Mitigating Storage Side Channels Using Statistical Privacy Mechanisms

Qiuyu Xiao
University of North Carolina
Chapel Hill, NC, USA
qiuyu@cs.unc.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Yinqian Zhang
The Ohio State University
Columbus, OH, USA
yinqian@cse.osu.edu

ABSTRACT

A storage side channel occurs when an adversary accesses data objects influenced by another, victim computation and infers information about the victim that it is not permitted to learn directly. We bring advances in privacy for statistical databases to bear on storage side-channel defense, and specifically demonstrate the feasibility of applying differentially private mechanisms to mitigate storage side channels in `procfs`, a pseudo file system broadly used in Linux and Android kernels. Using a principled design with quantifiable security, our approach injects noise into kernel data-structure values that are used to generate `procfs` contents, but also reestablishes invariants on these noised values so as to not violate assumptions on which `procfs` or its clients depend. We show that our modifications to `procfs` can be configured to mitigate known storage side channels while preserving its utility for monitoring and diagnosis.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Information flow controls*

General Terms

Security

Keywords

Side channels; differential privacy

1. INTRODUCTION

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.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
DOI: http://dx.doi.org/10.1145/2810103.2813645.

In this paper, 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 [35] 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 [35]. 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 paper, 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 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 of this paper. In theory, these methods can be applied to mitigating a variety of storage side channels. However, in this paper 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 Sec. 3.1 for a summary of these attacks). Our work builds on the works of Dwork et al. [20, 21] and Chan et al. [12], 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 [13] 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. [12] 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 [25]. 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. [12] 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 paper is organized as follows. Sec. 2 summarizes related work. Sec. 3 provides an overview of storage side channel attacks via `procfs`, and the theoretical basis of d-privacy. Sec. 4 presents our design of `dpprocfs`, which is followed by details of its implementation in Sec. 5. We evaluate both the security and utility of `dpprocfs` in Sec. 6 and discuss remaining challenges in Sec. 7. We conclude the paper in Sec. 8. The proofs of propositions stated in this paper can be found in App. A.

2. RELATED WORK

Relevant to our work is privacy in the context of statistical databases. Statistical database systems allow users to query aggregate statistics of subsets of entities in the database. Privacy concerns arise when a database client learns information about individuals represented in the database through one or multiple queries to the database [4]. This concern has driven decades of innovation in stronger privacy definitions in this context and algorithms to achieve them (e.g., [4, 38, 41, 29, 18, 27, 36, 8]). Of particular interest here is differential privacy and extensions thereof; please see the works due to Dwork [19] and Fung et al. [23] that survey advances in differential privacy and compare it to other privacy models, respectively.

Prior to our work, differential privacy has been implemented in practical systems, e.g., to support privacy for data accessed through SQL-like queries [32] or MapReduce computations [37]. 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 [20, 21, 12]. 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. [13] 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 [13].) We build from this approach, defining a metric d that applies to storage side channels and implementing this defense in a working system.

While several prior works also extend the definition of differential privacy to settings that are not statistical databases (e.g., geo-location services [5, 9] and smart metering [3, 2, 17, 26, 31, 42, 7]), our work is the first to our knowledge to apply differential privacy concepts in operating system security and side-channel defense. Moreover, the domain of storage side-channel defense introduces important differences that

require innovation. In particular, since the values that our system must perturb to interfere with side channels are ones that are used by other software, it is important that our modifications do not violate invariants on which that software depends. To our knowledge, this aspect distinguishes the problem we address from work in geo-location and smart metering and drives us to a novel design as discussed in the balance of the paper.

3. BACKGROUND

3.1 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 this paper are the attacks exploiting publicly available per-process information to infer secrets of the targeted process; see Table 1 for examples. The techniques underlying these attacks are similar. Jana et al. [25] 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 footprint. Zhou et al. [44] 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. [14] 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. [28] also used `utime` to recognize a user's operation of the software keyboard on Android.

3.2 d-Privacy

In this paper we leverage a generalization of differential privacy due to Chatzikokolakis et al. [13] 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*

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

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

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 $r_i \xleftarrow{\$} \mathsf{Lap}\left(\frac{1}{\epsilon}\right)$. Then, for any $x, x' \in \mathbb{Z}^n$ and $Z \subseteq \mathbb{R}^n$,*

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

4. DESIGN OF A d-PRIVATE PROCFS

In an effort to suppress information leakages in `procfs` such as those described in Sec. 3.1, 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`.

4.1 Threat Model

This paper 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. 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.

4.2 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.

Reference	Description	procfs files used	Underlying kernel data-structure fields
Jana et al. [25]	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. [44]	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. [14]	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. [28]	Use of software keyboard detected using CPU utilization time	/proc/<pid>/stat	task_struct.utime

Table 1: Selected attacks leveraging storage side channels in the procfs file system

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 paper, 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. [44] 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)." [44, Sec. 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.*

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.*

Since *any* p-point metric space can be embedded in L1 distance with $O(\log p)$ distortion [1], 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 Sec. 4.4, 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., [24]).

4.3 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. [12], 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

$$\tilde{x}[i] = \tilde{x}[G(i)] + (x[i] - x[G(i)]) + r_i \tag{1}$$

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

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

$$r_i \sim \begin{cases} \mathsf{Lap}\left(\frac{1}{\epsilon}\right) & \text{if } i = D(i) \\ \mathsf{Lap}\left(\frac{\lfloor \log_2 i \rfloor}{\epsilon}\right) & \text{otherwise} \end{cases} \tag{3}$$

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

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

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

PROPOSITION 5 ([12]). *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. 1–3 is* $(d^*, 2\epsilon)$*-private.*

4.4 Consistency Enforcement

The values provided to `procfs` code, once rendered d^*-private by the mechanism described in Sec. 4.3, 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., [45]) to dynamic (e.g., [22]) and combinations thereof (e.g., [16]). 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 Sec. 7.) In Sec. 5.2, 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 Sec. 4.3 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 Sec. 4.3, 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 Sec. 5.3, 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 in-

variants, 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[1]) 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.

5. 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 Sec. 4.3) 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 discussed in Sec. 4.4); and a userspace daemon, `privfsd`, that interacts with the kernel extension and facilitates consistency enforcement in real time.

5.1 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}, ... 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.

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

`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 Sec. 5.3, after we discuss how data-structure invariants are identified in Sec. 5.2.

5.2 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 Sec. 4.4. 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 Sec. 4.4, 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 paper, 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[2]. By executing each benchmark application three times, we collected 22.6MB of trace files.

We then used Daikon [22] 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 $c_x \in \{-1, 0, 1\}$. 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.

The invariants produced in this way are shown in Table 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 Sec. 3.1), 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.

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

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

Table 2: Selected kernel data-structure fields (Linux kernel 3.13) and generated invariants (Sec. 5.2) that reference them. "Protected" fields are rendered d^*-private as described in Sec. 5.1, either because they have been utilized in published side-channel attacks (✠) or because they are involved in invariants that include such fields (✓).

The upper right corner of the right half of Table 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 augment our dynamic approach with static analysis (e.g., [16, 45]) 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.

5.3 Reestablishing Invariants

Upon producing $\tilde{x}[i]$ for each protected field x needed by a kernel routine to respond to a `procfs` query,[3] `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.

`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). `privfs` forms the set of relevant invariants from Table 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 that includes field x; and (ii) if any protected field x' from Table 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 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. In Sec. 6, 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.

6. 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 Sec. 6.1 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 Sec. 6.2.

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

6.1 Security Evaluation

In this section, we evaluate the capability of `dpprocfs` to defend against side-channel attacks discussed in Sec. 3.1. 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.

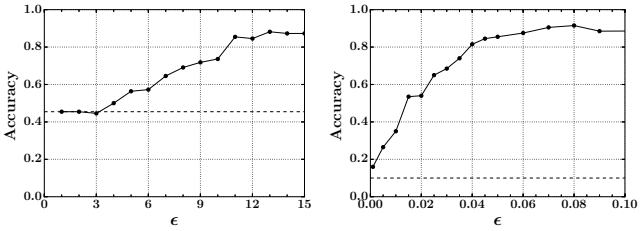
We perform this measurement of the attacker's likely success by building a multiclass classifier for classifying `procfs` features (which are attack-dependent) into one of m classes. We use the `scikit-learn`[4] support-vector-machine (SVM) implementation to build the multiclass classifier. We then report the *accuracy* of the classifier in a testing phase, namely the fraction of test instances that it classifies correctly.

6.1.1 Defending Against Keystroke Timing Attacks

The voluntary context switch counter (nvcsw in Table 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., [40]). 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 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[5] to obtain the voluntary context switch counter nvcsw, yielding six readings (a vector in $\mathbb{N}^6$). Invariant enforcement (Sec. 5.3) 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 [34, 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 Fig. 1(a). 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.



(a) Keystroke timing attack (b) Website inference attack

Figure 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

6.1.2 Mitigating Website Inference

The memory footprint of a browser can leak the website it visits (as discussed in Sec. 3.1). 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 totalVM − sharedVM (using the labels defined in Table 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 (Sec. 5.3), 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 Sec. 6.1.1, 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 Fig. 1(b). The most important distinction from the graph in Fig. 1(a) 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 set-

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

[5] This interval is much longer than in the demonstrated attack of Jana et al. [25], 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).

tings 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.

6.2 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 Sec. 5.3 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.

6.2.1 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 Sec. 6.1.2). 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 Sec. 6.1.2 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.

Fig. 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 Sec. 6.1 (see Fig. 1(b)). 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× 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 50μs—it might be preferable.

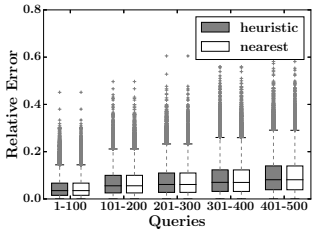


Figure 2: Comparison between nearest and heuristic invariant reestablishment for drs field; $\epsilon = 0.005$

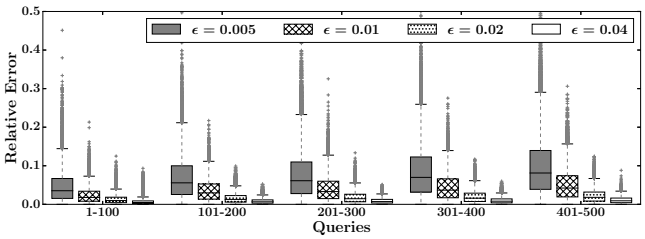


Figure 3: Relative error for drs field under nearest invariant reestablishment

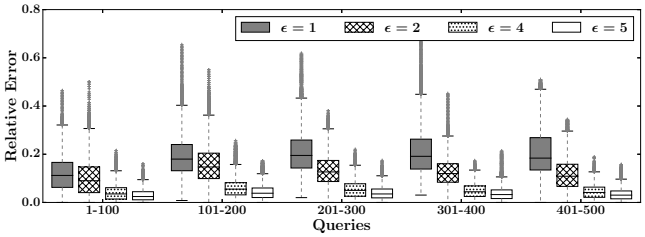


Figure 4: Relative error for utime field under nearest invariant reestablishment

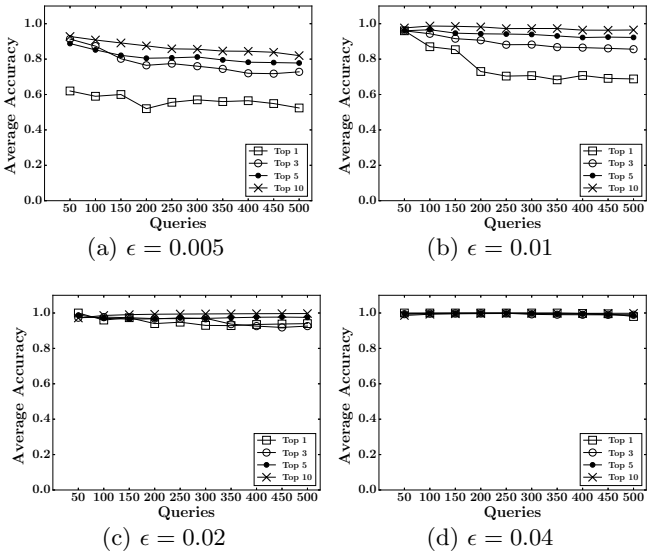
Fig. 3 and Fig. 4 represent the relative error in readings of the drs field and of the utime field from the `/proc/<pid>/stat` file, respectively, for various values of ϵ. The values of ϵ in Fig. 3 were chosen to overlap those used in the security evaluation depicted in Fig. 1(b). The ϵ values tested in Fig. 4 were chosen based on our simulation of the software-keyboard side-channel attack of Lin et al. [28], 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 in Fig. 1(a).[6] In the tests in Fig. 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 Fig. 3 and $< 30\%$ in Fig. 4, though outliers can be large.

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

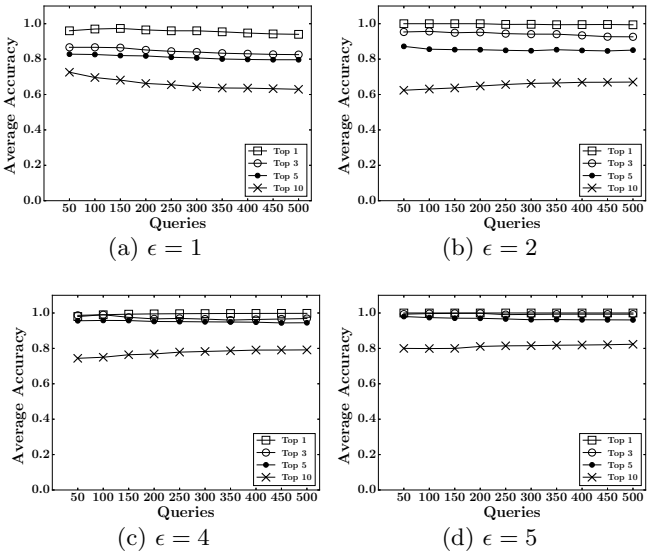
To measure the rank accuracy, we ran two `top` processes on one computer. These two `top` processes were started at the same time and updated information with the same frequency (every two seconds in our tests). The only difference was that one `top` process read from `dpprocfs` (with heuristic invariant 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

[6] 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 5: Average rank accuracy based on RES field



(a) $\epsilon = 1$ (b) $\epsilon = 2$ (c) $\epsilon = 4$ (d) $\epsilon = 5$

Figure 6: Average rank accuracy based on %CPU field

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).

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 $\frac{1}{k}|\mathsf{R}(k) \cap \mathsf{R}'(k)|$. Fig. 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. Fig. 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 Fig. 5 and Fig. 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 (Fig. 5(a) and Fig. 6(a)), 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 Fig. 5, the reverse is true in Fig. 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.

7. 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 Sec. 5.2) 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 Sec. 6.1, 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 Sec. 4.3 and Sec. 5.1). As such, the utility of `procfs` outputs generated from those fields will decay. 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. Or, as suggested in Sec. 5.1, a separate `privfs_struct` could be maintained per querying user, though that obviously weakens the mechanism against colluding users.

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. A natural way to do so would be to restart the process itself, since restarting a process also refreshes its associated kernel data structures. (This is also beneficial for performance and reliability [15].)

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., [10, 30, 43, 11, 6, 33, 39]). However, it is unclear how

adding noise, e.g., to smartphone gyroscopes, will affect the usability of the apps that rely on their readings.

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[7] 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 needs accesses to these system statistics—e.g., most traffic- and system-monitoring apps on Google Play, as well as the sysstat utilities[8]—will no longer work.

8. CONCLUSION

In this paper 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. 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.

Acknowledgments

This work was supported in part by grant 1330599 from the National Science Foundation. We are grateful to the anonymous reviewers for their helpful comments.

9. REFERENCES

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

[2] G. Ács and C. Castelluccia. I have a DREAM! (differentially private smart metering). In *13th International Conference on Information Hiding*, 2011.

[3] G. Acs, C. Castelluccia, and W. Lecat. Protecting against physical resource monitoring. In *10th ACM Workshop on Privacy in the Electronic Society*, 2011.

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

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

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

[5] 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.

[6] 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.

[7] M. Backes and S. Meiser. Differentially private smart metering with battery recharging. In *Data Privacy Management and Autonomous Spontaneous Security, 8th International Workshop*, volume 8247 of *Lecture Notes in Computer Science*. 2014.

[8] A. Blum, K. Ligett, and A. Roth. A learning theory approach to non-interactive database privacy. In *40th ACM Symposium on Theory of Computing*, 2008.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] G. Danezis, M. Kohlweiss, and A. Rial. Differentially private billing with rebates. In *13th International Conference on Information Hiding*, 2011.

[18] C. Dwork. Differential privacy. In *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, 2006.

[19] 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.

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

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

[22] 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.

[23] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys*, 42(4), June 2010.

[24] 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.

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

[26] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *11th International Conference on Privacy Enhancing Technologies*, 2011.

[27] 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.

[28] 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.

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

[30] 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.

[31] S. McLaughlin, P. McDaniel, and W. Aiello. Protecting consumer privacy from electric load monitoring. In *18th ACM Conference on Computer and Communications Security*, 2011.

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

[33] 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.

[34] 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.

[35] 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.

[36] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *39th ACM Symposium on Theory of Computing*, 2007.

[37] 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.

[38] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6), Nov. 2001.

[39] 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.

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

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

[42] D. Varodayan and A. Khisti. Smart meter privacy using a rechargeable battery: Minimizing the rate of information leakage. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2011.

[43] 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.

[44] 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.

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

APPENDIX

A. PROOFS

Proof of Prop. 1.

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

□

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. □

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 $x, x', x'' \in \mathbb{Z}^n$:

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

□

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

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

□

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

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

by Prop. 2, and so

$$\begin{aligned} &\prod_{i : i = D(i) \geq 2} \frac{\mathbb{P}\left((x[i] - x\left[\tfrac{i}{2}\right]) + r_i = z_i\right)}{\mathbb{P}\left((x'[i] - x'\left[\tfrac{i}{2}\right]) + r_i = z_i\right)} \\ &\leq \exp\left(\epsilon \times \sum_{i : i = D(i) \geq 2} \left|\left(x[i] - x\left[\frac{i}{2}\right]\right) - \left(x'[i] - x'\left[\frac{i}{2}\right]\right)\right|\right) \\ &\leq \exp\left(\epsilon \times \sum_{i \geq 2} \left|(x[i] - x[i-1]) - \left(x'[i] - x'[i-1]\right)\right|\right) \end{aligned} \tag{4}$$

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

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

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

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

And so,

$$\begin{aligned} &\prod_{i : i > D(i)} \frac{\mathbb{P}\left((x[i] - x[i - D(i)]) + r_i = z_i\right)}{\mathbb{P}\left((x'[i] - x'[i - D(i)]) + r_i = z_i\right)} \\ &= \prod_{k > 0} \prod_{j \in [0,k)} \prod_{\substack{i \,\in\, (2^k,\, 2^{k+1}) \\ :\, D(i) = 2^j}} \frac{\mathbb{P}\left((x[i] - x[i - D(i)]) + r_i = z_i\right)}{\mathbb{P}\left((x'[i] - x'[i - D(i)]) + r_i = z_i\right)} \\ &\leq \prod_{k > 0} \prod_{j \in [0,k)} \exp\left(\frac{\epsilon}{k} \times \sum_{i \in (2^k, 2^{k+1})} \left| \begin{array}{l} (x[i] - x[i-1]) \\ -(x'[i] - x'[i-1]) \end{array} \right|\right) \\ &= \prod_{k > 0} \exp\left(\epsilon \times \sum_{i \in (2^k, 2^{k+1})} \left| \begin{array}{l} (x[i] - x[i-1]) \\ -(x'[i] - x'[i-1]) \end{array} \right|\right) \\ &\leq \exp\left(\epsilon \times \sum_{i \geq 2} \left|(x[i] - x[i-1]) - \left(x'[i] - x'[i-1]\right)\right|\right) \end{aligned} \tag{5}$$

Finally, note that

$$\begin{aligned} &\frac{\mathbb{P}\left((x[1] - x[0]) + r_i = z_i\right)}{\mathbb{P}\left((x'[1] - x'[0]) + r_i = z_i\right)} \\ &\leq \exp\left(\epsilon \times |(x[1] - x[0]) - (x'[1] - x'[0])|\right) \end{aligned} \tag{6}$$

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

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

where the last step follows by multiplying Eqn. 6, Eqn. 4, and Eqn. 5. □