP connection with the client and then learns which tenant server the client wants to contact from the `ClientHello` SNI field (like our SDN controller does). The proxy then opens another TCP connection with the tenant server and relays traffic between the client and server, without handing off that connection to the server.

We built this proxy alternative and evaluated its throughput. The proxy was implemented in C with the UNIX socket API as a multithreaded application. When a new TCP connection is initiated by the client, the proxy spawns a new thread, accepts this connection, and establishes a TCP connection with the tenant server to relay packets. To ensure that the clients and servers were not bottlenecks, we set up three httperf clients connecting to three tenant web servers, each on its own physical node. As can be seen in Figure 3.4b, the proxy alternative outperformed ours when the retrieved web object was small. But as the web-object size increased, the maximum throughput of the proxy decayed dramatically. In contrast, the maximum throughput of PoPSiCl stayed the same as the web-object size increased. The throughput of PoPSiCl is more stable because each packet only makes the SDN switch rewrite the IP address field. However, each packet makes the proxy copy packet content from one socket to another socket and each TCP connection costs extra resources to maintain TCP states. Aurelius et al. [10] found that 70% of Flash video flows from various services transferred at least $1\mathrm{MiB}$ data. If such streaming services were deployed in the cloud, a proxy could be easily saturated.

SDN switch rules. By default, one million rules can be installed simultaneously in a vSwitch, which would thus accommodate up to a half million concurrent client TCP connections in support of PoPSiCl-based server accesses. This introduces two scalability concerns.

First, since some clients (notably browsers) tend to open multiple connections per server access, a vSwitch's default rule capacity might accommodate far fewer than a half million concurrent clients using PoPSiCls. Statistics from a commercial multi-tenant data center shows that 50% of the hypervisors had mean flow counts of 107 or less [125]. Our use cases certainly require more rules. However, this concern can be addressed by increasing the rule capacity of a vSwitch and also load-balancing rule installation across the potentially multiple vSwitches that a client's connections traverse. Indeed, additional vSwitches could be added in the cloud—even elastically—to boost rule capacity, if needed.

Second, deploying several hundred thousand rules to a vSwitch could slow the process by which the vSwitch matches incoming packets to rules, and so we conducted experiments to evaluate this performance degradation. Figure 3.5a shows the average latency (over 200 trials) suffered by a client using a PoPSiCl to open a connection and retrieve an object of either $10\mathrm{KiB}$ or $100\mathrm{KiB}$ from a tenant web server, when the switch starts with the number of rules indicated on the horizontal axis. The performance clearly shows two "levels" of latency per object size. The latency jumps to another level after the number of rules exceeds 200000. The reason is that Open vSwitch uses a kernel cache to store the switching rules and the cache size is 200000 [125]. Nevertheless, the performance impact with nearly 1000000 rules is less than $2\times$. It is conceivable that engineering a switch specifically to accommodate the usage that our design imposes might further reduce this degradation.



(a) Open vSwitch rules (b) Virtual hosts

Figure 3.5: Scalability of PoPSiCl access along several dimensions

Virtual hosts. As discussed in Section 3.4.3, our implementation deploys a virtual host to a tenant web server per client that registers a PoPSiCl for it. This virtual host is associated with the PoPSiCl and the server certificate that the web server should use in TLS connections using that PoPSiCl (and that doubles as the certificate for verifying the client certificate). Admittedly this is perhaps an abuse of the virtual-host mechanism, which presumably was not designed to accommodate a virtual host per web-server client—a popular web server could have millions of virtual hosts.

To get a sense for the scalability limitations that the existing Nginx virtual-host design would impose, Figure 3.5b shows the degradation in responsiveness of the tenant web server as a function of the number of virtual hosts installed. These tests were conducted by first creating the number of virtual hosts on the horizontal axis and then connecting to the server using a PoPSiCl that matches one of these virtual hosts, to retrieve an object of either $10\mathrm{KiB}$ or $100\mathrm{KiB}$.

Figure 3.5b shows the performance impact of the number of virtual hosts set up before the connection. Each point is an average over 200 trials. The number of virtual hosts had no impact on the response latency for the numbers we tested. However, the memory consumption of the server with 60000 virtual hosts approached $2\mathrm{GiB}$. As in the case of vSwitch rules above, we anticipate that this scalability limitation could be addressed with a virtual-host design that anticipates our proposed usage, e.g., relieving this memory pressure by writing the data for inactive virtual hosts to stable storage.

DNS entries. Each PoPSiCl registration results in the creation of a new DNS record for the PoPSiCl, mapping the PoPSiCl to the addresses of SDN controllers in the datacenter where the tenant resides. The number of DNS records could thus grow large, if the use of PoPSiCls became popular. We have not evaluated the potential performance impact of this growth on DNS resolutions, however, since backing the DNS server with a simple database for these records would support ample storage and fast access. Going further, the *str* portion of a PoPSiCl *str*`.popsicls.com` could be computed to be the encryption (using a chosen-ciphertext-secure scheme) of the IP address(es) to which it should be mapped, using a key that the DNS server holds. The DNS server would then not need to store the mapping, but upon receiving a request to resolve *str*`.popsicls.com` could instead decrypt *str* and return the result.

3.6 Traffic Analysis

As discussed in Section 3.2.1, PoPSiCls hide the identity of the servers contacted by clients from being directly disclosed to an attacker between the clients and the cloud. However, they do not hide connection characteristics from the attacker. The connection characteristics include packet size, packet timing, the number of packets per connection, connection volume, the number of connections, connection duration, etc. What can be inferred from these features has long been studied and debated, particularly in the context of traffic directed through anonymizing proxies (e.g., [93, 65, 158, 69, 122, 86]) and, similarly, traffic logs in which payloads have been removed (e.g., [48]).

Our design so far has left this issue to tenant servers to address, should they choose to. However, in this section we summarize an approach that we have developed for HTTP servers that can be used to implement some proposed defenses to address attacks leveraging packet size to identify servers. We leave defenses for attacks leveraging other features to future work. In keeping with **R6**, this defense can be deployed with very modest adaptations to web content. Also, the proposed approach relies on client-side Javascript and does not require the client user to install new software (in keeping with **R5**). We stress that our goal here is not to innovate in terms of new per-connection defenses, but instead to provide a framework in which such defenses can be implemented.

3.6.1 Design

The key enabler for these defenses without requiring modifications to the client platform is Javascript blobs[14], which provide a way for client-side Javascript to construct file-like objects and pass them to APIs that expect URLs. This functionality permits a tenant server to serve Javascript to the client browser that customizes how objects are retrieved from the server, and then post-processes the retrieved contents and provides them to the browser (as blobs) for rendering. So, for example, the client-side Javascript could replace the retrieval of one web object with that of many smaller objects and then reassemble the original object before providing it to the browser.

We have prototyped this approach in a simple adaptation to the Ruby on Rails web-content development framework. This adaptation prepends a Javascript script to every HTML file; the script takes control of retrieving the objects that otherwise would have been hyperlinked directly in that page. This script patches the `XMLHTTPRequest` class to remove padding and reassemble the original object from smaller pieces of that object. When the `send` method of a `XMLHTTPRequest` instance is called, instead of sending a single HTTP request to the server, it sends several HTTP requests. The URL of each request contains the original URL and also an extra query string that informs the server of the number of pieces into which to split the object at that URL and the index of the piece to return in response. After receiving the first such request, the modified Ruby on Rails framework splits the web object into the requested number of pieces; the piece at the index indicated in each request is then returned as the request's response, after appending padding and prepending the length of that padding to the piece (to allow for padding removal). After the `XMLHTTPRequest` instance gets all the pieces of the original object, it reads

[14]`http://developer.mozilla.org/en-US/docs/Web/API/Blob`

a length field at the beginning of each piece, strips the piece of any content (i.e., padding) that extends past that length indicator, reassembles the original object, creates a blob containing the resulting object, and submits it to the browser for rendering. Rewriting the `XMLHTTPRequest` class ensures that even objects retrieved by other Javascript scripts in the page will be split and reassembled in this way.

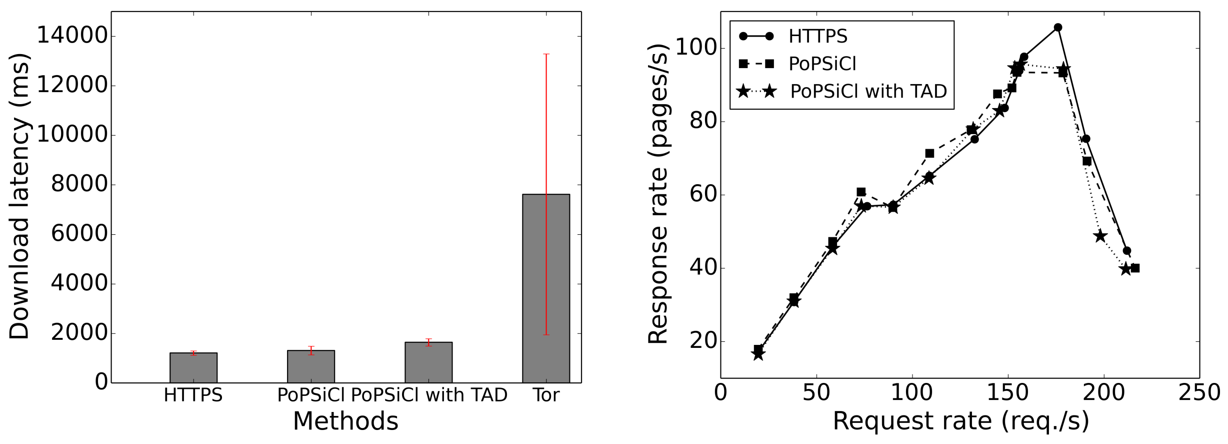
While providing a foothold for addressing traffic analysis based on the features of individual object retrievals, this design does not interfere with hints that might be available to the attacker based on his viewing multiple connections in aggregate, such as the number of servers that are accessed simultaneously. Developing a similar foothold for obfuscating these features is a topic of ongoing work.

3.6.2 Evaluation

To evaluate the performance impact of this form of traffic-analysis defense, we modified an open-source blog site[15], which is written in Ruby on Rails, to adopt it. In terms of modifications to the application-specific web content itself, we needed to modify only one line of embedded Ruby code in two templates; the rest of the implementation is embedded in the Ruby on Rails framework, hidden from the web-content developer. In our implementation, the Javascript code chooses uniformly from among retrieving a web object in one, two, or three pieces, and each piece is padded to make its size a multiple of 512 bytes. The root page of this website hyperlinks to thirteen web objects of total size 474.3KiB. We evaluated our traffic analysis defense approach, in terms of latency and throughput, by visiting this root page repeatedly, being sure to clear the browser cache between retrievals.

Latency. We chose the Firefox browser for evaluating the latency of root-page retrievals via HTTPS, PoPSiCls, and PoPSiCls with traffic-analysis defense (TAD) implemented as above, and the Tor browser (v5.5.5). The latency was measured as the time of downloading and rendering the whole web page. The experiment was repeated 20 times for each setting. As can be seen from

[15]`https://github.com/natew/obtvse`



(a) Latency (b) Throughput

Figure 3.6: Performance of PoPSiCl with traffic-analysis defense (TAD), for retrieving a web page hyperlinking to thirteen objects of total size 474.3KiB

Figure 3.6a, the latency of accessing the page using PoPSiCls with traffic-analysis defense is only 1.35× that of HTTPS, while the latency of Tor is 6.3× that of HTTPS.

Throughput. To measure throughput of root-page retrievals, we chose a headless browser, PhantomJS[16], as the web client. Though lacking a graphical user interface, PhantomJS still parses HTML documents, runs Javascript code, and downloads hyperlinked web objects in a web page. (The httperf tool, used in the throughput experiments of Section 3.5, does not parse returned HTML.) We wrote a script to spawn PhantomJS processes in the background, and each PhantomJS instance was scripted to visit the root page once and then terminate. By adjusting the spawning rate, we adjusted the web-page request rate, and the response rate was measured as the number of root-page retrievals completed per second. To ensure that the client was not the bottleneck in these experiments, we ran the client in a physical node with 32 cores and 128GiB memory. As can be seen in Figure 3.6b, the throughput of root-page retrievals using PoPSiCls was approximately the same with or without traffic-analysis defense, and only slightly lower than the throughput using HTTPS alone.

[16]http://phantomjs.org

3.7 Discussion

Scope of defense. Our design provides PoPSiCls only for servers hosted in a cloud that supports their use. While major cloud operators host substantial numbers of web servers (e.g., [75]), for example, obviously numerous web servers do not fall into this category, as well. Related to this limitation is that a user must know or be directed to the cloud that hosts a server in order to register a PoPSiCl for it. This information could be disseminated by the cloud, provided that the cloud is trusted to not claim to host a web server that it does not (as a major cloud operator might be); by a link to the appropriate PoPSiCl store from the web server itself, so that a user could leverage one access to the server to be able to access it using a PoPSiCl subsequently; or by myriad other means (e.g., social media).

Censorship. We assume a trustworthy cloud operator that is motivated to help tenants protect the privacy of their customers from an attacker who might try to observe their customers connecting to them. This assumption is arguably stronger than most (though not all, c.f., [160]) threat models considered in works addressing censorship resistance. Indeed, in the context of censorship by governments, clouds are often *used* by activists to circumvent censors, but this is typically done without the cloud operator's consent [58]. Major clouds have admittedly shown little cooperation for resisting censorship by governments (e.g., [71]), preferring instead to accommodate censorship for business reasons. Moreover, the PoPSiCl store is vulnerable to being blocked. As such, our design seems unlikely to be deployed specifically to resist government censorship, but it still offers an opportunity for a cloud to actively contribute to privacy for customers of tenant servers to which censors permit access (even while disallowing access to servers that censors forbid).

Going further, it is conceivable that our design offers an attractive balance between social responsibility and client privacy by assuming a trusted cloud that retains the ability to censor servers. For example, evidence suggests that a majority of Tor "hidden services" are criminally oriented and the most frequently requested sites host child abuse imagery [121]. Such abusive sites could be shut down by the operator once it is informed of the abusive content.

Traffic analysis. As discussed previously, our basic design (Section 3.3) leaves traffic analysis to the tenant server to address, should it choose to. While we have made initial steps to support per-connection traffic-analysis defense (Section 3.6), that defense does not immediately address traffic analysis based on aggregates of connections, e.g., the number of servers to which connections are made or the relative timings of these connections. Defending against this type of attack remains a very active area of research independent of our proposal (e.g., [65, 158, 122, 66]).

OS compatibility. Our current implementation of TCP hand-off (Section 3.4) requires that both the controller and the tenant server run on the same Linux OS kernel version. We also disabled the TCP timestamp and selective acknowledgment (SACK) options to facilitate TCP state migration. In future work, we aim to support TCP hand-off across different TCP stack implementations so that PoPSiCls will be suitable for more heterogeneous deployments.

Compatibility with Universal 2nd Factor (U2F) protocol. Universal 2nd Factor (U2F) protocol [89] aims to improve web security and usability by specifying a standard for using a second-factor device in web authentication. U2F protocol is supported by major browsers, including Chrome, and popular websites. In U2F, a second-factor device, usually a USB-based security key, generates a public-private key pair for each website in the registration phase. The key pair is associated with the domain name of the registered website and stored in the second-factor device. The public key is associated with the user account and stored in the web server. In the authentication phase, only if the user can generate a valid signature with the expected private key, will she be successfully authenticated by the web server. Also, the second-factor device ensures that the signature can be only used for the associated website to prevent phishing attack. PoPSiCl is used as a normal domain name during the whole web session so it is compatible with the U2F protocol. In order to use U2F, the user needs to register her second-factor device with the PoPSiCl-enabled website every time she registers a new PoPSiCl.

3.8 Summary

In this chapter we presented PoPSiCls, which are personalized pseudonyms for servers that a client can use like regular, long-lived server domain names to open TLS connections to those servers. We described a design and implementation for PoPSiCls that leverages trust in a cloud to implement PoPSiCls for tenant servers that it hosts. PoPSiCls have several desirable security and usability properties in our threat model. First, TLS connections established using a PoPSiCl exhibit identifiers (domain names, IP addresses, and server public keys) to the attacker that he cannot correlate against those exhibited in connections involving other clients or other tenant servers. Second, the burdens placed on various parties in our implementation of PoPSiCls correspond to their levels of technical capabilities: cloud operators bear the most (which, based on our experience, is still minor); changes to tenant web servers can be hidden from web-content developers by packaging these changes within VMs (in an IaaS scenario) or the cloud platform (in a PaaS scenario); and tenants' clients need only suffer minor changes to the user experience and no changes to client software (in the case of web browsers). The last is important since client users often lack the permissions or willingness to modify their platforms (e.g., due to the vulnerabilities that those modifications can introduce [129, 46, 139]). We also illustrated extensions of our design to permit the implementation of defenses against traffic analysis with no client-side changes.

CHAPTER 4: SNOWMAN: METERING GRAPHICAL DATA LEAKAGE TO DETECT SENSITIVE DATA EXFILTRATION

Data theft by insiders is a threat that is especially difficult to prevent, as it involves misuse of permissions that the insider presumably must be given to perform his/her duties in the organization. It is thus not surprising that such data thefts are so common; e.g., in healthcare, insider threat is the most common cause of data leakage, accounting for 58% of incidents [145]. And, of course, insiders were behind some of the highest profile data breaches of U.S. government data, with the Manning [157] and Snowden [149] cases being two exemplars. All U.S. executive agencies and military departments are now required "to monitor user activity on all classified networks in order to detect activity indicative of insider threat behavior" [119, §H.1], and a National Insider Threat Task Force has been established "to develop a Government-wide insider threat program for deterring, detecting, and mitigating insider threats" [2].

An approach to address data theft by insiders is to make sensitive data available to users only by secure remote access. In this approach, programs execute on sensitive data only on computers trusted by the organization, while users are permitted to interact with those programs/data only remotely, perhaps from a less-trusted (even user-owned) computer or a "thin" client having no persistent storage of its own. While this approach has been practiced for decades in various forms (e.g., [167]), a current product embodying this approach is Citrix Virtual Apps and Desktops [1], formerly marketed as XenApp and XenDesktop. The deployment of XenDesktop by Osaka Gas [3] provides an illustrative example of the data security benefits that this approach can offer.

Despite the data-protection benefits of remote-only access, this approach is fundamentally limited by the possibility that the user screenshots sensitive content or even photographs it using another device. Distribution of such images can be discouraged through the introduction of watermarks (e.g., by varying screen luminescence [72]). However, the ability to attribute the data leak to an individual after the data is already leaked might be ineffective in deterring the leak. Moreover, since these watermarks must not interfere with the user experience, text data can be recovered, sans watermarks, by applying optical character recognition to the images [39]. For small text data files (e.g., a Word file that fits on one or two screens), it seems that there is little hope for defending further against such insiders.

The premise of this work, however, is that for *large* amounts of sensitive text—e.g., large documents, databases, or codebases—an effort to quickly display significant amounts of that data to a computer screen to record it (e.g., using another device) is likely to induce patterns of accessing that data that departs from the norm for interacting with it for legitimate purposes. Combined with remote-only access, this observation might be leveraged to detect data theft *as it is occurring*. In this chapter, we propose a system, called Snowman, to accomplish this goal. Roughly speaking, Snowman monitors the transmission of sensitive data by a program to a remote client via a graphical user interface, and raises an alert when the rate of transmission exceeds what is typical for interacting with that data.

A central challenge in this approach is how to measure the amount of sensitive data transmitted to a remote client. While the total volume of GUI data transmitted to the remote client is an upper bound, such a bound is very coarse. For example, numerous ways of interacting with a program—e.g., scrolling a document a few lines, up and down repeatedly—would result in an ever growing estimate of leakage, even though only the same few lines of data are being rendered to the user's screen. Snowman therefore employs taint analysis (e.g., [115]) to track which sensitive bytes taint each byte output to the remote user. By tracking the cumulative set of sensitive bytes that taint the output to the remote user, Snowman can improve the accuracy of this upper bound considerably, and even determine *which* sensitive bytes might have been leaked.

Unfortunately, multi-label (i.e., per sensitive byte) taint analysis on unmodified binaries is very expensive, incurring a typical overhead of $7\times$ or more [88]. To ensure that this overhead does not interfere with the user experience, Snowman thus performs taint analysis only on a *replica* of the program that replays the program's execution alongside the program instance that interacts with the user. This replica lags behind the user-facing instance due to the overhead of taint analysis, but as we will show, it needs not lag by much for typical user behavior. Herein lies a core technical challenge that we solve in Snowman, namely how to efficiently conduct taint analysis on a replica while forcing the replica to execute *identically* to the original, user-facing execution.

Replication systems (e.g., [120, 20]) need to record asynchronous signals and scheduling events, and to replay them to the replica at the exact same execution points as in the original execution. Those systems rely on CPU hardware performance counters to measure the number of instructions executed by the program to locate the right execution points at which to replay them. However, conventional taint analysis tools [88, 26] use binary rewriting techniques to insert analysis routines into the original code blocks, which would break the measurement of the replica's execution and so cause the replica to diverge from the original. Snowman employs a novel architecture that conducts taint analysis in the kernel without disturbing the performance counter measurements. To reduce the overhead of taint analysis, Snowman employs various optimizations, such as excluding application basic blocks from in-kernel analysis where it can be inferred that they have no tainted operands, caching instruction decoding results, copy-on-write taint propagation, and garbage collection of taint tags corresponding to already-leaked data.

We have implemented Snowman to work on unmodified x86-64 Linux binaries and off-the-shelf hardware. We evaluated Snowman on three widely used GUI programs: LibreOffice Writer (a word processor), LibreOffice Calc (a spreadsheet program), and Gedit (a code editor). Our evaluation shows that Snowman performs better than the Pin “null tool”, which serves as a baseline for any Pin-based [97] taint analysis solution, and introduces only moderate overhead on common user actions. The evaluation also shows that Snowman can easily distinguish normal user behaviors from ones reflecting data copying in all tested programs, by analyzing the sensitive data leakage patterns.

To summarize, our contributions are as follows:

- To our knowledge, we provide the first system designed to detect copying of graphical output to reconstruct sensitive data, e.g., to exfiltrate it later. Rather than focusing on watermarking to assign responsibility for the theft after the released data is recovered, Snowman instead seeks to detect the copying while it is occurring.
- We detail the design of Snowman, which performs multi-label taint-tracking only on a replica of the user-facing execution, to minimize the performance impact to the interactive user experience. In doing so, Snowman simultaneously achieves exact replication of the user-facing execution while performing multi-label taint tracking on it, without modification to the program binary. Central to its efficiency are a variety of optimizations that render it far more lightweight than straightforward solutions (e.g., based on Pin [97]).
- We show through evaluations of Snowman on a fully-featured word processor, spreadsheet program, and code editor, that sufficiently aggressive copying can easily be differentiated from normal usage examples based on the rate of GUI leakage of sensitive file output. We also show that Snowman supports copying detection with minimal penalties to the responsiveness observed by the user, and with modest delays from the time at which the leakage occurs.

The rest of this chapter is organized as follows. We give background in Section 4.1 and describe the design and implementation of Snowman in Section 4.2. In Section 4.3, we evaluate Snowman in terms of its performance on various user actions and its capability of differentiating malicious data-access patterns from normal ones. We discuss remaining challenges and possible extensions in Section 4.4, and summarize this chapter in Section 4.5.

4.1 Background

Anomaly detection. Here we treat a malicious insider exfiltrating a large volume of sensitive data from an organization as an anomalous behavior to be detected using anomaly-detection techniques, of which many have been proposed (see, e.g., [34]). In anomaly-detection systems, various events of the user and the running programs are logged. The logs might include, e.g., system calls, shell commands, file reads and writes, and others. These logged data are then provided to the feature-based detection algorithms, which can be based on machine learning [117], data mining [159], statistics [90], or information theory [91], to identify anomalies.

The main contribution of Snowman is offering a novel system architecture that (i) restricts user's interaction with the sensitive files to the GUI interface in a thin client and (ii) accurately monitors the sensitive bytes leaked to the user. Snowman can generate logs containing the indices of the leaked bytes and the timestamps of the leakage events, which can be used as features by the anomaly-detection algorithms. This fine-grained sensitive data leakage pattern gives more insight into the user's intent compared with coarser patterns of access to files or file blocks. In this sense, Snowman provides a new type of feature for anomaly-detection systems to analyze, though our goal here is not to develop new anomaly detection algorithms ourselves.

Taint analysis. The conventional approach to monitor sensitive information flow is taint analysis (e.g., [115, 45]). Taint analysis systems attach taint tags to the memory and register locations whenever the program consumes sensitive data. Along with the execution of the program, the taint tags are propagated from one location to another. Since the taint propagation rules are dictated by the instruction semantics, taint analysis can accurately monitor how the sensitive data is transformed and transferred, and whether it is leaked through the program's execution.

Taint analysis can be implemented with dynamic binary instrumentation (e.g., [128, 88, 26, 106, 105, 83]) or virtualization (e.g., [77, 127, 41]). In inlined taint analysis systems [128, 88, 26, 77, 127, 41], the taint analysis logic directly interferes with the analyzed program's execution flow and thus introduces substantial overhead. For example, libdft, a state-of-art taint analysis system, imposes $7.06\times$ slowdown to the Firefox browser even after employing various optimization techniques [88]. Some recent systems [106, 105, 83] aim to reduce overhead of the analyzed program by decoupling taint analysis from the program's execution. These systems record the control flow and memory access information with Pin, a dynamic binary instrumentation tool, and run the taint analysis logic on a separate thread with the recorded information. Snowman also decouples taint analysis by replicating the program's execution and conducting taint analysis only on the replica. Since Snowman doesn't do heavyweight binary instrumentation, it adds less overhead to the analyzed program compared with the Pin "null tool" (as will be shown in Section 4.3) and hence all other Pin-based tools.

Some systems choose to implement taint analysis in hardware [50, 147, 155, 87] and usually have better performance than the software based systems. However, custom hardware is not widely deployed. Snowman has better applicability since it works on off-the-shelf hardware and unmodified binaries.

Replicated execution. Replicating the execution of a multithreaded program in multicore systems is a challenging problem. Many sources of nondeterminism can lead to divergence of the replicated execution. The first type of nondeterminism is caused by the program's communication with the system or other programs via systems calls, e.g., `read()`, or instructions, e.g., `RDTSC`. Replication systems (e.g., [120, 20]) usually address this type of nondeterminism by recording the nondeterministic inputs to the original execution and replaying the recorded values to the replica. The second type of nondeterminism is caused by shared-memory interactions among threads or processes. To address this type of nondeterminism, the approaches taken by replication systems include replicating all shared-memory accesses [124, 21]; scheduling one thread or process at a time and replicating the scheduling decisions [112, 144, 120]; assuming the program is race-free and replicating the synchronization events by instrumenting the synchronization library [15, 14, 53]; or applying a deterministic scheduling algorithm to remove the nondeterminism in shared-memory interactions [20, 19, 54].

Replicating the program's execution can be also achieved by running the program in a virtual machine and replicating the execution of an entire virtual machine [59, 164, 60, 29]. However, doing so introduces unnecessary overhead of replicating the execution of the operating system and other programs. There are also replication systems [113, 79, 163] relying on custom hardware to reduce overhead. We adapted RR [120], an open-source tool from Mozilla, to implement Snowman's replication subsystem. We chose RR because it works on unmodified binaries and off-the-shelf hardware. Additionally, RR doesn't assume race-freedom of the replicated program so that programs with data races (e.g., programs using lock-free data structures[153]) can be directly replicated by RR.

Replication and dynamic analysis. Several previous works [124, 53, 57, 84] explored the idea of combining replication and dynamic analysis. Unlike Snowman, which replicates and analyzes the execution of the program concurrently with the original execution, these systems can replicate the previously recorded execution of the program only after the program exits, to conduct analysis for debugging, auditing, or attack provenance. In addition, the high overhead caused by replicating every shared-memory interaction [124] or replicating a virtual machine [57], as well as assuming race-freedom [53, 84], do not fit our use cases.

4.2 System Design and Implementation

4.2.1 Overview

In Snowman, GUI programs with permissions to access sensitive files run in a remote server. A user employs her personal computer to interact with the GUI programs over the network. The GUI programs take user inputs, such as mouse clicks and keyboard strokes, do computations, and deliver graphical outputs to the user computer. In our threat model, the user computer is not trusted and the user might intend to steal sensitive information. However, we assume the server computer and all software running on the server are trusted and don't have security vulnerabilities. The only channel where a user can get sensitive information is the graphical outputs generated by the remote GUI programs. Our system aims to

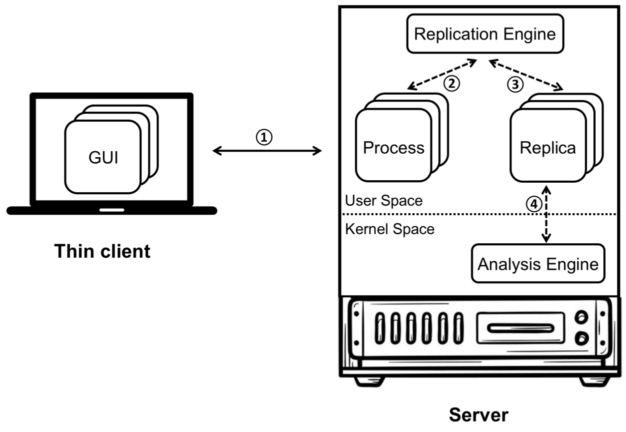


Figure 4.1: Snowman architecture: (1) communicate user inputs and graphical outputs (over the network); (2) record program execution; (3) replicate program execution; (4) monitor sensitive data leakage

accurately monitor the amount of sensitive file information flowing from the remote server to the user computer without affecting the normal usage of the GUI programs.

To accomplish this, Snowman replicates the execution of the analyzed program and conducts taint analysis on the replica, to minimize the impact of that analysis on the performance of the original execution. Our system works on legacy x86-64 binaries without requiring custom hardware or recompilation. Figure 4.1 shows the overall architecture of our system. The graphical user interfaces of the program are displayed in the user computer (a thin client). In the remote server, the replication engine creates and maintains a replica for every thread and process of the monitored program. The replicas maintain the exact same execution states as the original threads and processes, including memory values, register values, and control flows. The analysis engine conducts taint analysis on the replicated processes by attaching and propagating taint tags to monitor the sensitive information flow. There are many challenges in efficiently performing taint analysis and, at the same time, faithfully replicating the original program's execution. We will give detailed descriptions of our system in the following sections.

4.2.2 Replication Engine

The implementation of the replication engine is based on RR [120], which is designed to record and replay multi-threaded Linux programs with low overhead, and is useful in debugging concurrency bugs. Instead of replaying the whole execution after the program exits, we adapt RR to run a replicated program side-by-side along with the original program execution throughout its lifetime.

The core problem solved by RR is faithfully replicating the execution of a multi-threaded program. In principle, if all non-deterministic inputs and events of the original execution are recorded and replayed to the replica, the replicated execution should be the exact same as the original one. RR runs in user-space and monitors the target program via the `ptrace` system call. RR can observe various events of the monitored program including system calls and signals. Whenever a monitored process enters or exits a system call, it is suspended and RR is notified.

For system calls that spawn a new process or thread, like `fork()` and `exev()`, RR creates a corresponding replicated process or thread and copies the original memory and register state to the replica. For the system calls that consume non-deterministic inputs, such as `read()` and `gettimeofday()`, RR records the inputs to the system call from the original process and replays them to the replicated process without actually executing the system call. RR also deals with inputs from non-deterministic instructions, including `RDTSC` and `RDRAND`, by emulating or rewriting those instructions.

Besides non-deterministic inputs to system calls and from non-deterministic instructions, RR also needs to record and replay the non-deterministic events. The first type of non-deterministic events is scheduling events. Multiple threads of the same process run concurrently and do computations on shared data, and so different thread schedules could lead to different outcomes of the program. Without replaying the scheduling events, the replica's execution could diverge. Another type of non-deterministic events is signals. Signals usually interrupt the normal execution flow of the program. If a signal handler is registered, it will be called to handle the arrived signal. Since the signal handler could compute on data shared with normal program code, similar to scheduling events, we have to record and replay the signals to avoid divergence.

RR acts as a scheduler to the monitored program and only schedules one thread at a time. By using a deterministic performance counter of the Intel CPU, RR tracks the number of retired conditional branches (RCB) and uses the RCB counts to mark the progress of the program's execution. Whenever a non-deterministic event happens in the original program, RR records the timing, measured by the RCB count, of that event, and replays this event in the exact same execution point of the replica. RR instructs the CPU to fire an interrupt after a specified number of conditional branches are retired by the replica to control the timing of the event replay. If it is a scheduling event, RR preempts the replicated threads to replay the schedule. If it is a signal, RR emulates the execution of the signal handler without delivering a real signal to the replica.

During the original execution of the program, RR records the aforementioned data. Consumption of the recorded data is sometimes slower than its generation because conducting taint analysis on the replicated program could slow down its execution. The replication engine buffers the recorded data in the file system, so that the original program execution can advance normally without having to wait for the replica.

4.2.3 Analysis Engine

If the replication engine maintains a replicated program execution that progresses exactly as the original one, conducting taint analysis on the replica should expose the same sensitive information flows as occurred in the original. The analysis engine's goal is thus to track which sensitive file bytes taint GUI outputs of the replica execution (and so of the original execution, to the client computer) without causing the replica's execution to diverge from the original. libdft [88] and other similar tools (e.g., [45, 26]) use dynamic binary instrumentation to transform the original code blocks to semantically equivalent ones intertwined with the taint analysis logic. We don't take such an approach to implement the analysis engine since adding additional instructions in the replica would confuse the RCB counts measured by the replication engine, which might lead to divergence of the replicated execution due to the non-deterministic events being inserted in the wrong execution point. As such, the analysis engine in Snowman takes a different approach, which we summarize in this section.

4.2.3.1 Architecture

Snowman defines each memory or register byte as an individual taint unit, to which it associates taint tags dynamically. The instructions executed by the program dictate how the taint tags should be propagated. For example, if the program executes "`mov ebx, eax`", which moves four bytes of data from the `eax` register to the `ebx` register, the analysis engine should move the taint tags on each byte of the `eax` register to the corresponding byte of the `ebx` register. Strictly speaking, the analysis engine is required to perform this type of analysis on every instruction executed by the program.

In Snowman, the analysis engine is implemented as a Linux kernel module, and all taint analysis operations are done in kernel space, which don't interfere with the RCB counts of the user-space replica. For the replica, the analysis engine maintains shadow memory address spaces and shadow registers that store taint tags for the corresponding memory and register bytes. Different threads of the same process share the same shadow memory address space but have their own shadow registers. A strawman approach to implement taint analysis is running the replica in single-step mode. After the execution of each instruction, the CPU traps to kernel mode and transfers control to the analysis engine. The analysis engine then decodes the binary instruction and decides how to propagate taint tags based on the instruction's opcode (e.g., `mov`) and operands (either memory or register operands). This approach works but is too slow since each instruction triggers a CPU context switch from user mode to kernel mode.

One source of optimization is the observation that we need to analyze an instruction only if its operands have taint tags. Leveraging this observation, the analysis engine checks, at the beginning of each basic block, whether the register operands of the instructions in the basic block contain taint tags by inspecting the shadow registers. If any register operand has taint tags, the analysis engine sets the CPU to single-step mode for this basic block. If not, the analysis engine sets a breakpoint at the last instruction of the basic block, which lets the analysis engine seize control and check the next basic block. This design often induces one context switch per basic block instead of per instruction, which could be a big performance gain if only a small percentage of instructions touch tainted registers.

Figuring out whether the memory operands of a basic block contain taint tags is a more complicated task. Since some memory operands can be indirectly addressed—e.g., in "`mov eax, [ebx]`", where the address of the memory operand is the value of the `ebx` register—we may not know the address of the memory operand at the beginning of the basic block. As such, we cannot decide whether the basic block should run in single-step mode by only inspecting the shadow memory. To solve this problem, the analysis engine changes each memory page that contains tainted memory to kernel-only pages by modifying its page table protection bit. Whenever an instruction in the program accesses those protected pages, the CPU generates a page fault and transfers control to the analysis engine. The analysis engine then changes the accessed page to user-accessible and sets the CPU to single-step mode. After that, every instruction in that basic block will be analyzed by the analysis engine.

As shown in Figure 4.2, each replicated thread is classified as being in one of four states by the analysis engine. When the replica process has not consumed any sensitive data and there are no taint tags in the shadow memory or shadow registers, all threads of the replicated process are in *Initial* state, and the analysis engine does not set breakpoints at basic-block boundaries. As soon as the replica process consumes sensitive data, all threads transition to *Undecided*. When a thread is in the *Undecided* state, the analysis engine checks whether the register operands in the current basic block contain any taint tags. If so, the thread is transitioned to the *Single-step* state and every instruction in the basic block will be analyzed individually. After the thread finishes the last instruction of the basic block, it is transitioned back to the *Undecided* state. If the register operands are not tainted, the thread transitions to the *Block-step* state, and the analysis engine sets a breakpoint at the last instruction of the basic block and changes all tainted pages to be kernel-only pages. If the thread accesses any tainted page, it is transitioned to *Single-step*. If the thread finishes the basic block without accessing any tainted pages, then it is transitioned to *Undecided*. This state machine ensures that every possible tainted data flow will be captured and analyzed by the analysis engine.

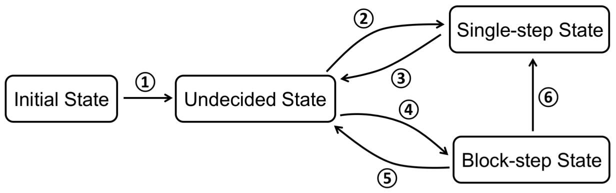


Figure 4.2: State transitions: (1) consume sensitive data; (2) contain taint tags; (3) finish basic block; (4) no taint tag; (5) finish basic block; (6) protection page fault

4.2.3.2 Taint Analysis

To monitor the amount of sensitive data leaked out of the GUI program replica through its graphical outputs, the analysis engine assigns a different taint tag to each byte of the sensitive data consumed by the replica. Each shadow memory or register byte can be attached with a list of different taint tags. When the graphical output data is sent out of the system, the analysis engine inspects what taint tags are contained in the output bytes to track which sensitive bytes have been leaked out.

During taint analysis, the analysis engine applies different analysis rules on different instructions based on their semantics. The instructions involving direct taint propagation can be divided into four categories: movement instructions, arithmetic instructions, logical instructions, and transformation instructions.

Movement instructions. Movement instructions move data among memory and registers, or assign immediate values to memory or registers. By this definition, `mov`, `pop`, and `push` are movement instructions. Bit shift instructions, like `shr` and `shl`, are also categorized as movement instructions since they can be considered as moving data within a register or memory location. For the movement instructions, the analysis engine first locates the source and destination operands, and then replaces the taint tags in the destination shadow memory or register, byte by byte, with the ones from the source. If the source operand doesn't contain any taint tags or is an immediate value, then the analysis engine clears the taint tags in the destination.

Arithmetic instructions. Arithmetic instructions do arithmetic operations on the source operands and save the result to the destination operand. `add`, `sub`, `mul`, `div`, and `inc` are common arithmetic instructions. For unary instructions like `inc`, we don't need to change the taint state of their operand. For binary instructions, the analysis engine first accumulates all the taint tags from every byte of the source operands, and then assigns this list of taint tags to every byte of the destination operand in the shadow memory or shadow register. There is a special case for `sub`, or `sbb`, instruction. If the source operands of `sub` are the same, the analysis engine clears the taint tags in the destination operand. For example in "`sub eax, eax`", the taint tags in `eax` are removed.

Logical instructions. Logical instructions do bitwise logical operations on memory or register operands. Different from the arithmetic instructions, in a logical instruction, each byte of the destination operand is affected only by the corresponding bytes of the source operands, and so the analysis engine assigns only the taint tags from those source-operand bytes to the corresponding byte in the destination operand. Special cases here are `and` and `xor`. If one of the source operands in `and` is the immediate value `0`, then the analysis engine clears the taint tags in the destination operand. If the source operands of `xor` are the same, the taint tags in the destination operand are also cleared.

Transformation instructions. Transformation instructions change the data representation of an operand from one type to another. For example, `cvtsi2sd` changes the data from an integer representation to a scalar double-precision floating-point representation and moves the data from the memory or a general register to a SIMD (single instruction, multiple data) register. Semantically, every byte of the source operand affects every byte of the destination operand, so analysis engine accumulates all the taint tags from every byte of the source operand, and then assigns this list of taint tags to every byte of the destination operand.

Figure 4.3 gives examples of taint propagation for different types of instructions.

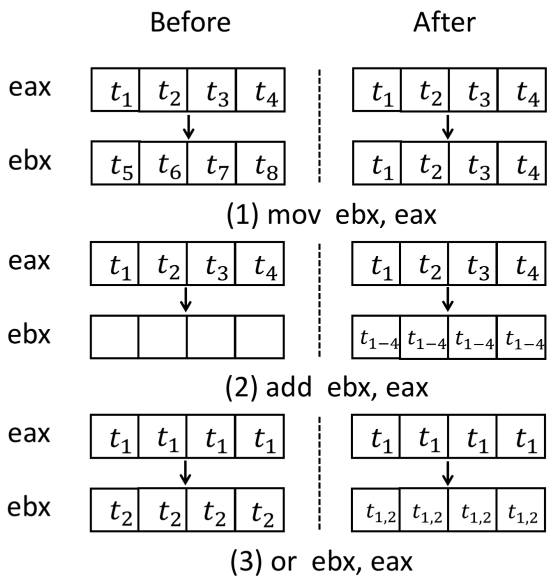


Figure 4.3: Examples of taint propagation

4.2.3.3 Code Caches

The analysis engine decodes each instruction to figure out 1) its opcode; 2) the names of any register operands; 3) the base register, index register, scale factor, and displacement of any memory operands; and 4) the value of any immediate operands. For each basic block, the analysis engine needs to know which registers are included in the basic block to decide whether they are tainted, to make decisions about state transitions (as described in Section 4.2.3). Decoding instructions is expensive, taking around 1000 CPU cycles per instruction. To leverage the space and time locality of code execution, similar to the CPU instruction cache, we use two in-memory software caches to speed up taint analysis.

Instruction cache. The instruction cache stores the decoded information—opcode, operands, etc.—of instructions. Whenever an instruction is executed, the analysis engine checks whether the instruction is in the cache with an index calculated by the instruction's virtual memory address. If it is not in the cache, the analysis engine decodes the instruction and caches the results. Otherwise, the cached information is directly used without decoding the instruction.

Block cache. We also use a block cache to store the names of register operands in basic blocks. At the beginning of each basic block, the analysis engine checks whether the register names are cached, with an index calculated by the starting address of the basic block. If it is a cache miss, the analysis engine has to decode each instruction in the basic block to get the register names. This decoding step can be accelerated by the instruction cache.

4.2.4 Implementation

We implemented the replication engine with 61 lines of C++ code on top of Mozilla RR v5.2.0. We implemented the analysis engine with 4454 lines of C code as a kernel module in Linux v4.11.12. We integrated the Zydis[1] disassembler into the analysis engine to decode x86-64 instructions. We will discuss some implementation details in this section.

Analyzed instructions and functions. We implemented taint analysis rules for 28 movement instructions, 14 arithmetic instructions, 12 logical instructions, and 5 transformation instructions. Those are commonly used instructions including a set of SIMD (single instruction, multiple data) instructions. We do not propagate taint tags to the `eflags` register and we ignore implicit data flows caused by control-flow dependencies. Many previous works (e.g., [115, 88]) also ignore the implicit data flow to avoid over-tainting.

[1]https://zydis.re/

Some library functions are hooked and analyzed by our system. The analysis engine hooks `open()`, `close()`, and `read()` in the glibc library to add taint tags when the program opens and reads sensitive files. The analysis engine also hooks `XRenderCompositeText()` in the xrender library to inspect whether the rendered text contains taint tags.

Shadow memory and registers. Each memory and register byte (both for general registers and SIMD registers) has a corresponding "shadow byte" maintained by the analysis engine. The "shadow byte" is actually a pointer to the head of a singly linked list of taint tags. A taint tag is a 32-bit integer, and so taint tags can track up to 4GB of sensitive data. We use a linear array to store the register "shadow bytes" for each thread. The data structure for the memory "shadow bytes" is a combination of a hash table and a linear array. Specifically, "shadow bytes" of a memory page are stored in a 4096-entry array. The location of this page array is saved in a hash table using the page address as the hash key. This hybrid design strikes a good balance between using a pure hash table and a pure linear array, where the former might trigger too many hash collisions while the latter consumes too much memory.

Taint tag allocation is a time-consuming operation and costs kernel memory. We implemented a *copy-on-write* taint propagation scheme to avoid unnecessary tag allocation. For the movement instructions, we only copy the pointer of the taint list from the source "shadow byte" to the destination "shadow byte" and increase the reference count of that pointer by one. For the arithmetic and logical instructions, where the source taint tags will be merged into the destination taint tags, we make a new copy of the taint list from the destination "shadow byte" if that list is also referenced elsewhere and then merge the source taint tags.

We also implemented a *garbage collection* scheme to free memory used to track already-leaked sensitive bytes. In our system, if a sensitive byte is leaked, we increase the leakage count by one. Future leakages of that same byte don't leak any more information and we don't have to keep tracking it. Therefore, we maintain a list of already leaked taint tags and periodically invoke garbage collection to remove those tags from the shadow memory and shadow registers.

Cache settings. As described in Section 4.2.3.3, we implemented a block cache and an instruction cache. The block cache is a direct-mapped cache that has only one element in each cache entry. If a new basic block is mapped to the same entry as an old one, the old cache entry will be replaced. Since a program usually executes a relatively small number of basic blocks, this direct-mapped cache worked well in practice. Cache collisions are more frequent in the instruction cache, and so we implemented it as a two-way set associative cache (two elements in one entry). If a cache collision happens, the least recently used cache block will be replaced. To ensure the correctness of the cached data, we don't cache instructions and basic blocks if they are in a writable and executable page.

Control transfers. The analysis engine needs to take control from the replica at the right times to do taint analysis (see Figure 4.2). To make the replica run in single-step mode, we set the TF bit in `eflags` register. We use the x86 debug register to set breakpoints at basic block boundaries. We add hooks in the debug trap handler (`do_debug()`) and the page fault handler (`do_page_fault()`) to transfer control to the analysis engine. Since the replication engine also sets single-step mode for some of its replay operations, we maintain an internal state to indicate to the analysis engine to transfer control to the replication engine instead of directly to the replica.

4.3 Evaluation

In this section, we focus on evaluating Snowman's performance by measuring the reaction time of various GUI programs to user actions. We also evaluate Snowman's capability of differentiating the data-leakage patterns of malicious insiders from those of normal users.

GUI programs. We selected three typical and widely used GUI programs—a word processor, a spreadsheet, and a code editor—for our evaluations. The code editor is Gedit[2] (v3.18.3), which is pre-installed in many Linux distributions and has common features like syntax highlighting

[2]https://wiki.gnome.org/Apps/Gedit

and word completion. The word processor and spreadsheet program are from LibreOffice[3] (v5.4), which is an open-sourced office suite (comparable to Microsoft office) and has a large user base. In particular, LibreOffice is a fairly complicated multi-threaded program that has 9 million lines of code (including C++, Java, and Python components)[4]. We believe testing Snowman with LibreOffice would make a comprehensive validation of our design and implementation.

Environment. In the remote-access scenarios, the GUI programs ran in a Linux server that installed Snowman with the customized v4.11.12 kernel. We interacted with the GUI programs in a client machine that installed Ubuntu 16.04 with the v4.15.0 kernel. The client took mouse and keyboard inputs and sent the inputs to the server. The server did computation, generated graphical outputs, and sent the outputs to the client. The inputs and outputs were exchanged through the X11 protocol[5], which is the basic component of the Linux GUI framework, via TCP connections. The client machine was equipped with a 2-core $3\mathrm{GHz}$ CPU and $4\mathrm{GiB}$ of memory. The server machine was equipped with a 4-core $3.5\mathrm{GHz}$ CPU and $8\mathrm{GiB}$ of memory. The client and server were connected by $1\mathrm{Gbps}$ Ethernet links in a local area network.

4.3.1 Performance for Benign Users

A core indicator of the GUI program's performance is its reaction time to user actions. To accurately measure the reaction time, we used Wireshark[6] to monitor the X11 packets passed through the TCP socket in the client machine. The reaction time was calculated as the difference between the departure time of the first user input packet and the arrival time of the last graphical output packet triggered by the user action.

We measured the reaction time of various actions performed on LibreOffice Writer (the word processor), LibreOffice Calc (the spreadsheet), and Gedit (the code editor). Each measurement was repeated 10 times. The actions we tested are the following.

[3] https://www.libreoffice.org/

[4] https://www.openhub.net/p/libreoffice/analyses/latest/languages_summary

[5] https://www.x.org/

[6] https://www.wireshark.org/

Start the program. Each program was started by entering a command in the terminal. The reaction time was measured as the duration between the command key press and the first window of the program displayed on the screen.

Open a file. We opened a 46KiB text file in Writer, a 25KiB spreadsheet in Calc, and a 88KiB source code file in Gedit. The reaction time was measured as the duration between the open button click and the full text rendered on the screen.

Close the program. We exited each program by closing its first window. The reaction time was measured as the duration between the close button click and all graphical resources released by the program.

Scroll down. For each program, we clicked the scroll bar once to scroll down the window by one page. The reaction time was measured as the duration between the scroll bar click and the new text rendered on the screen.

Search a string. We searched a string in each program. The reaction time was measured as the duration between the search button click and the string being located on the screen.

Paste text. We pasted a 3984-character sentence in Writer, a 47-by-14 spreadsheet table in Calc, and a 13-line source code snippet in Gedit. The reaction time was measured as the duration between the paste button click and the pasted text rendered on the screen.

These actions were tested in five settings. In the first setting, the GUI program ran locally in the client machine, which is the normal setting without data protection from remote-only access. To assist reaction-time measurement, the X11 packets were transmitted through local TCP sockets. In the other four settings, the GUI program ran on the server machine, and the user interacted with the program through the client machine. Among these four settings, the first one ran the program natively without instrumentation; the second one ran the program under the protection of Snowman; the third one ran the program under the "null tool" of Pin [97] (v3.6); and the last one ran the program under the "taint tool" of Pin (v3.6) with the reading bytes from the opened file marked as tainted. The Pin "null tool" does the minimal amount of instrumentation to maintain

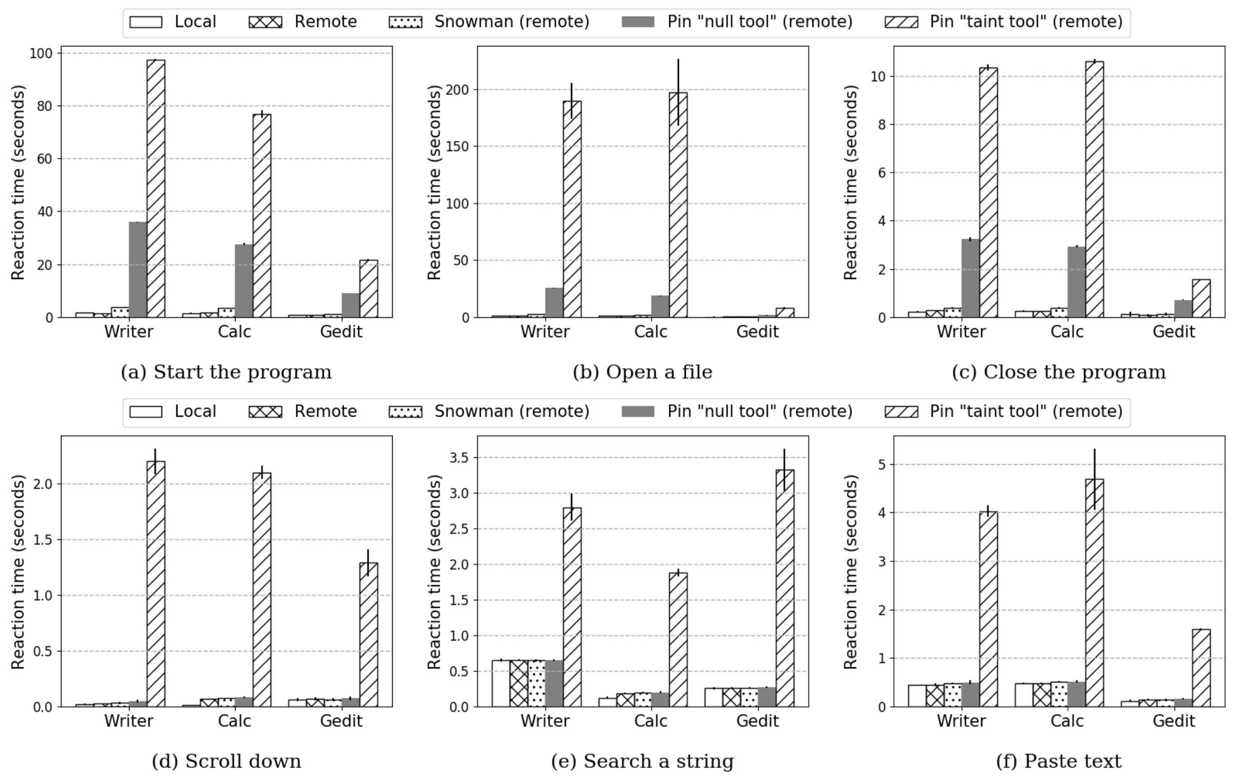


Figure 4.4: Reaction time for common user actions when running the programs locally 1) in the client machine, remotely 2) in the server machine without instrumentation, 3) monitored by Snowman, 4) by the Pin "null tool", or 5) by the Pin "taint tool"

supervised execution of the program. The Pin "taint tool" conducts multi-label taint tracking and employs the same set of taint analysis rules as Snowman. It was implemented by us and was used to debug and validate the implementation of Snowman's taint analysis engine.

Figure 4.4 shows the average reaction time of various actions in different settings, where the error bar represents the standard deviation. Overall, the reaction time of the actions in Snowman was 0.92× to 2.41× the reaction time of those actions in the remote-only setting without any instrumentation. Among these actions, opening a file, starting and closing the program have relatively large overhead (1.16× to 2.41×) because Snowman needs to record the non-deterministic inputs and take extra steps to set up and tear down the environment for recording. However, we don't expect this would have significant impact on user experience since these actions are not frequently triggered by the user in typical workloads. The reaction time of other actions in Snowman are comparable to those in the remote-only setting (0.92× to 1.19×).

Additionally, Snowman performs better than the Pin “null tool” and “taint tool” in all tests. Compared with the remote-only setting without any instrumentation, the reaction time overhead is 1.05× to 23.07× in the Pin “null tool” and 4.3× to 133.1× in the Pin “taint tool”. The actions triggering taint propagation (opening a file, scrolling down, searching a string, and pasting text) have huge overhead in the Pin “taint tool”. We don’t claim this multi-label taint tracking tool implemented by us has the best possible implementation. But we expect other similar tools would have similarly considerable overhead because the taint analysis routines are inlined with the normal program code by the Pin instrumentation. Besides, considering that the “null tool” doesn’t implement any instrumentation for taint analysis, which represents a lower bound for taint analysis approaches implemented with Pin, Snowman should perform better than any Pin-based taint analysis tools.

4.3.2 Data Exfiltration Detection

Snowman aims to detect data exfiltration by monitoring the amount of sensitive data leaked to the user. Here we describe our evaluation of its efficacy in this regard, using the same applications as used for the performance evaluation for benign users in Section 4.3.1. That said, we caution the reader that a holistic evaluation of Snowman as an anomaly detector is not possible without a broad corpus of normal usage profiles for these programs. Given the variety of tasks for which these types of applications are used and the varying levels of expertise of the users who leverage these programs, it is difficult to envision how such a holistic evaluation could be performed, or even if one anomaly detection algorithm would be suitable for all cases. Indeed, we envision that individual deployments might leverage custom detectors, based on the types of activities typically conducted by users, or even by individual users or on individual files.

Given these complexities, here we settle for a more primitive demonstration of the detection capabilities of Snowman: we simulated a “typical” normal user session and a malicious user session for each GUI program, and then showed that the leakage profiles of the two sessions as observed by Snowman could be statistically differentiated with overwhelming ease (and the speed with which this differentiation could occur, etc.). We designed the “normal” sessions based primarily on their representation in publicly available resources (see below), so that readers can easily assess the nature of activities in each, should they so choose. Moreover, these sessions were performed without undue delay or extra “thinking time,” so as to simulate a more rapid leakage of data—and so, presumably, yielding a reasonably conservative evaluation. We discuss the settings and results of this evaluation in this section.

4.3.2.1 Settings

In our evaluations, the GUI program ran in the remote server and we interacted with the program in the client machine. Snowman maintained a replica of the program and conducted taint analysis on the replica to measure leakage.

LibreOffice Writer sessions. In the normal session, we formatted an ebook document by following the instructions from the Kindle ebook formatting guide[7]. The document we used was the first three chapters of the Python tutorial[8] with all formatting removed. We started the session by opening the document. Following the guide, we restored the format of the original tutorial by changing fonts of the section titles, inserting hyperlinks, adding footnotes and page numbers, and creating a table of contents. In the malicious session, we opened the same document, quickly scrolled down the document, and physically took pictures of all the pages with our phone.

LibreOffice Calc sessions. In the normal session, we did calculations on a spreadsheet containing employment and salary information. The spreadsheet was created with an online template[9] and filled with synthetic data (100 rows and 36 columns). Throughout the session, we calculated the number of employees taking more than two days off, the average medical expenses, and the total basic salary by using the built-in functions from Calc. In the malicious session, we quickly scanned the whole spreadsheet and took pictures of all the rows and columns.

[7]https://kdp.amazon.com/en_US/help/topic/G200645680 (This guide is based on Microsoft Word but we can find the same functionalities in LibreOffice Writer.)

[8]https://docs.python.org/3/tutorial/

[9]https://exceldatapro.com/download-salary-sheet-template/

Gedit sessions. In the normal session, we edited a C file to finish an assignment from the MIT Operating System Engineering class. We implemented the `env_init()` and `env_setup_vm()` functions by editing the `env.c` file, as required by exercise 2 of lab 3.[10] We also opened the `env.h` file to reference the related data structures. To best enable repeatability, we simply followed the solution from a github repository.[11] In the malicious session, we opened the `env.c` and `init.c` files, and took pictures of all the file contents.

4.3.2.2 Leakage Detection

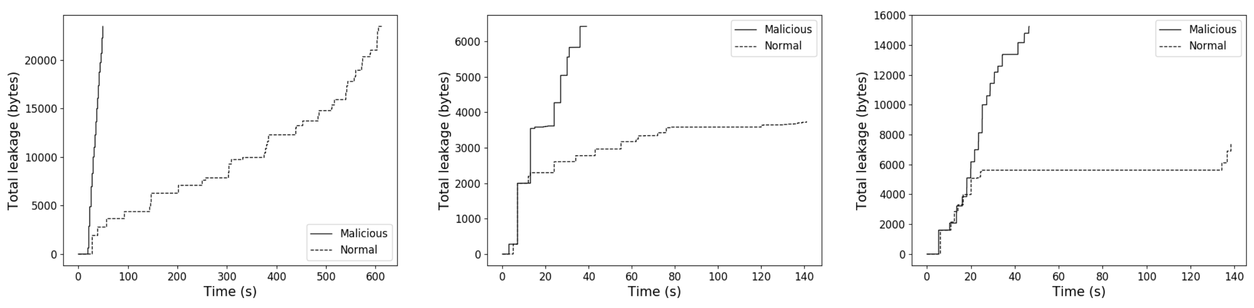
Snowman records various events of the monitored GUI program as described in Section 4.2.2. Besides the event data, Snowman also records the timestamp of each event. So, when Snowman detects a new leakage via its taint-tracking in the replica, it can report the timestamp of that leakage event from the original execution.

Figure 4.5 reports the total leakage from the sensitive files over time, as detected by Snowman, in the normal and malicious session of each program. The x-axis is the time of the original user session. The y-axis is the total leakage. As can be seen there, in all three malicious sessions, the sensitive bytes were leaked out within one minute. In the normal sessions, the leakage occurred at a slower speed. Additionally, the normal sessions of the Calc and Gedit programs leaked only a part of the file contents.

We applied a statistical analysis to test whether the malicious session and the normal session can be easily differentiated. The analysis we used is the logrank test [22], which is usually applied to compare the survival experience of two groups of patients. We treat the new leakage of a sensitive byte as analogous to the death event of a patient in this analysis. With the collected data, we can calculate the time intervals between leakages and so the frequencies of data leakages. We subjected the time intervals from the malicious session and the normal session of each program to the logrank test. The p-value calculated from the Writer, Calc, and Gedit sessions is 0,

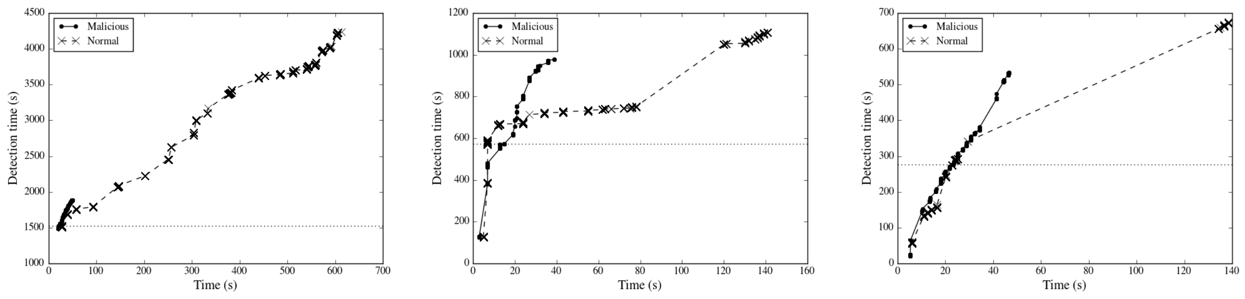
[10]https://pdos.csail.mit.edu/6.828/2018/labs/lab3/

[11]https://github.com/Babtsov/jos/tree/lab3



(a) Writer (b) Calc (c) Gedit

Figure 4.5: The amount of leakage in normal and malicious sessions, as a function of time



(a) Writer (b) Calc (c) Gedit

Figure 4.6: The time at which the Snowman replica detected each new leakage, as a function of the time (in the user-facing execution) that the leakage originally occurred

9.925×10^{-262}, and 9.157×10^{-168}, respectively. Thus, we can safely reject the null hypothesis that the leakage experience of the malicious session and the normal session are the same for all three programs.

4.3.2.3 Detection Delay

Although the taint analysis conducted by Snowman doesn't affect the program execution with which the user interacts, it slows down the execution of the replica and adds delay to the detection of the leakage event.

Figure 4.6 plots the time at which the replica detected each new leakage, as a function of the time that the leakage actually occurred. In the normal sessions, the Snowman replica detected the last leakage event with a lag of 6.9×, 7.8×, and 4.9× for Writer, Calc, and Gedit, respectively, behind when that leakage event occurred. In the malicious sessions, detection of the last leakage event lagged by 38.2×, 25.1×, and 11.5× for Writer, Calc, and Gedit, respectively. Gedit has the smallest lag, presumably because it has simpler code logic and takes less CPU cycles to process user requests. For all three programs, the malicious sessions lag more than normal ones, since the malicious user sent requests to the program at a higher frequency, leaving the replica fewer idle cycles to catch up.

We also want to figure out how quickly Snowman can differentiate the malicious session from the normal one. We adopted the following procedure to answer that question. In the timeline of the replica's execution, the leakage events from the malicious session, as detected by Snowman, were added to one dataset, and the leakage events from the normal session were added to another. These two datasets were updated at the end of each second *of the replica's execution time*. We started to apply the logrank test (the same as described in Section 4.3.2.2) on the updated datasets after both datasets had more than 1000 leakage events. We stopped the procedure when the calculated p-value dropped below 0.05. The time at which we stopped is reported as the earliest time Snowman could detect the malicious session (using the normal session as a "typical" baseline). The earliest detection times were 1522s, 574s, and 276s for Writer, Calc, and Gedit, respectively (shown as the dotted horizontal line in Figure 4.6).

4.4 Discussion

Taint analysis accuracy. Currently, we have implemented taint analysis rules for only a subset of x86-64 instructions in Snowman. This subset includes every instruction that explicitly propagates taint tags during the execution of the three tested programs (LibreOffice Writer, LibreOffice Calc, and Gedit). To capture this subset of instructions, we added an assertion in the taint analysis engine, which gives an alert if an executed instruction contains tainted operands but no taint analysis rule was implemented for that instruction. For other programs, we can take the same approach to select instructions for taint analysis.

Taint propagations caused by implicit data flows are ignored by Snowman. Implicit data flows can exist in branch instructions where the branch condition variable decides which value will be saved to a given variable. There are also implicit data flows in pointer dereferences. In some cases, the data value in a memory location is decided by the value of the memory address, such as table lookups. Ignoring these implicit data flows can cause false negatives in the leakage detection. However, enabling taint analysis for implicit data flows leads to false positives and taint explosion [143], and for this reason has been excluded in many prior tools (e.g., [88, 115, 128]).

To reduce false negatives caused by table lookups, we choose to manually identify functions which might propagate taint tags through table lookups and implement taint analysis rules based on function semantics. Specifically, we hook and analyze the `pango_shape_full()` function from the `libpango` library, which translates font indexes to unicode characters through table lookups. Other functions can be analyzed in the same way as soon as they are identified. We also plan to explore some existing techniques [73, 13] to address implicit data flows in the future.

Replication delay. The replica has to trap into the kernel to be analyzed by the taint analysis engine. This trap happens at least once per basic block and can increase to once per instruction if the basic block contains tainted operands. The frequent context switches between the user and kernel modes add significant overhead. We chose to implement the taint analysis engine in the kernel to avoid adding extra instructions into the replica. The benefit of doing so is that the replication engine can then determine when to inject the asynchronous signals and scheduling events by using the CPU retired conditional branch counter to measure the progress of the replica. In theory it should be possible to implement taint analysis in user space by instrumenting the binary and adjusting the measurements of the replica's execution accordingly. We plan to explore this option in the future.

Each CPU cycle in the original execution is amplified in the replicated execution due to taint analysis overhead. The experiments in Section 4.3.2.3 show that the replica cannot catch up with the original execution during the simulated session. However, the simulated user sessions don't represent real use cases since we omitted some idle time during the simulation. The idle time is often caused by application waiting for user input when the user spends time on thinking or working on other applications. With enough idle time, the replica could catch up with the original execution because the replica can fully utilize a CPU core. To better understand the implications of replica delay, real usage patterns of the applications are needed. We plan to investigate this in future work. It is also possible to run the replica in a machine with higher CPU clock rate than the one where the original execution runs, so the replica can better keep up with the original execution. On the other hand, the malicious user might intentionally trigger expensive operations to increase the delay by which replica processing lags behind the original, in order to "buy time" for copying data. Some malicious actions might also increase replication time due to excessive computation. To detect such attacks, we could build a model of normal lag, i.e., a profile based on the lag during normal user sessions for each program. If the lag during a user session deviates substantially from that profile, Snowman can raise an alert of potential malicious behavior, or alternatively slow down the user-facing execution.

Quick leakage estimation. Snowman's accurate leakage tracking could be augmented with a much quicker method of estimating the leakage based on examining the GUI traffic between the user-facing execution and the thin client. Ideally this estimator would quickly approximate the leakage with good recall and reasonable precision, to be corroborated (or corrected) by the replica when its analysis is complete. In doing so, Snowman could be made even more responsive to data exfiltration attempts. However, to tolerate false alarms by the estimator, a response to an estimator-based alarm might be to only slow down the user-facing execution, for example, until the taint-tracking replica catches up. We plan to investigate such an estimator in the future.

From differentiation to anomaly detection. We showed in Section 4.3.2 that it was trivial to statistically distinguish our own sessions of normal activity from ones in which we simply paged through files and photographed them, based on the leakage patterns determined by Snowman. While these tests provide strong evidence that detecting theft (as long as it is sufficiently aggressive) on the basis of a leakage profile is possible, the dearth of datasets characterizing the leakage patterns of either normal or theft-oriented usage of the applications tested there renders it impossible to properly evaluate an anomaly detection methodology based on Snowman. Moreover, since a data-leakage profile during normal use might be highly dependent on the expertise of the user and the type of file being accessed, there may not be a one-size-fits-all detector; rather, leakage models created per user, per file, or at least per organization might be more appropriate. Of course, there will be limits to what a purely volume-based approach can detect; i.e., a detector based solely on the *amount* of data leaked so far will presumably fail to detect data exfiltration performed very slowly, at a speed similar to normal usage. For such cases, analyzing the exact *order* in which bytes are leaked—which Snowman also provides—might be necessary. Besides, the presence of Snowman could at least slow down the data leakage rate in malicious situations. Still, we believe that even our primitive studies already provide a strong basis to motivate the further study of GUI leakage measurement as enabled by Snowman.

CPU usage. Snowman creates a replica for each GUI program. Each replica completely utilizes a single CPU core for taint analysis. To make Snowman available for multiple users running multiple GUI programs simultaneously, it is necessary to provision enough processors in the datacenter beforehand based on the common load of Snowman. It is possible to run the replica in a different machine from the one where the original program runs, so it is more convenient to scale the system by dynamically migrating the replica to an idle machine.

4.5 Summary

In this chapter we presented Snowman, which aims to deter data theft by malicious insiders through strong data isolation and fine-grained data monitoring. In Snowman, a user is restricted to accessing sensitive data only remotely on a trusted server, via the GUI presented by the application to a thin client. In the server, Snowman detects data theft by monitoring the number of sensitive bytes leaked to the user. Maintaining good performance for normal usage of the monitored program while accurately monitoring for data theft is challenging. Snowman addresses this problem by replicating the execution of the program alongside its original execution and conducting multi-label taint analysis on the replicated execution. Our implementation of Snowman works on unmodified Linux binaries and off-the-shelf hardware without assuming that the replicated application is race-free (unlike some previous replication solutions). Our evaluations show that Snowman adds only moderate overhead for common user actions and, thanks to several novel optimizations, is far more efficient than, e.g., Pin-based taint analysis solutions. We also demonstrated that the data-leakage patterns of sufficiently aggressive malicious insiders can be leveraged to easily distinguish them from normal ones.

CHAPTER 5: CONCLUSION

In this dissertation we have presented the design, implementation, and evaluation of `dpprocfs`, `PoPSiCl`, and `Snowman`. These three systems all aim to efficiently protect data secrecy but face unique challenges in their own scenarios. There is hardly a one-size-fits-all solution for data protection. We choose to either noise, mask, or meter sensitive data based on sensitivity of the system data, trustworthiness of the system components, and functional requirements of the system.

`dpprocfs` applies differential privacy to introduce noise into the data reporting from the `procfs` file system so as to bound information leakage mathematically. `dpprocfs` also reestablishes invariants before releasing the noised outputs to avoid breaking applications that depend on the invariants. Our evaluations show that `dpprocfs` can simultaneously defend against known storage side-channel attacks while retaining the utility of `procfs` for monitoring and diagnosis. Our solution provides a configurable framework to suppress new storage side channels as they are discovered, through adding protection to additional kernel data-structure fields or updating the ϵ values associated with each field and application. We further believe that the mechanisms we have developed within our solution might be applicable to other storage side channels, and we plan to explore this direction in future work.

In `PoPSiCl`, a trusted cloud operator cooperates with its tenants' clients to mask server identifiers with personalized pseudonyms before transmitting server connections over untrusted networks. When instantiated for TLS-based access to tenant web servers, `PoPSiCl` works with all major browsers and requires no additional client-side software and minimal changes to the client user experience. Moreover, changes to tenant servers can be hidden in supporting software (operating systems and web-programming frameworks) without imposing on web-content development. Our security analysis shows that `PoPSiCl` enforces server anonymity when facing both passive and active network attackers. Our evaluations show that performance for `PoPSiCl` access to tenant servers is competitive with baseline HTTPS and scales well as `PoPSiCl` use grows. We thus believe that `PoPSiCl` provides a promising opportunity for cloud operators to improve privacy for its tenants' clients.

`Snowman` restricts the user to access data only remotely and accurately meters the sensitive data output to the user through the graphical user interfaces. To conduct this metering without slowing the interactive user session, leakage is concurrently tracked in a replica of the application execution. This, in turn, introduces a key technical challenge that `Snowman` solves, namely identically replicating execution of an unmodified Linux binary while also performing efficient multi-label taint-tracking on it. We show through empirical measurements with a word processor, a spreadsheet program, and a code editor that `Snowman` induces little overhead on interactive user sessions and easily differentiates data-access patterns induced by normal usage and sufficiently aggressive data theft with reasonable responsiveness.

BIBLIOGRAPHY

[1] Citrix virtual apps and desktops. `https://www.citrix.com/products/citrix-virtual-apps-and-desktops/`. Accessed: 13 May 2019.

[2] National insider threat task force. `https://www.dni.gov/index.php/ncsc-how-we-work/ncsc-nittf`. Accessed: 12 May 2019.

[3] Osaka Gas provides secure application availability with XenDesktop. `https://www.citrix.com/customers/osaka_gas_en.html`. Accessed: 13 May 2019.

[4] I. Abraham, Y. Bartal, and O. Neimany. Advances in metric embedding theory. In *38th ACM Symposium on Theory of Computing*, May 2006.

[5] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: A comparative study. *ACM Computing Surveys*, 21(4), Dec. 1989.

[6] D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 247–255, 2001.

[7] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 439–450, 2000.

[8] W. Almesberger. TCP connection passing. In *Linux Symposium*, volume 1, July 2004.

[9] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *ACM Conference on Computer and Communications Security*, 2013.

[10] A. Aurelius, C. Lagerstedt, and M. Kihl. Streaming media over the Internet: Flow based analysis in live access networks. In *Broadband Multimedia Systems and Broadcasting, 2011 IEEE International Symposium on*, 2011.

[11] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *28th Annual Computer Security Applications Conference*, 2012.

[12] R. Ayyagari. An exploratory analysis of data breaches from 2005-2011: Trends and insights. *Journal of Information Privacy and Security*, pages 33–56, 2012.

[13] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *19th International Symposium on Software Testing and Analysis*, pages 13–24, 2010.

[14] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *2003 International Conference on Dependable Systems and Networks*, pages 149–158, 2003.

[15] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 448–465, 2006.

[16] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation. In *Technical Report ESD-TR-75-306*. Mitre Corporation, 1976.

[17] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403, Oct 1997.

[18] M. Bellare, S. Keelveedhi, and T. Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, pages 179–194, 2013.

[19] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010.

[20] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *9th USENIX Conference on Operating Systems Design and Implementation*, pages 177–191, 2010.

[21] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *2nd International Conference on Virtual Execution Environments*, pages 154–163, 2006.

[22] J. M. Bland and D. G. Altman. The logrank test. *BMJ*, 328:1073, Apr. 2004.

[23] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: The sulq framework. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 128–138, 2005.

[24] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 79–90, 2009.

[25] N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Optimal geo-indistinguishable mechanisms for location privacy. In *ACM Conference on Computer and Communications Security*, 2014.

[26] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *14th International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2011.

[27] J. Boyan. The Anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine*, 4(9), Sept. 1997.

[28] C. Brubaker, A. Houmansadr, and V. Shmatikov. CloudTransport: Using cloud storage for censorship-resistant networking. In *Privacy Enhancing Technologies, 14th International Symposium*, volume 8555 of *Lecture Notes in Computer Science*. July 2014.

[29] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr. Abstractions for practical virtual machine replay. In *12th ACM International Conference on Virtual Execution Environments*, pages 93–106, 2016.

[30] L. Cai and H. Chen. TouchLogger: Inferring keystrokes on touch screen from smartphone motion. In *6th USENIX Conference on Hot Topics in Security*, 2011.

[31] L. Cai and H. Chen. On the practicality of motion based keystroke inference attack. In *Trust and Trustworthy Computing, 5th International Conference*, volume 7344 of *Lecture Notes in Computer Science*. June 2012.

[32] Y. Cao, S. Li, and E. Williams. (cross-)browser fingerprinting via OS and hardware level features. In *ISOC Network and Distributed System Security Symposium*, Feb. 2017.

[33] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *ACM Transactions on Information and System Security*, 14(3), Nov. 2011.

[34] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, pages 15:1–15:58, July 2009.

[35] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In *13th Privacy Enhancing Technologies Symposium*, July 2013.

[36] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient unlinkability. *Journal of Cryptology*, 1(1), 1988.

[37] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. Hornet: High-speed onion routing at the network layer. In *22nd ACM Conference on Computer and Communications Security*, pages 1441–1454, 2015.

[38] C. Chen and A. Perrig. Phi: Path-hidden lightweight anonymity protocol at network layer. *Proceedings on Privacy Enhancing Technologies*, 2017(1):100–117, 2017.

[39] D. Chen, J.-M. Odobez, and H. Bourlard. Text detection and recognition in images and video frames. *Pattern Recognition*, 37(3):595 – 608, 2004.

[40] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In *23th USENIX Security Symposium*, 2014.

[41] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, pages 22–22, 2004.

[42] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, pages 238–251, 2001.

[43] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, pages 181–199, 2005.

[44] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, pages 321–371, Aug. 2007.

[45] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[46] L. Constantin. Antivirus software could make your company more vulnerable. *PCWorld*, Jan. 2016. `http://goo.gl/Amju2A`.

[47] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems*, 10(1), Jan. 2014.

[48] S. Coull, M. P. Collins, C. V. Wright, F. Monrose, and M. K. Reiter. On web browsing privacy in anonymized NetFlows. In *16th USENIX Security Symposium*, Aug. 2007.

[49] S. Cowley. 2.5 million more people potentially exposed in equifax breach. `https://www.nytimes.com/2017/10/02/business/equifax-breach.html`, 2017.

[50] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232, Dec 2004.

[51] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *30th International Conference on Software Engineering*, 2008.

[52] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 748–759, 2012.

[53] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 525–540, 2014.

[54] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared-memory multiprocessing. *IEEE Micro*, pages 40–49, 2010.

[55] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation Onion Router. In *13th USENIX Security Symposium*, Aug. 2004.

[56] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 202–210, 2003.

[57] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with panda. In *5th Program Protection and Reverse Engineering Workshop*, pages 4:1–4:11, 2015.

[58] E. Dou and A. Barr. U.S. cloud providers face backlash from China's censors. *The Wall Street Journal*, 16 March 2015.

[59] 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 *5th USENIX Symposium on Operating Systems Design and implementation*, pages 211–224, 2002.

[60] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *4th ACM International Conference on Virtual Execution Environments*, pages 121–130, 2008.

[61] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488, 2014.

[62] C. Dwork. Differential privacy: A survey of results. In *Theory and Applications of Models of Computation, 5th International Conference*, volume 4978 of *Lecture Notes in Computer Science*, Apr. 2008.

[63] C. Dwork. Differential privacy in new settings. In *21st ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[64] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *42nd ACM Symposium on Theory of Computing*, 2010.

[65] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy*, May 2012.

[66] K. P. Dyer, S. E. Coull, and T. Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *24th USENIX Security Symposium*, 2015.

[67] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69, 2007.

[68] N. Feamster, M. Balazinska, W. Wang, H. Balakrishnan, and D. Karger. Thwarting web censorship with untrusted messenger discovery. In *3rd International Workshop on Privacy Enhancing Technologies*, 2003.

[69] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2, 2015.

[70] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. `https://nmap.org/nmap-fingerprinting-article.txt`, Oct. 1998.

[71] D. Goldman. Google: The reluctant censor of the Internet. *CNN Money*, 4 January 2015.

[72] D. Gugelmann, D. Sommer, V. Lenders, M. Happe, and L. Vanbever. Screen watermarking for data theft investigation and attribution. In *10th International Conference on Cyber Conflict*, May 2018.

[73] M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *ISOC Network and Distributed System Security Symposium*, 2011.

[74] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *Proceedings of the VLDB Endowment*, 3(1-2), Sept. 2010.

[75] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Internet Measurement Conference*, Oct. 2013.

[76] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 3–18, June 2012.

[77] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *1st ACM European Conference on Computer Systems*, pages 29–41, 2006.

[78] J. Holowczak and A. Houmansadr. CacheBrowser: Bypassing Chinese censorship without proxies using cached content. In *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[79] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *2008 International Symposium on Computer Architecture*, pages 265–276, June 2008.

[80] H. C. Hsiao, T. H. J. Kim, A. Perrig, A. Yamada, S. C. Nelson, M. Gruteser, and W. Meng. Lap: Lightweight anonymity and privacy. In *2012 IEEE Symposium on Security and Privacy*, pages 506–520, 2012.

[81] J. Hur and D. K. Noh. Attribute-based access control with efficient revocation in data outsourcing systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 1214–1221, July 2011.

[82] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[83] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, pages 235–246, 2013.

[84] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee. RAIN: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM Conference on Computer and Communications Security*, pages 377–390, 2017.

[85] N. Jones, M. Arye, J. Cesareo, and M. J. Freedman. Hiding amongst the clouds: A proposal for cloud-based Onion Routing. In *Free and Open Communications on the Internet*. USENIX, 2011.

[86] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A critical evaluation of website fingerprinting attacks. In *ACM Conference on Computer and Communications Security*, 2014.

[87] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 105–114, June 2009.

[88] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *8th ACM International Conference on Virtual Execution Environments*, Mar. 2012.

[89] J. Lang, A. Czeskis, D. Balfanz, M. Schilder, and S. Srinivas. Security keys: Practical cryptographic second factors for the modern web. In J. Grossklags and B. Preneel, editors, *Financial Cryptography and Data Security*, pages 422–440, 2017.

[90] J. Laurikkala, M. Juhola, and E. Kentala. Informal identification of outliers in medical data. *5th International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, July 2000.

[91] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *2001 IEEE Symposium on Security and Privacy*, pages 130–143, May 2001.

[92] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and ℓ-diversity. In *23rd International Conference on Data Engineering*, Apr. 2007.

[93] M. Liberatore and B. N. Levine. Inferring the source of encrypted HTTP connections. In *13th ACM Conference on Computer and Communications Security*, Oct. 2006.

[94] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your Android screen for secrets. In *21st ISOC Network and Distributed System Security Symposium*, 2014.

[95] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 75–86, 2009.

[96] G. Lowe. Quantifying information flow. In *Proceedings 15th IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[97] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[98] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramanian. ℓ-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering*, 2006.

[99] A. Machanavajjhala, A. Korolova, and A. D. Sarma. Personalized social recommendations: Accurate or private. *Proc. VLDB Endow.*, pages 440–450, 2011.

[100] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (Sp)iPhone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *18th ACM Conference on Computer and Communications Security*, 2011.

[101] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel&reg; software guard extensions (intel&reg; sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 10:1–10:9, 2016.

[102] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 627–636, 2009.

[103] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *ACM SIGMOD International Conference on Management of Data*, 2009.

[104] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: Your finger taps have fingerprints. In *10th International Conference on Mobile Systems, Applications, and Services*, 2012.

[105] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 308–319, 2016.

[106] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 65–80, 2015.

[107] P. Mockapetris. Domain names – implementation and specification. RFC 1035, RFC Editor, Nov. 1987. `http://www.rfc-editor.org/rfc/rfc1035.txt`.

[108] R. Moore. TLS Prober – an SSL/TLS server fingerprinting tool. `https://github.com/WestpointLtd/tls_prober/blob/master/doc/tls_prober.md`, Mar. 2015.

[109] R. Mortier, A. Madhavapeddy, T. Hong, D. Murray, and M. Schwarzkopf. Using dust clouds to enhance anonymous communication. In *18th International Workshop on Security Protocols*, 2014.

[110] A. C. Myers, A. C. Myers, and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, 1997.

[111] M. Naldi and G. D'Acquisto. Differential privacy for counting queries: Can Bayes estimation help uncover the true value? *CoRR*, abs/1407.0116, July 2014.

[112] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *18^{th} IEEE Symposium on Reliable Distributed Systems*, 1999.

[113] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture*, pages 284–295, 2005.

[114] National Computer Security Center. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, volume NCSC-TG-030 of *NSA/NCSC Rainbow Series*. Nov. 1993.

[115] J. Newsome and D. Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *ISOC Network and Distributed System Security Symposium*, Feb. 2005.

[116] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Peissens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, May 2013.

[117] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *9th ACM International Conference on Knowledge Discovery and Data Mining*, pages 631–636, 2003.

[118] J. C. Norte. Advanced Tor browser fingerprinting. `http://jcarlosnorte.com/security/2016/03/06/advanced-tor-browser-fingerprinting.html`, Mar. 2016.

[119] B. Obama. Presidential memorandum: National insider threat policy and minimum standards for executive branch insider threat programs. `https://www.dni.gov/index.php/ic-legal-reference-book/presidential-memorandum-nitp-minimum-standards-for-insider-threat-program`, 21 Nov. 2012.

[120] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *USENIX Annual Technical Conference*, pages 377–389, July 2017.

[121] G. Owen and N. Savage. The Tor dark net. No. 20, Global Commission on Internet Governance Paper Series, Sept. 2015.

[122] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website fingerprinting at Internet scale. In *ISOC Network and Distributed System Symposium*, Feb. 2016.

[123] A. Parsovs. Practical issues with TLS client certificate authentication. In *ISOC Network and Distributed System Security Symposium*, Feb. 2014.

[124] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *8th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–11, 2010.

[125] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, May 2015.

[126] A. Pfitzmann and M. Waidner. Networks without user observability. *Computers and Security*, 6(2), Apr. 1987.

[127] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *ACM European Conference on Computer Systems*, pages 15–27, 2006.

[128] 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 *39th IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Dec. 2006.

[129] S. Ragan. Hola VPN client vulnerabilities put millions of users at risk. *CSO*, Mar. 2015. `http://goo.gl/yZnkzF`.

[130] E. Rescorla. The transport layer security (tls) protocol version 1.2. `https://tools.ietf.org/html/rfc5246`, 2008.

[131] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, pages 120–126, 1978.

[132] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[133] J. Ruderman. Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`, Mar. 2016.

[134] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, pages 5–19, 2006.

[135] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, pages 38–47, Feb 1996.

[136] R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, pages 40–48, Sep. 1994.

[137] J. Sankey and M. Wright. Dovetail: Stronger anonymity in next-generation internet routing. *Proceedings on Privacy Enhancing Technologies*, pages 283–303, 2014.

[138] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[139] L. Seltzer. Research shows antivirus products vulnerable to attack. *ZDNet*, Feb. 2016. `http://goo.gl/9kbgqX`.

[140] Y. Shen and H. Jin. Privacy-preserving personalized recommendation: An instance-based approach via differential privacy. In *Proceedings of the 2014 IEEE International Conference on Data Mining*, pages 540–549, 2014.

[141] Y. Shen and H. Jin. Epicrec: Towards practical differentially private framework for personalized recommendation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 180–191, 2016.

[142] L. Simon and R. Anderson. PIN skimmer: Inferring PINs through the camera and microphone. In *3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2013.

[143] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 61–74, 2009.

[144] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Symposium on Fault Tolerant Computing*, pages 250–259, 1996.

[145] E. Snell. 58% of healthcare PHI data breaches caused by insiders. `https://healthitsecurity.com/news/58-of-healthcare-phi-data-breaches-caused-by-insiders`, 5 Mar. 2018.

[146] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, 2001.

[147] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[148] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5), 2002.

[149] P. Szoldra. This is everything Edward Snowden revealed in one year of unprecedented top-secret leaks. `https://www.businessinsider.com/snowden-leaks-timeline-2016-9`, 16 Sept. 2016.

[150] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[151] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Computing Surveys*, pages 29–41, 2005.

[152] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 639–644, 2002.

[153] J. D. Valois. *Lock-free Data Structures*. PhD thesis, 1996. Rensselaer Polytechnic Institute.

[154] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, pages 41–46, 2011.

[155] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *14th IEEE International Symposium on High Performance Computer Architecture*, pages 173–184, 2008.

[156] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in privacy preserving data mining. *ACM SIGMOD Record*, pages 50–57, 2004.

[157] P. Walker. Bradley Manning trial: what we know from the leaked WikiLeaks documents. `https://www.theguardian.com/world/2013/jul/30/bradley-manning-wikileaks-revelations`, 30 July 2013.

[158] T. Wang, X. Cai, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium*, Aug. 2014.

[159] G. Williams, R. Baxter, H. He, S. Hawkins, and L. Gu. A comparative study of RNN for outlier detection in data mining. In *2002 IEEE International Conference on Data Mining*, pages 709–712, Dec. 2002.

[160] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the network infrastructure. In *20th USENIX Security Symposium*, Aug. 2011.

[161] Q. Xiao, M. K. Reiter, and Y. Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1582–1594, 2015.

[162] Q. Xiao, M. K. Reiter, and Y. Zhang. Personalized pseudonyms for servers in the cloud. In *Proceedings on Privacy Enhancing Technologies*, PoPETS'17, 2017.

[163] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th International Symposium on Computer Architecture*, pages 122–133, June 2003.

[164] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Workshop on Modeling, Benchmarking and Simulation*, 2007.

[165] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[166] K. Yang and X. Jia. An efficient and secure dynamic auditing protocol for data storage in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, pages 1717–1726, 2013.

[167] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The performance of remote display mechanisms for thin-client computing. In *USENIX Annual Technical Conference*, pages 131–146, 2002.

[168] T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture. `https://tools.ietf.org/html/rfc4251`, 2006.

[169] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*, pages 915–930, 2015.

[170] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *20th ACM SIGSAC Conference on Computer and Communications Security*, pages 827–838. ACM, 2013.

[171] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from Android public resources. In *20th ACM Conference on Computer and Communications Security*, 2013.

[172] Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang. Static Evaluation of Noninterference Using Approximate Model Counting. In *39th IEEE Symposium on Security and Privacy*, pages 1029–1043. IEEE Computer Society, 2018.

[173] F. Zhu and J. Wei. Static analysis based invariant detection for commodity operating systems. *Computers & Security*, 43, June 2014.

[174] H. Zolfaghari and A. Houmansadr. Practical censorship evasion leveraging content delivery networks. In *ACM Conference on Computer and Communications Security*, Oct. 2016.