Vanish: Increasing Data Privacy with Self-Destructing Data

Roxana Geambasu Tadayoshi Kohno Amit A. Levy Henry M. Levy

University of Washington
{*roxana, yoshi, levya, levy*}*@cs.washington.edu*

Abstract

Today's technical and legal landscape presents formidable challenges to personal data privacy. First, our increasing reliance on Web services causes personal data to be cached, copied, and archived by third parties, often without our knowledge or control. Second, the disclosure of private data has become commonplace due to carelessness, theft, or legal actions.

Our research seeks to protect the privacy of *past, archived data* — such as copies of emails maintained by an email provider — against accidental, malicious, and legal attacks. Specifically, we wish to ensure that *all* copies of certain data become unreadable after a user-specified time, *without* any specific action on the part of a user, and even if an attacker obtains both a cached copy of that data and the user's cryptographic keys and passwords.

This paper presents *Vanish*, a system that meets this challenge through a novel integration of cryptographic techniques with global-scale, P2P, distributed hash tables (DHTs). We implemented a proof-of-concept Vanish prototype to use both the million-plus-node Vuze BitTorrent DHT and the restricted-membership OpenDHT. We evaluate experimentally and analytically the functionality, security, and performance properties of Vanish, demonstrating that it is practical to use and meets the privacy-preserving goals described above. We also describe two applications that we prototyped on Vanish: a Firefox plugin for Gmail and other Web sites and a Vanishing File application.

1 Introduction

We target the goal of creating data that self-destructs or vanishes *automatically* after it is no longer useful. Moreover, it should do so *without* any explicit action by the users or any party storing or archiving that data, in such a way that *all* copies of the data vanish simultaneously from all storage sites, online or offline.

Numerous applications could benefit from such self-destructing data. As one example, consider the case of email. Emails are frequently cached, stored, or archived by email providers (e.g., Gmail, or Hotmail), local backup systems, ISPs, etc. Such emails may cease to have value to the sender and receiver after a short period of time. Nevertheless, many of these emails are private, and the act of storing them indefinitely at intermediate locations creates a potential privacy risk. For example, imagine that Ann sends an email to her friend discussing a sensitive topic, such as her relationship with her husband, the possibility of a divorce, or how to ward off a spurious lawsuit (see Figure 1(a)). This email has no value as soon as her friend reads it, and Ann would like that *all* copies of this email — regardless of where stored or cached — be automatically destroyed after a certain period of time, rather than risk exposure in the future as part of a data breach, email provider mismanagement [41], or a legal action. In fact, Ann would prefer that these emails disappear early — and *not* be read by her friend — rather than risk disclosure to unintended parties. Both individuals and corporations could benefit from self-destructing emails.

More generally, self-destructing data is broadly applicable in today's Web-centered world, where users' sensitive data can persist "in the cloud" indefinitely (sometimes even after the user's account termination [61]). With self-destructing data, users can regain control over the lifetimes of their Web objects, such as private messages on Facebook, documents on Google Docs, or private photos on Flickr.

Numerous other applications could also benefit from self-destructing data. For example, while we do not condone their actions, the high-profile cases of several politicians [4, 62] highlight the relevance for self-destructing SMS and MMS text messages. The need for self-destructing text messages extends to the average user as well [42, 45]. As a news article states, "don't ever say anything on e-mail or text messaging that you don't want

(a) Example Scenario. (b) Vanishing Emails plugin for Gmail.

Figure 1: **Example Scenario and Vanish Email Screenshot.** (a) Ann wants to discuss her marital relationship with her friend, Carla, but does not want copies stored by intermediate services to be used in a potential child dispute trial in the future. (b) The screenshot shows how Carla reads a vanishing email that Ann has already sent to her using our Vanish Email Firefox plugin for Gmail.

to come back and bite you [42]." Some have even argued that the right and ability to destroy data are essential to protect fundamental societal goals like privacy and liberty [34, 44].

As yet another example, from a data sanitation perspective, many users would benefit from self-destructing trash bins on their desktops. These trash bins would preserve deleted files for a certain period of time, but after a timeout the files would self-destruct, becoming unavailable even to a forensic examiner (or anyone else, including the user). Moreover, the unavailability of these files would be guaranteed even if the forensic examiner is given a pristine copy of the hard drive from *before* the files self-destructed (e.g., because the machines were confiscated as part of a raid). Note that employing a whole disk encryption scheme is not sufficient, as the forensic examiner might be able to obtain the user's encryption passwords and associated cryptographic keys through legal means. Other time-limited temporary files, like those that Microsoft Word periodically produces in order to recover from a crash [17], could similarly benefit from self-destructing mechanisms.

Observation and Goals. A key observation in these examples is that users need to keep certain data for only a limited period of time. After that time, access to that data should be revoked for *everyone* — including the legitimate users of that data, the known or unknown entities holding copies of it, and the attackers. This mechanism will not be universally applicable to all users or data types; instead, we focus in particular on sensitive data that a user would prefer to see destroyed early rather than fall into the wrong hands.

Motivated by the above examples, as well as our observation above, we ask whether it is possible to create a system that can permanently delete data after a timeout:

1. even if an attacker can retroactively obtain a pristine copy of that data *and* any relevant persistent cryptographic keys and passphrases from *before* that timeout, perhaps from stored or archived copies;
2. without the use of any explicit delete action by the user or the parties storing that data;
3. without needing to modify any of the stored or archived copies of that data;
4. without the use of secure hardware; and
5. without relying on the introduction of any *new* external services that would need to be deployed (whether trusted or not).

A system achieving these goals would be broadly applicable in the modern digital world as we've previously noted, e.g., for files, private blog posts, on-line documents, Facebook entries, content-sharing sites, emails, messages, etc. In fact, the privacy of any digital content could potentially be enhanced with self-deleting data.

However, implementing a system that achieves this goal set is challenging. Section 2 describes many natural approaches that one might attempt and how they all fall short. In this paper we focus on a specific self-deleting data scheme that we have implemented, using email as an example application.

Our Approach. The key insight behind our approach and the corresponding system, called *Vanish*, is to leverage the services provided by decentralized, global-scale P2P infrastructures and, in particular, Distributed Hash Tables (DHTs). As the name implies, DHTs are designed

to implement a robust index-value database on a collection of P2P nodes [64]. Intuitively, Vanish encrypts a user's data locally with a random encryption key not known to the user, *destroys* the local copy of the key, and then sprinkles bits (Shamir secret shares [49]) of the key across random indices (thus random nodes) in the DHT.

Our choice of DHTs as storage systems for Vanish stems from three unique DHT properties that make them attractive for our data destruction goals. First, their huge scale (over 1 million nodes for the Vuze DHT [28]), geographical distribution of nodes across many countries, and complete decentralization make them robust to powerful and legally influential adversaries. Second, DHTs are designed to provide reliable distributed storage [35, 56, 64]; we leverage this property to ensure that the protected data remains available to the user for a desired interval of time. Last but not least, DHTs have an inherent property that we leverage in a unique and non-standard way: the fact that the DHT is constantly changing means that the sprinkled information will naturally disappear (vanish) as the DHT nodes churn or internally cleanse themselves, thereby rendering the protected data permanently unavailable over time. In fact, it may be impossible to determine retroactively which nodes were responsible for storing a given value in the past.

Implementation and Evaluation. To demonstrate the viability of our approach, we implemented a proof-of-concept Vanish prototype, which is capable of using either Bittorrent's Vuze DHT client [3] or the PlanetLab-hosted OpenDHT [54]. The Vuze-based system can support 8-hour timeouts in the basic Vanish usage model and the OpenDHT-based system can support timeouts up to one week.[1] We built two applications on top of the Vanish core — a Firefox plugin for Gmail and other Web sites, and a self-destructing file management application — and we intend to distribute all of these as open source packages in the near future. While prototyping on existing DHT infrastructures not designed for our purpose has limitations, it allows us to experiment at scale, have users benefit immediately from our Vanish applications, and allow others to build upon the Vanish core. Figure 1(b) shows how a user can decapsulate a vanishing email from her friend using our Gmail plugin (complete explanation of the interface and interactions is provided in Section 5). Our performance evaluation shows that simple, Vanish-local optimizations can support even latency-sensitive applications, such as our Gmail plugin, with acceptable user-visible execution times.

Security is critical for our system and hence we consider it in depth. Vanish targets *post-facto, retroactive attacks*; that is, it defends the user against future attacks on old, forgotten, or unreachable copies of her data. For example, consider the subpoena of Ann's email conversation with her friend in the event of a divorce. In this context, the attacker does not know what specific content to attack until *after* that content has expired. As a result the attacker's job is very difficult, since he must develop an infrastructure capable of attacking *all* users at *all* times. We leverage this observation to estimate the cost for such an attacker, which we deem too high to justify a viable threat. While we target no formal security proofs, we evaluate the security of our system both analytically and experimentally. For our experimental attacks, we leverage Amazon's EC2 cloud service to create a Vuze deployment and to emulate attacks against medium-scale DHTs.

Contributions. While the basic idea of our approach is simple conceptually, care must be taken in handling and evaluating the mechanisms employed to ensure its security, practicality, and performance. Looking ahead, and after briefly considering other tempting approaches for creating self-destructing data (Section 2), the key contributions of this work are to:

- identify the principal requirements and goals for self-destructing data (Section 3);
- propose a novel method for achieving these goals that combines cryptography with decentralized, global-scale DHTs (Section 4);
- demonstrate that our prototype system and applications are deployable today using existing DHTs, while achieving acceptable performance, and examine the tensions between security and availability for such deployments (Section 5);
- experimentally and analytically evaluate the privacy-preservation capabilities of our DHT-based system (Section 6).

Together, these contributions provide the foundation for empowering users with greater control over the lifetimes of private data scattered across the Internet.

2 Candidate Approaches

A number of existing and seemingly natural approaches may appear applicable to achieving our objectives. Upon deeper investigation, however, we find that none of these approaches are sufficient to achieve the goals enumerated in Section 1. We consider these strawman approaches here and use them to further motivate our design constraints in Section 3.

The most obvious approach would require users to explicitly and manually delete their data or install a `cron` job to do that. However, because Web-mails and other Web data are stored, cached, or backed up at numerous places throughout the Internet or on Web servers,

[1] We have an external mechanism to extend Vuze timeouts beyond 8 hours, which we describe later.

this approach does not seem plausible. Even for a self-destructing trash bin, requiring the user to explicitly delete data is incompatible with our goals. For example, suppose that the hard disk fails and is returned for repairs or thrown out [15]; or imagine that a laptop is stolen and the thief uses a cold-boot [32] attack to recover its primary whole-disk decryption keys (if any). We wish to ensure data destruction even in cases such as these.

Another tempting approach might be to use a standard public key or symmetric encryption scheme, as provided by systems like PGP and its open source counterpart, GPG. However, traditional encryption schemes are insufficient for our goals, as they are designed to protect against adversaries without access to the decryption keys. Under our model, though, we assume that the attacker will be able to obtain access to the decryption keys, e.g., through a court order or subpoena.[2]

A potential alternative to standard encryption might be to use forward-secure encryption [6, 13], yet our goal is strictly stronger than forward secrecy. Forward secrecy means that if an attacker learns the state of the user's cryptographic keys at some point in time, they should not be able to decrypt data encrypted at an earlier time. However, due to caching, backup archives, and the threat of subpoenas or other court orders, we allow the attacker to either view past cryptographic state or force the user to decrypt his data, thereby violating the model for forward-secure encryption. For similar reasons, plus our desire to avoid introducing new trusted agents or secure hardware, we do not use other cryptographic approaches like key-insulated [5, 23] and intrusion-resilient [21, 22] cryptography. Finally, while exposure-resilient cryptography [11, 24, 25] allows an attacker to view parts of a key, we must allow an attacker to view all of the key.

Another approach might be to use steganography [48], deniable encryption [12], or a deniable file system [17]. The idea is that one could hide, deny the contents of, or deny the existence of private historical data, rather than destroying it. These approaches are also attractive but hard to scale and automate for many applications, e.g., generating plausible cover texts for emails and photos. In addition to the problems observed with deniable file systems in [17] and [38], deniable file systems would also create additional user hassles for a trash bin application, whereas our approach could be made invisible to the user.

For online, interactive communications systems, an ephemeral key exchange process can protect derived symmetric keys from future disclosures of asymmetric private keys. A system like OTR [1, 10] is particularly attractive, but as the original OTR paper observes, this approach is not directly suited for less-interactive email applications, and similar arguments can be made for OTR's unsuitability for the other above-mentioned applications as well.

An approach with goals similar to ours (except for the goal of allowing users to create self-destructing objects without having to establish asymmetric keys or passphrases) is the Ephemerizer family of solutions [39, 46, 47]. These approaches require the introduction of one or more (possibly thresholded) trusted third parties which (informally) escrow information necessary to access the protected contents. These third parties destroy this extra data after a specified timeout. The biggest risks with such centralized solutions are that they may either not be trustworthy, or that even if they are trustworthy, users may still not trust them, hence limiting their adoption. Indeed, many users may be wary to the use of dedicated, centralized trusted third-party services after it was revealed that the Hushmail email encryption service was offering the cleartext contents of encrypted messages to the federal government [59]. This challenge calls for a decentralized approach with fewer real risks *and* perceived risks.

A second lesson can be learned from the Ephemerizer solutions in that, despite their introduction several years ago, these approaches have yet to see widespread adoption. This may in part be due to the perceived trust issues mentioned above, but an additional issue is that these solutions require the creation of new, supported and maintained services. We theorize that solutions that require *new* infrastructures have a greater barrier to adoption than solutions that can "parasitically" leverage *existing* infrastructures. A variant of this observation leads us to pursue approaches that do not require secure hardware or other dedicated services.

3 Goals and Assumptions

To support our target applications (self-destructing emails, Facebook messages, text messages, trash bins, etc.), we introduce the notion of a *vanishing data object* (*VDO*). A VDO encapsulates the user's data (such as a file or message) and prevents its contents from persisting indefinitely and becoming a source of retroactive information leakage. Regardless of whether the VDO is copied, transmitted, or stored in the Internet, it becomes unreadable after a predefined period of time even if an attacker *retroactively* obtains both a *pristine* copy of the VDO from before its expiration, and all of the user's past persistent cryptographic keys and passwords. Figure 2 illustrates the above properties of VDOs by showing the timeline for a typical usage of and attack against a VDO. We crystallize the assumptions underlying our

[2] U.S. courts are debating whether citizens are required to disclose private keys, although the ultimate verdict is unclear. We thus target technologies robust against a verdict in either direction [29, 40]. Other countries such as the U.K. [43] require release of keys, and coercion or force may be an issue in yet other countries.

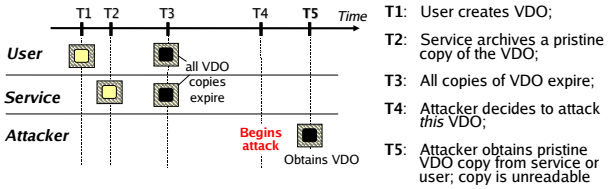


Figure 2: **Timeline for VDO usage and attack.**

VDO model and the central aspects of the threat model below.

Assumptions. Our VDO abstraction and Vanish system make several key assumptions:

1. *Time-limited value.* The VDO will be used to encapsulate data that is only of value to the user for a limited period of time.
2. *Known timeout.* When a user encapsulates data in a VDO, she knows the approximate lifetime that she wants for her VDO.
3. *Internet connectivity.* Users are connected to the Internet when interacting with VDOs.
4. *Dispensability under attack.* Rather than risk exposure to an adversary, the user prefers the VDO to be destroyed, even if prematurely.

We consider encapsulation of data that only needs to be available for hours or days; *e.g.*, certain emails, Web objects, SMSs, trash bin files, and others fall into this category. Internet connectivity is obviously required for many of our applications already, such as sending and receiving emails. More generally, the promise of ubiquitous connectivity makes this assumption reasonable for many other applications as well. Internet connectivity is not required for deletion, i.e., a VDO will become unreadable even if connectivity is removed from its storage site (or if that storage site is offline). Finally, Vanish is designed for use with data that is private, but whose persistence is not critical. That is, while the user prefers that the data remain accessible until the specified timeout, its premature destruction is preferable to its disclosure.

Goals. Having stated these assumptions, we target the following functional goals and properties for Vanish:

1. *Destruction after timeout.* A VDO must expire automatically and *without* any explicit action on the part of its users or any party storing a copy of the VDO. Once expired, the VDO must also be inaccessible to any party who obtains a *pristine* copy of the VDO from *prior* to its expiration.
2. *Accessible until timeout.* During its lifetime, a VDO's contents should be available to legitimate users.
3. *Leverage existing infrastructures.* The system must leverage existing infrastructures. It must not rely on external, special-purpose dedicated services.
4. *No secure hardware.* The system must not require the use of dedicated secure hardware.
5. *No new privacy risks.* The system should not introduce new privacy risks to the users.

A corollary of goal (1) is that the VDO will become unavailable to the legitimate users after the timeout, which is compatible with our applications and assumption of time-limited value.

Our desire to leverage existing infrastructure (goal (3)) stems from our belief that special-purpose services may hinder adoption. As noted previously, Hushmail's disclosure of the contents of users' encrypted emails to the federal government [59] suggests that, even if the centralized service or a threshold subset of a collection of centralized services is trustworthy, users may still be unwilling to trust them.

As an example of goal (5), assume that Ann sends Carla an email *without* using Vanish, and then another email using Vanish. If an attacker cannot compromise the privacy of the first email, then we require that the same attacker — regardless of how powerful — cannot compromise the privacy of the second email.

In addition to these goals, we also seek to keep the VDO abstraction as generic as possible. In Vanish, the process of encapsulating data in a VDO does *not* require users to set or remember passwords or manage cryptographic keys. However, to ensure privacy under stronger threat models, Vanish applications may compose VDOs with traditional encryption systems like PGP and GPG. In this case, the user will naturally need to manipulate the PGP/GPG keys and passphrases.

Threat Models. The above list enumerates the intended properties of the system *without* the presence of an adversary. We now consider the various classes of potential adversaries against the Vanish system, as well as the desired behavior of our system in the presence of such adversaries.

The central security goal of Vanish is to ensure the destruction of data after a timeout, despite potential adversaries who might attempt to access that data after its timeout. Obviously, care must be taken in defining what a plausible adversary is, and we do that below and in Section 6. But we also stress that we explicitly do *not* seek to preserve goal (2) — accessible prior to a timeout — in the presence of adversaries. As previously noted, we believe that users would prefer to sacrifice availability pre-timeout in favor of assured destruction for the types of data we are protecting. For example, we do not defend against denial of service attacks that could prevent reading of the data during its lifetime. Making this assumption allows us to focus on the primary novel insights in this work: methods for leveraging decentralized, large-scale P2P networks in order to make data vanish over time.

We therefore focus our threat model and subsequent analyses on attackers who wish to compromise data privacy. Two key properties of our threat model are:

1. *Trusted data owners.* Users with legitimate access to the same VDOs trust each other.
2. *Retroactive attacks on privacy.* Attackers do not know which VDOs they wish to access until *after* the VDOs expire.

The former aspect of the threat model is straightforward, and in fact is a shared assumption with traditional encryption schemes: it would be impossible for our system to protect against a user who chooses to leak or permanently preserve the cleartext contents of a VDO-encapsulated file through out-of-band means. For example, if Ann sends Carla a VDO-encapsulated email, Ann must trust Carla not to print and store a hard-copy of the email in cleartext.

The latter aspect of the threat model — that the attacker does not know the identity of a specific VDO of interest until *after* its expiration — was discussed briefly in Section 1. For example, email or SMS subpoenas typically come long after the user sends a particular sensitive email. Therefore, our system defends the user against *future attacks against old copies of private data*.

Given the retroactive restriction, an adversary would have to do some precomputation prior to the VDO's expiration. The precise form of precomputation will depend on the adversary in question. The classes of adversaries we consider include: the user's employer, the user's ISP, the user's web mail provider, and unrelated malicious nodes on the Internet. For example, foreshadowing to Section 6, we consider an ISP that might spy on the connections a user makes to the Vuze DHT on the off chance that the ISP will later be asked to assist in the retroactive decapsulation of the user's VDO. Similarly, we consider the potential for an email service to proactively try to violate the privacy of VDOs prior to expiration, for the same reason. Although we deem both situations unlikely because of public perception issues and lack of incentives, respectively, we can also provide defenses against such adversaries.

Finally, we stress that we do not seek to provide privacy against an adversary who gets a warrant to intercept *future* emails. Indeed, such an attacker would have an arsenal of attack vectors at his disposal, including not only *a priori* access to sensitive emails but also keyloggers and other forensic tools [37].

4 The Vanish Architecture

We designed and implemented Vanish, a system capable of satisfying all of the goals listed in Section 3. A key contribution of our work is to leverage existing, decentralized, large-scale Distributed Hash Tables (DHTs). After providing a brief overview of DHTs and introducing the insights that underlie our solution, we present our system's architecture and components.

Overview of DHTs. A DHT is a distributed, peer-to-peer (P2P) storage network consisting of multiple participating *nodes* [35, 56, 64]. The design of DHTs varies, but DHTs like Vuze generally exhibit a put/get interface for reading and storing data, which is implemented internally by three operations: `lookup`, `get`, and `store`. The data itself consists of an $(index, value)$ pair. Each node in the DHT manages a part of an astronomically large index name space (e.g., 2^{160} values for Vuze). To store data, a client first performs a `lookup` to determine the nodes responsible for the index; it then issues a `store` to the responsible node, who saves that $(index, value)$ pair in its local DHT database. To retrieve the value at a particular index, the client would `lookup` the nodes responsible for the index and then issue `get` requests to those nodes. Internally, a DHT may replicate data on multiple nodes to increase availability.

Numerous DHTs exist in the Internet, including Vuze, Mainline, and KAD. These DHTs are *communal*, *i.e.*, any client can join, although DHTs such as OpenDHT [54] only allow authorized nodes to join.

Key DHT-related Insights. Three key properties of DHTs make them extremely appealing for use in the context of a self-destructing data system:

1. *Availability.* Years of research in availability in DHTs have resulted in relatively robust properties of today's systems, which typically provide good availability of data prior to a specific timeout. Timeouts vary, e.g., Vuze has a fixed 8-hour timeout, while OpenDHT allows clients to choose a per-data-item timeout of up to one week.
2. *Scale, geographic distribution, and decentralization.* Measurement studies of the Vuze and Mainline DHTs estimate in excess of one million simultaneously active nodes in each of the two networks [28]. The data in [63] shows that while the U.S. is the largest single contributor of nodes in Vuze, a majority of the nodes lie outside the U.S. and are distributed over 190 countries.
3. *Churn.* DHTs evolve naturally and dynamically over time as new nodes constantly join and old nodes leave. The average lifetime of a node in the DHT varies across networks and has been measured from minutes on Kazaa [30] to hours on Vuze/Azureus [28].

The first property provides us with solid grounds for implementing a useful system. The second property makes DHTs more resilient to certain types of attacks than centralized or small-scale systems. For example,

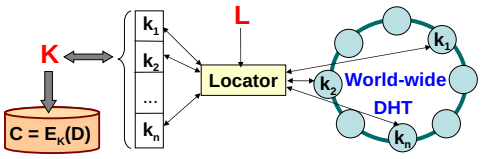


Figure 3: **The Vanish system architecture.**

while a centrally administered system can be compelled to release data by an attacker with legal leverage [59], obtaining subpoenas for multiple nodes storing a VDO's key pieces would be significantly harder, and in some cases impossible, due to their distribution under different administrative and political domains.

Traditionally, DHT research has tried to counter the negative effects of churn on availability. For our purposes, however, the constant churn in the DHT is an advantage, because it means that data stored in DHTs will naturally and irreversibly disappear over time as the DHT evolves. In many cases, trying to determine the contents of the DHT one week in the past — let alone several months or years — may be impossible, because many of the nodes storing DHT data will have left or changed their locations in the index space. For example, in Vuze, a node changes its location in the DHT whenever its IP address or port number changes, which typically happens periodically for dynamic IP addresses (*e.g.*, studies show that over 80% of the IPs change within 7 days [65]). This self-cleansing property of DHTs, coupled with its scale and global decentralization, makes them a felicitous choice for our self-destructing data system.

Vanish. Vanish is designed to leverage one or more DHTs. Figure 3 illustrates the high-level system architecture. At its core, Vanish takes a data object D (and possibly an explicit timeout T), and encapsulates it into a VDO V.

In more detail, to encapsulate the data D, Vanish picks a random data key, K, and encrypts D with K to obtain a ciphertext C. Not surprisingly, Vanish uses threshold secret sharing [58] to split the data key K into N pieces (*shares*) $K_1,\ldots,K_N$. A parameter of the secret sharing is a *threshold* that can be set by the user or by an application using Vanish. The threshold determines how many of the N shares are required to reconstruct the original key. For example, if we split the key into $N = 20$ shares and the threshold is 10 keys, then we can compute the key given any 10 of the 20 shares. In this paper we often refer to the *threshold ratio* (or simply threshold) as the percentage of the N keys required, e.g., in the example above the threshold ratio is 50%.

Once Vanish has computed the key shares, it picks at random an *access key*, L. It then uses a cryptographically secure pseudorandom number generator [7], keyed by L, to derive N indices into the DHT, $I_1,\ldots,I_N$. Vanish then sprinkles the N shares $K_1,\ldots,K_N$ at these pseudorandom locations throughout the DHT; specifically, for each $i \in \{1,\ldots,N\}$, Vanish stores the share K_i at index I_i in the DHT. If the DHT allows a variable timeout, e.g., with OpenDHT, Vanish will also set the user-chosen timeout T for each share. Once more than $(N - threshold)$ shares are lost, the VDO becomes permanently unavailable.

The final VDO V consists of $(L,C,N,threshold)$ and is sent over to the email server or stored in the file system upon encapsulation. The decapsulation of V happens in the natural way, assuming that it has not timed out. Given VDO V, Vanish (1) extracts the access key, L, (2) derives the locations of the shares of K, (3) retrieves the required number of shares as specified by the threshold, (4) reconstructs K, and (5) decrypts C to obtain D.

Threshold Secret Sharing, Security, and Robustness. For security we rely on the property that the shares $K_1,\ldots,K_N$ will disappear from the DHT over time, thereby limiting a retroactive adversary's ability to obtain a sufficient number of shares, which must be $\geq$ the threshold ratio. In general, we use a ratio of $< 100\%$, otherwise the loss of a single share would cause the loss of the key. DHTs do lose data due to churn, and therefore a smaller ratio is needed to provide robust storage prior to the timeout. We consider all of these issues in more detail later; despite the conceptual simplicity of our approach, significant care and experimental analyses must be taken to assess the durability of our use of large-scale, decentralized DHTs.

Extending the Lifetime of a VDO. For certain uses, the default timeout offered by Vuze might be too limiting. For such cases, Vanish provides a mechanism to refresh VDO shares in the DHT. While it may be tempting at first to simply use Vuze's republishing mechanism for index-value pairs, doing so would re-push the same pairs $(I_1,K_1),\ldots,(I_N,K_N)$ periodically, until the timeout. This would, in effect, increase the exposure of those key shares to certain attackers. Hence, our refresh mechanism retrieves the original data key K before its timeout, re-splits it, obtaining a fresh set of shares, and derives new DHT indices $I_1,\ldots,I_N$ as a function of L and a weakly synchronized clock. The weakly synchronized clock splits UTC time into roughly 8-hour epochs and uses the epoch number as part of the input to the location function. Decapsulations then query locations generated from both the current epoch number and the neighboring epochs, thus allowing clocks to be weakly synchronized.

Naturally, refreshes require periodic Internet connectivity. A simple home-based setup, where a broadband connected PC serves as the user's refreshing proxy, is in our view and experience a very reasonable choice given today's highly connected, highly equipped homes. In fact, we have been using this setup in our in-house deployment of Vanish in order to achieve longer timeouts for our emails (see Section 5).

Using multiple or no DHTs. As an extension to the scheme above, it is possible to store the shares of the data key K in *multiple* DHTs. For example, one might first split K into two shares K' and K'' such that both shares are required to reconstruct K. K' is then split into N' shares and sprinkled in the Vuze DHT, while K'' is split into N'' shares and sprinkled in OpenDHT. Such an approach would allow us to argue about security under different threat models, using OpenDHT's closed access (albeit small scale) and Vuze's large scale (albeit communal) access.

An alternate model would be to abandon DHTs and to store the key shares on distributed but managed nodes. This approach bears limitations similar to Ephemerizer (Section 2). A hybrid approach might be to store shares of K' in a DHT and shares of K'' on managed nodes. This way, an attacker would have to subvert both the privately managed system *and* the DHT to compromise Vanish.

Forensic Trails. Although not a common feature in today's DHTs, a future DHT or managed storage system could additionally provide a forensic trail for monitoring accesses to protected content. A custom DHT could, for example, record the IP addresses of the clients that query for particular indices and make that information available to the originator of that content. The existence of such a forensic trail, even if probabilistic, could dissuade third parties from accessing the contents of VDOs that they obtain prior to timeout.

Composition. Our system is not designed to protect against all attacks, especially those for which solutions are already known. Rather, we designed both the system and our applications to be composable with other systems to support defense-in-depth. For example, our Vanish Gmail plugin can be composed with GPG in order to avoid VDO sniffing by malicious email services. Similarly, our system can compose with Tor to ensure anonymity and throttle targeted attacks.

5 Prototype System and Applications

We have implemented a Vanish prototype capable of integrating with both Vuze and OpenDHT. In this section, we demonstrate that (1) by leveraging existing, unmodified DHT deployments we can indeed achieve the core functions of vanishing data, (2) the resulting system supports a variety of applications, and (3) the performance of VDO operations is reasonable. We focus our discussions on Vuze because its large scale and dynamic nature make its analysis both more interesting and more challenging. A key observation derived from our study is a tension in setting VDO parameters (N and threshold) when targeting both high availability prior to the timeout and high security. We return to this tension in Section 6.

To integrate Vanish with the Vuze DHT, we made two minor changes ($<$ 50 lines of code) to the existing Vuze BitTorrent client: a security measure to prevent `lookup` sniffing attacks (see Section 6.2) and several optimizations suggested by prior work [28] to achieve reasonable performance for our applications. All these changes are *local* to Vanish nodes and do not require adoption by any other nodes in the Vuze DHT.

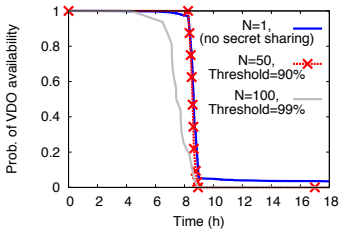


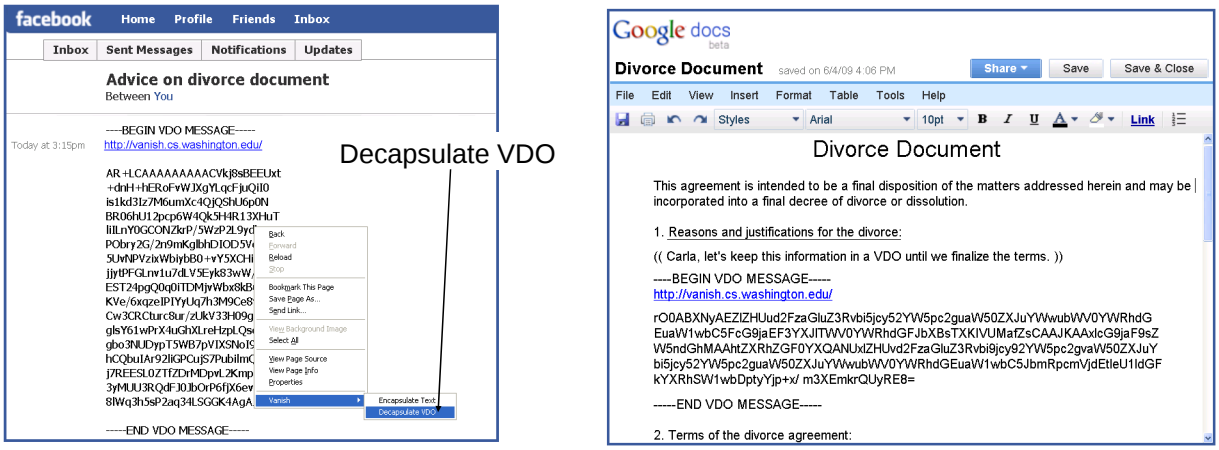
Figure 4: **VDO availability in the Vuze-based Vanish system.** The availability probability for single-key VDOs ($N = 1$) and for VDOs using secret sharing, averaged over 100 runs. Secret sharing is required to ensure pre-timeout availability and post-timeout destruction. Using $N = 50$ and a threshold of 90% achieves these goals.

5.1 Vuze Background

The Vuze (*a.k.a.* Azureus) DHT is based on the Kademlia [35] protocol. Each DHT node is assigned a "random" 160-bit ID based on its IP and port, which determines the index ranges that it will store. To store an $(\text{index}, \text{value})$ pair in the DHT, a client looks up 20 nodes with IDs closest to the specified index and then sends `store` messages to them. Vuze nodes republish the entries in their cache database every 30 minutes to the other 19 nodes closest to the value's index in order to combat churn in the DHT. Nodes further *remove* from their caches all values whose `store` timestamp is more than 8 hours old. This process has a 1-hour grace period. The originator node must re-push its 8-hour-old $(\text{index}, \text{value})$ pairs if it wishes to ensure their persistence past 8 hours.

5.2 VDO Availability and Expiration in Vuze

We ran experiments against the real global Vuze P2P network and evaluated the availability and expiration guarantees it provides. Our experiments pushed 1,000 VDO shares to pseudorandom indices in the Vuze DHT and then polled for them periodically. We repeated this experiment 100 times over a 3-day period in January 2009. Figure 4 shows the average probability that a VDO remains available as a function of the time since creation, for three different N and threshold values. For these experiments we used the standard 8-hour Vuze timeout (i.e., we did not use our refreshing proxy to re-push shares).



(a) Vanishing Facebook messages.

(b) Google Doc with vanishing parts.

Figure 5: **The Web-wide applicability of Vanish.** Screenshots of two example uses of vanishing data objects on the Web. (a) Carla is attempting to decapsulate a VDO she received from Ann in a Facebook message. (b) Ann and Carla are drafting Ann's divorce document using a Google Doc; they encapsulate sensitive, draft information inside VDOs until they finalize their position.

The $N = 1$ line shows the lifetime for a single share, which by definition does not involve secret sharing. The single-share VDO exhibits two problems: non-negligible probabilities for premature destruction ($\approx$1% of the VDOs time out before 8 hours) and prolonged availability ($\approx$5% of the VDOs continue to live long after 8 hours). The cause for the former effect is churn, which leads to early loss of the unique key for some VDOs. While the cause for the latter effect demands more investigation, we suspect that some of the single VDO keys are stored by DHT peers running non-default configurations. These observations suggest that the naive (one share) approach for storing the data key K in the DHT meets neither the availability nor the destruction goals of VDOs, thereby motivating our need for redundancy.

Secret sharing can solve the two lifetime problems seen with $N = 1$. Figure 4 shows that for VDOs with $N = 50$ and threshold of 90%, the probability of premature destruction and prolonged availability both become vanishingly small ($< 10^{-3}$). Other values for $N \geq 20$ achieve the same effect for thresholds of 90%. However, using very high threshold ratios leads to poor pre-timeout availability curves: e.g., $N = 100$ and a threshold of 99% leads to a VDO availability period of 4 hours because the loss of only two shares share makes the key unrecoverable. We will show in Section 6 that increasing the threshold increases security. Therefore, the choice of N and the threshold represents a tradeoff between security and availability. We will investigate this tradeoff further in Section 6.

5.3 Vanish Applications

We built two prototype applications that use a Vanish daemon running locally or remotely to ensure self-destruction of various types of data.

FireVanish. We implemented a Firefox plugin for the popular Gmail service that provides the option of sending and reading self-destructing emails. Our implementation requires no server-side changes. The plugin uses the Vanish daemon both to transform an email into a VDO before sending it to Gmail and similarly for extracting the contents of a VDO on the receiver side.

Our plugin is implemented as an extension of FireGPG (an existing GPG plugin for Gmail) and adds Vanish-related browser overlay controls and functions. Using our FireVanish plugin, a user types the body of her email into the Gmail text box as usual and then clicks on a *"Create a Vanishing Email"* button that the plugin overlays atop the Gmail interface. The plugin encapsulates the user's typed email body into a VDO by issuing a VDO-create request to Vanish, replaces the contents of the Gmail text box with an encoding of the VDO, and uploads the VDO email to Gmail for delivery. The user can optionally wrap the VDO in GPG for increased protection against malicious services. In our current implementation, each email is encapsulated with its own VDO, though a multi-email wrapping would also be possible (*e.g.*, all emails in the same thread).

When the receiving user clicks on one of his emails, FireVanish inspects whether it is a VDO email, a PGP email, or a regular email. Regular emails require no further action. PGP emails are first decrypted and then inspected to determine whether the underlying message is a VDO email. For VDO emails, the plugin overlays a link "*Decapsulate this email*" atop Gmail's regular interface (shown previously in Figure 1(b)). Clicking on this link causes the plugin to invoke Vanish to attempt to retrieve the cleartext body from the VDO email. If the VDO has not yet timed out, then the plugin pops up a new window showing the email's cleartext body; otherwise, an error message is displayed.

FireVanish Extension for the Web. Self-destructing data is broadly applicable in today's Web-oriented world, in which users often leave permanent traces on many Web sites [61]. Given the opportunity, many privacy-concerned users would likely prefer that certain messages on Facebook, documents on Google Docs, or instant messages on Google Talk disappear within a short period of time.

To make Vanish broadly accessible for Web usage, FireVanish provides a simple, generic, yet powerful, interface that permits all of these applications. Once the FireVanish plugin has been installed, a Firefox user can select text in any Web page input box, right click on that selected text, and cause FireVanish to replace that text *inline* with an ecapsulated VDO. Similarly, when reading a Web page containing a VDO, a user can select that VDO and right click to decapsulate it; in this case, FireVanish leaves the VDO in place and displays the cleartext in a separate popup window.

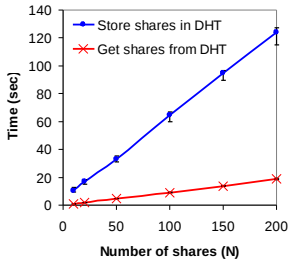
Figure 5 shows two uses of FireVanish to encapsulate and read VDOs within Facebook and Google Docs. The screenshots demonstrate a powerful concept: FireVanish can be used seamlessly to empower privacy-aware users with the ability to limit the lifetime of their data on Web applications that are unaware of Vanish.

Vanishing Files. Finally, we have implemented a vanishing file application, which can be used directly or by other applications, such as a self-destructing trash bin or Microsoft Word's autosave. Users can wrap sensitive files into self-destructing VDOs, which expire after a given timeout. In our prototype, the application creates a VDO wrapping one or more files, deletes the cleartext files from disk, and stores the VDO in their place. This ensures that, even if an attacker copies the raw bits from the laptop's disks after the timeout, the data within the VDO will be unavailable. Like traditional file encryption, Vanishing Files relies upon existing techniques for securely shredding data stored on disks or memory.

5.4 Performance Evaluation

We measured the performance of Vanish for our applications, focusing on the times to encapsulate and decapsulate a VDO. Our goals were to (1) identify the system's performance bottlenecks and propose optimizations, and (2) determine whether our Vuze-based prototype is fast enough for our intended uses. Our measurements use an Intel T2500 DUO with 2GB of RAM, Java 1.6, and a broadband network.

To identify system bottlenecks, we executed VDO operations and measured the times spent in the three main runtime components: DHT operations (storing and getting shares), Shamir secret sharing operations (splitting/recomposing the data key), and encryption/decryption. In general, the DHT component accounts for over 99% of the execution time for all Vanish operations on small and medium-size data (up to tens of MB, like most emails). For much larger data sizes (*e.g.*, files over hundreds of MB), the encryption/decryption becomes the dominating component.



(a) Scalability of DHT operations.

N	Time (seconds)		
	Encapsulate VDO		Decapsulate VDO
	Without prepush	With prepush	
10	10.5	0.082	0.9
20	16.9	0.082	2.0
50	32.8	0.082	4.7
100	64.5	0.082	9.2
150	94.7	0.082	14.0
200	124.3	0.082	19.0

(b) VDO operation execution times.

Figure 6: **Performance in the Vuze-based Vanish system.** (a) The scalability of DHT operation times as a function of the number of shares being gotten from or stored in the DHT (results are averages over 20 trials and error bars indicate standard deviations). (b) Total VDO encapsulation (with and without pre-push) and decapsulation times for FireVanish for a 2KB email, $N = 50$, and threshold 90%.

Our experiments also revealed the importance of configuring Vuze's parameters on our latency-aware applications. With no special tuning, Vuze took 4 minutes to store 50 shares, even using parallel stores. By employing several Vuze optimizations we lowered the 50-share store time by a factor of 7 (to 32 seconds). Our most effective optimization — significantly lowering Vuze's UDP timeout based on suggestions from previous research [28] — proved non-trivial, though. In particular, as we deployed Vanish within our group, we learned that different Internet providers (*e.g.*, Qwest, Comcast) exhibited utterly different network behaviors and latencies, making the setting of any one efficient value for the timeout impossible. Hence, we implemented a control-loop-based mechanism by which Vanish automatically configures Vuze's UDP timeout based on current network conditions. The optimization requires only node-local changes to Vuze.

Figure 6(a) shows how the optimized DHT operation times scale with the number of shares (N), for a fixed threshold of 90%, over a broadband connection (Comcast). Scaling with N is important in Vanish, as its se-

curity is highly dependent on this parameter. The graph shows that getting DHT shares are relatively fast — under 5 seconds for $N = 50$, which is reasonable for emails, trash bins, etc. The cost of storing VDO shares, however, can become quite large (about 30 seconds for $N = 50$), although it grows liniarly with the number of shares. To mask the store delays from the user, we implemented a simple optimization, where Vanish proactively generates data keys and pre-pushes shares into the DHT. This optimization leads to an unnoticeable DHT encapsulation time of $82ms$.

Combining the results in this section and Section 6, we believe that parameters of $N = 50$ and a threshold of 90% provide an excellent tradeoff of security and performance. With these parameters and the simple pre-push optimization we've described, user-visible latency for Vanish operations, such as creating or reading a Vanish email, is relatively low — just a few seconds for a 2KB email, as shown in Figure 6(b).

5.5 Anecdotal Experience with FireVanish

We have been using the FireVanish plugin within our group for several weeks. We also provided Vanish to several people outside of our group. Our preliminary experience has confirmed the practicality and convenience of FireVanish. We also learned a number of lessons even in this short period; for example, we found our minimalistic interface to be relatively intuitive, even for a non-CS user to whom we gave the system, and the performance is quite acceptable, as we noted above.

We also identified several limitations in the current implementation, some that we solved and others that we will address in the future. For example, in the beginning we found it difficult to search for encrypted emails or data, since their content is encrypted and opaque to the Web site. For convenience, we modified FireVanish to allow users to construct emails or other data by mixing together non-sensitive cleartext blocks with self-destructing VDOs, as illustrated in Figure 5(b). This facilitates identifying information over and above the subject line. We did find that certain types of communications indeed require timeouts longer than 8 hours. Hence, we developed and used Vanish in a proxy setting, where a Vanish server runs on behalf of a user at an online location (e.g., the user's home) and refreshes VDO shares as required to achieve each VDO's intended timeout in 8-hour units. The user can then freely execute the Vanish plugin from any connection-intermittent location (*e.g.*, a laptop).

We are planning an open-source release of the software in the near future and are confident that this release will teach us significantly more about the usability, limitations, and security of our system.

6 Security Analyses

To evaluate the security of Vanish, we seek to assess two key properties: that (1) Vanish does not introduce any *new* threats to privacy (goal (5) in Section 3), and (2) Vanish is secure against adversaries attempting to retroactively read a VDO post-expiration.

It is straightforward to see that Vanish adds no new privacy risks. In particular, the key shares stored in the DHT are *not* a function of the encapsulated data D; only the VDO is a function of D. Hence, if an adversary is unable to learn D when the user does not use Vanish, then the adversary would be unable to learn D if the user does use Vanish. There are three caveats, however. First, external parties, like the DHT, might infer information about who is communicating with whom (although the use of an anonymization system like Tor can alleviate this concern). Second, given the properties of Vanish, users might choose to communicate information that they might not communicate otherwise, thus amplifying the consequences of any successful data breach. Third, the use of Vanish might raise new legal implications. In particular, the new "eDiscovery" rules embraced by the U.S. may require a user to preserve emails and other data once in anticipation of a litigious action. The exact legal implications to Vanish are unclear; the user might need to decapsulate and save any relevant VDOs to prevent them from automatic expiration.

We focus the remainder of this section on attacks targeted at retroactively revoking the privacy of data encapsulated within VDOs (this attack timeline was shown in Figure 2). We start with a broad treatment of such attacks and then dive deeply into attacks that integrate adversarial nodes directly into the DHT.

6.1 Avoiding Retroactive Privacy Attacks

Attackers. Our motivation is to protect against retroactive data disclosures, e.g., in response to a subpoena, court order, malicious compromise of archived data, or accidental data leakage. For some of these cases, such as the subpoena, the party initiating the subpoena is the obvious "attacker." The final attacker could be a user's ex-husband's lawyer, an insurance company, or a prosecutor. But executing a subpoena is a complex process involving many other actors, including potentially: the user's employer, the user's ISP, the user's email provider, unrelated nodes on the Internet, and other actors. For our purposes, we define *all* the involved actors as the "adversary."

Attack Strategies. The architecture and standard properties of the DHT cause significant challenges to an adversary who does *not* perform any computation or data interception prior to beginning the attack. First, the key

shares are unlikely to remain in the DHT much after the timeout, so the adversary will be incapable of retrieving the shares directly from the DHT. Second, even if the adversary could legally subpoena the machines that hosted the shares in the past, the churn in Vuze makes it difficult to determine the identities of those machines; many of the hosting nodes would have long disappeared from the network or changed their DHT index. Finally, with Vuze nodes scattered throughout the globe [63], gaining legal access to those machines raises further challenges. In fact, these are all reasons why the use of a DHT such as Vuze for our application is compelling.

We therefore focus on what an attacker might do *prior* to the expiration of a VDO, with the goal of amplifying his ability to reveal the contents of the VDO in the *future*. We consider three principal strategies for such precomputation.

Strategy (1): Decapsulate VDO Prior to Expiration. An attacker might try to obtain a copy of the VDO and revoke its privacy *prior* to its expiration. This strategy makes the most sense when we consider, e.g., an email provider that proactively decapsulates all VDO emails in real-time in order to assist in responding to future subpoenas. The natural defense would be to further encapsulate VDOs in traditional encryption schemes, like PGP or GPG, which we support with our FireVanish application. The use of PGP or GPG would prevent the webmail provider from decapsulating the VDO prior to expiration. And, by the time the user is forced to furnish her PGP private keys, the VDO would have expired. For the self-destructing trash bin and the Vanishing Files application, however, the risk of this attack is minimal.

Strategy (2): Sniff User's Internet Connection. An attacker might try to intercept and preserve the data users push into or retrieve from the DHT. An ISP or employer would be most appropriately positioned to exploit this vector. Two natural defenses exist for this: the first might be to use a DHT that by default encrypts communications between nodes. Adding a sufficient level of encryption to existing DHTs would be technically straightforward assuming that the ISP or employer were passive and hence not expected to mount man-in-the-middle attacks. For the encryption, Vanish could compose with an ephemeral key exchange system in order to ensure that these encrypted communications remain private even if users' keys are later exposed. Without modifying the DHT, the most natural solution is to compose with Tor [19] to tunnel one's interactions with a DHT through remote machines. One could also use a different exit node for each share to counter potentially malicious Tor exit nodes [36, 66], or use Tor for only a subset of the shares.

Strategy (3): Integrate into DHT. An attacker might try to integrate itself into the DHT in order to: create copies of all data that it is asked to `store`; intercept internal DHT `lookup` procedures and then issue `get` requests of his own for learned indices; mount a Sybil attack [26] (perhaps as part of one of the other attacks); or mount an Eclipse attack [60]. Such DHT-integrated attacks deserve further investigation, and we provide such an analysis in Section 6.2.

We will show from our experiments in Section 6.2 that an adversary would need to join the 1M-node Vuze DHT with approximately 80,000—90,000 malicious nodes to mount a `store`-based attack and capture a reasonable percentage of the VDOs (*e.g.*, 25%). Even if possible, sustaining such an attack for an extended period of time would be prohibitively expensive (close to $860K/year in Amazon EC2 computation and networking costs). The `lookup`-based attacks are easy to defeat using localized changes to Vanish clients. The Vuze DHT already includes rudimentary defenses against the Sybil attack and a full deployment of Vanish could leverage the existing body of works focused on hardening DHTs against Sybil and Eclipse attacks [9, 14, 16, 26, 51].

Deployment Decisions. Given attack strategies (1) and (2), a user of FireVanish, Vanishing Files, or any future Vanish-based application is faced with several options: to use the basic Vanish system or to compose Vanish with other security mechanisms like PGP/GPG or Tor. The specific decision is based on the threats to the user for the application in question.

Vanish is oriented towards personal users concerned that old emails, Facebook messages, text messages, or files might come back to "bite" them, as eloquently put in [42]. Under such a scenario, an ISP trying to assist in future subpoenas seems unlikely, thus we argue that composing Vanish with Tor is unnecessary for most users. The use of Tor seems even less necessary for some of the threats we mentioned earlier, like a thief with a stolen laptop.

Similarly, it is reasonable to assume that email providers will not proactively decapsulate and archive Vanishing Emails prior to expiration. One factor is the potential illegality of such accesses under the DMCA, but even without the DMCA this seems unlikely. Therefore, users can simply employ the FireVanish Gmail plugin without needing to exchange public keys with their correspondents. However, because our plugin extends FireGPG, any user already familiar with GPG could leverage our plugin's GPG integration.

Data Sanitization. In addition to ensuring that Vanish meets its security and privacy goals, we must verify that the surrounding operating environment does not preserve information in a non-self-destructing way. For this reason, the system could leverage a broad set of ap-

proaches for sanitizing the Vanish environment, including secure methods for overwriting data on disk [31], encrypting virtual memory [50], and leveraging OS support for secure deallocation [15]. However, even absent those approaches, forensic analysis would be difficult if attempted much later than the data's expiration for the reasons we've previously discussed: by the time the forensic analysis is attempted relevant data is likely to have disappeared from the user's machine, the churn in the DHT would have made shares (and nodes) vanish irrevocably.

6.2 Privacy Against DHT-Integrated Adversaries

We now examine whether an adversary who interacts with the DHT *prior* to a VDO's expiration can, in the future, aid in retroactive attacks against the VDO's privacy. During such a precomputation phase, however, the attacker does not know which VDOs (or even which users) he might eventually wish to attack. While the attacker could compile a list of worthwhile targets (*e.g.*, politicians, actors, etc.), the use of Tor would thwart such targeted attacks. Hence, the principle strategy for the attacker would be to create a copy of as many key shares as possible. Moreover, the attacker must do this continuously — 24x7 — thereby further amplifying the burden on the attacker.

Such an attacker might be *external* to the DHT — simply using the standard DHT interface in order to obtain key shares — or *internal* to the DHT. While the former may be the only available approach for DHTs like OpenDHT, the approach is also the most limiting to an attacker since the shares are stored at pseudorandomly generated and hence unpredictable indices. An attacker integrating into a DHT like Vuze has significantly more opportunities and we therefore focus on such DHT-integrating adversaries here.

Experimental Methodology. We ran extensive experiments on a private deployment of the Vuze DHT. In each experiment, a set of honest nodes pushed VDO shares into the DHT and retrieved them at random intervals of time, while malicious nodes sniffed `stores` and `lookups`.[3] Creating our own Vuze deployment allowed us to experiment with various system parameters and workloads that we would not otherwise have been able to manipulate. Additionally, experimenting with attacks against Vuze at sufficient scale would have been prohibitively costly for us, just as it would for an attacker.

Our experiments used 1,000, 2,000, 4,500, and 8,000-node DHTs, which are significantly larger than those used for previous empirical DHT studies (e.g. 1,000 nodes in [53]). For the 8,000-node experiments we used 200 machine instances of Amazon's EC2 [2] compute cloud. For smaller experiments we used 100 of Emulab's 3GHz, 2GB machines [27]. In general, memory is the bottleneck, as each Vuze node must run in a separate process to act as a distinct DHT node. Approximately 50 Vuze nodes fit on a 2-GB machine.

Churn (node death and birth) is modeled by a Poisson distribution as in [53]. Measurements of DHT networks have observed different median lifetime distributions, e.g., 2.4 minutes for Kazaa [30], 60 minutes for Gnutella [57], and 5 hours with Vuze [28] (although this measurement may be biased towards longer-lived nodes). We believe that these vast differences stem from different content and application types that rely on these networks (e.g., the difference between audio and video clips). We chose a 2-hour median node lifetime, which provides insight into the availability—security tradeoffs under high churn.

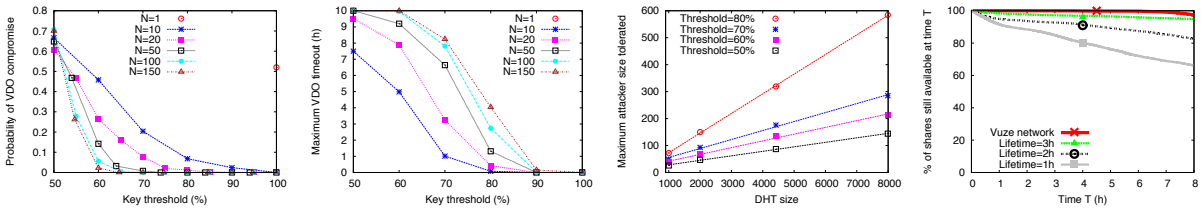
6.2.1 The Store Sniffing Attack

We first examine a `store` sniffing attack in which the adversary saves all of the index-to-value mappings it receives from peers via `store` messages. Such an attacker might receive a VDO's key shares in one of two ways: directly from the user during a VDO's creation or refresh, or via replication. In Vuze, nodes replicate their cached index-to-value mappings every 30 minutes by pushing each mapping to 20 nodes whose IDs are closest to the mapping's index.

Effects of VDO Parameters on Security. Our first goal is to assess how security is affected by the VDO parameters N (the number of key shares distributed for each VDO) and the key threshold (the percent of the N shares required to decrypt a VDO). Figure 7(a) plots the probability that an attacker can capture sufficient key shares to revoke the privacy of a given VDO as a function of N and the threshold. This figure assumes the attacker has compromised 5% of the nodes in a 1,000-node DHT. Not surprisingly, as the number of shares N increases, the attacker's success probability drops significantly. Similarly, increasing the threshold increases security (i.e., decreases the attacker's success probability).

Availability is also affected by the VDO parameters and the tradeoff is shown in Figure 7(b). Here we see the maximum timeout (i.e., the VDO's lifetime) as a function of N and the threshold. The maximum VDO timeout is the largest time at which 99% of a set of 1,000 VDOs remained available in our experiment. The timeout is capped by our 10-hour experimental limit. From the figure, we see that increasing N improves not only security, but also availability. We also see that smaller thresholds support longer timeouts, because the system can toler-

[3] Vuze `get` messages do not reveal additional information about values stored in the DHT, so we do not consider them.



(a) Parameters and security. (b) Parameters and availability. (c) Tolerated attacker sizes. (d) Churn effect on availability.

Figure 7: **Analysis of the store sniffing attack.** Fig. (a): the attacker's success probability with increasing N and key threshold for a 1000-node DHT with 50 malicious nodes. Larger N and high thresholds ($\geq 65\%$) provide good security. Fig. (b): maximum VDO timeout supported for a .99 availability level. Large N with smaller key thresholds ($\leq 70\%$) provide useful VDO timeouts. Fig. (c): maximum number of attacker nodes that a DHT can tolerate, while none of the 1,000 VDOs we pushed were compromised. Fig. (a), (b), and (c) assume 2-hour churn. Fig. (d): the single-share availability decreases over time for different churn models in our private network and for the real Vuze network.

ate more share loss. The choice of threshold thus involves a tradeoff between security and availability: high thresholds provide more security and low thresholds provide longer lifetime. For example, if a lifetime of only 4 hours is needed — which might be reasonable for certain emails or SMSs — then choosing $N = 50$ and threshold 75% leads to good security and performance. If a timeout of 8 hours is required, $N = 100$ and threshold of 70% is a good tradeoff for the 2-hour churn. Thus, by tuning N and the key threshold, we can obtain high security, good availability, and reasonable performance in the context of a small 1,000-node DHT and 5% attackers.

Attacker Sizes. We now consider how many attacker nodes a DHT deployment of a given size can tolerate with small chance that the attacker succeeds in preobtaining a sufficient number of shares for any VDO. Figure 7(c) shows the maximum attacker sizes tolerated by DHTs of increasing sizes, for various key thresholds. The values are calculated so as to ensure that none of the 1,000 VDOs we experimented with was compromised. We computed these values from experiments using $N = 150$, 2-hour churn, and various attacker sizes for each DHT size. For an 8,000-node DHT, even if 600 nodes are controlled by a store-sniffing attacker, the adversary would still not obtain any of our 1,000 VDOs.

More important, Figure 7(c) suggests that the number of attackers that the DHT can tolerate grows linearly with DHT size. Assuming this trend continues further, we estimate that, in a 1M-node DHT, an attacker with 35,000 nodes would still have less than 10^{-3} probability of recording a sufficient number of shares to compromise a single VDO with $N = 150$ and a threshold of 70%.

We have also experimented with a different metric of success: requiring an attacker to obtain enough key shares to compromise at least 25% of all VDOs. Concretely, for $N = 150$ and a threshold of 80%, our experiment with a 8,000 node DHT required the attacker to control over 710 nodes. This value also appears to grow linearly in the size of the DHT; extrapolating to a 1M-node DHT, such an attack would require at least 80,000 malicious nodes. We believe that inserting this number of nodes into the DHT, while possible for limited amounts of time, is too expensive to do continuously (we provide a cost estimate below).

Finally, we note that our refresh mechanism for extending Vuze timeouts (explained in Section 4) provides good security properties in the context of store sniffing attacks. Given that our mechanism pushes new shares in each epoch, an attacker who fails to capture sufficient shares in one epoch must start anew in the next epoch and garner the required threshold from zero.

Setting Parameters for the Vuze Network. These results provide a detailed study of the store sniffing attack in the context of a 2-hour churn model induced on a private Vuze network. We also ran a selected set of similar availability and store attack experiments against a private network with a 3-hour churn model, closer to what has been measured for Vuze.[4] The resulting availability curve for the 3-hour churn now closely resembles the one in the real Vuze network (see Figure 7(d)). In particular, for both the real network and the private network with a 3-hour churn model, a ratio of 90% and $N \geq 20$ are enough to ensure VDO availability of 7 hours with .99 probability. Thus, from an availability standpoint, the longer lifetimes allow us to raise the threshold to 90% to increase security.

From a security perspective, our experiments show that for an 8,000-node DHT, 3-hour churn model, and VDOs using $N = 50$ and threshold 90%, the attacker requires at least 820 nodes in order to obtain $\geq$25% of the VDOs. This extrapolates to a requirement of $\approx$87,000 nodes on Vuze to ensure attack effectiveness. Returning to our cost argument, while cloud computing in a system such as Amazon EC2 is generally deemed in-

[4] We used VDOs of $N = 50$ and thresholds of 90% for these experiments.

expensive [18], the cost to mount a year-long 87,000-node attack would be over $860K for processing and Internet traffic alone, which is sufficiently high to thwart an adversary's compromise plans in the context of our personal use targets (*e.g.*, seeking sensitive advice from friends over email). Of course, for larger N (*e.g.*, 150), an attacker would be required to integrate even more nodes and at higher cost. Similarly, the cost of an attack would increase as more users join the Vuze network.

Overall, to achieve good performance, security and availability, we recommend using $N = 50$ and a threshold of 90% for VDOs in the current Vuze network. Based on our experiments, we conclude that under these parameters, an attacker would be required to compromise between 8—9% of the Vuze network in order to be effective in his attack.

6.2.2 The Lookup Sniffing Attack

In addition to seeing `store` requests, a DHT-integrated adversary also sees `lookup` requests. Although Vuze only issues lookups prior to `storing` and `getting` data objects, the lookups pass through multiple nodes and hence provide additional exposure for VDO key shares. In a lookup sniffing attack, whenever an attacker node receives a lookup for an index, it actively fetches the value stored at that index, if any. While more difficult to handle than the passive `store` attack, the `lookup` attack could increase the adversary's effectiveness.

Fortunately, a simple, *node-local* change to the Vuze DHT thwarts this attack. Whenever a Vanish node wants to store to or retrieve a value from an index I, the node looks up an *obfuscated* index I', where I' is related to but different from I. The client then issues a `store`/`get` for the original index I to the nodes returned in response to the lookup for I'. In this way, the retrieving node greatly reduces the number of other nodes (and potential attackers) who see the real index.

One requirement governs our simple choice of an obfuscation function: the same set of replicas must be responsible for both indexes I and I'. Given that Vuze has 1M nodes and that IDs are uniformly distributed (they are obtained via hashing), all mappings stored at a certain node should share approximately the higher-order $\log_2(10^6) \approx 20$ bits with the IDs of the node. Thus, looking up only the first 20b of the 160b of a Vuze index is enough to ensure that the nodes resulted from the lookup are indeed those in charge of the index. The rest of the index bits are useless in lookups and can be randomized, and are rehabilitated only upon sending the final `get`/`store` to the relevant node(s). We conservatively choose to randomize the last 80b from every index looked up while retrieving or storing mappings.

Lacking full index information, the attacker would have to try retrieving all of the possible indexes starting with the obfuscated index (2^{80} indexes), which is impossible in a timely manner. This Vuze change was trivial (only 10 lines of modified code) and it is completely local to Vanish nodes. That is, the change does not require adoption by any other nodes in the DHT to be effective.

6.2.3 Standard DHT Attacks

In the previous sections we offered an in-depth analysis of two data confidentiality attacks in DHTs (store and lookup sniffing), which are specific in the context of our system. However, the robustness of communal DHTs to more general attacks has been studied profusely in the past and such analyses, proposed defenses, and limitations are relevant to Vanish, as well. Two main types of attacks identified by previous works are the Sybil attack [26] and the Eclipse (or route hijacking) attack [60]. In the Sybil attack, a few malicious nodes assume a large number of identities in the DHT. In the Eclipse attack, several adversarial nodes can redirect most of the traffic issued by honest nodes toward themselves by poisoning their routing tables with malicious node contact information [60].

The Vuze DHT already includes a rudimentary defense against Sybil attacks by constraining the identity of a Vuze node to a function of its IP address and port modulo 1999. While this measure might be sufficient for the early stages of a Vanish deployment, stronger defenses are known, e.g., certified identities [26] and periodic cryptographic puzzles [9] for defense against Sybil attacks and various other defenses against Eclipse attacks [14, 51]. Given that the core Vanish system is network-agnostic, we could easily port our system onto more robust DHTs implementing stronger defenses. Moreover, if Vanish-style systems become popular, it would also be possible to consider Vanish-specific defenses that could leverage, e.g., the aforementioned tight coupling between Vanish and the identities provided by PGP public keys. Finally, while we have focused on the Vuze DHT — and indeed its communal model makes analyzing security more interesting and challenging — Vanish could also split keys across *multiple* DHTs, or even DHTs and managed systems, as previously noted (Section 4). The different trust models, properties, and risks in those systems would present the attacker with a much more difficult task.

7 Related Work

We have discussed a large amount of related work in Section 2 and throughout the text. As additional related work, the Adeona system also leverages DHTs for increased privacy, albeit with significantly different goals [55]. Several existing companies aim to achieve

similar goals to ours (e.g., self-destructing emails), but with very different threat models (company servers must be trusted) [20]. Incidents with Hushmail, however, may lead users to question such trust models [59]. There also exists research aimed at destroying archived data where the data owner has the ability to explicitly and manually erase extra data maintained elsewhere, e.g., [8]; we avoid such processes, which may not always succeed or may be vulnerable to their own accidental copying or disclosures. Finally, albeit with different goals and perspectives, Rabin proposes an information-theoretically secure encryption system that leverages a decentralized collection of dedicated machines that continuously serve random pages of data [52], which is related to the limited storage model [33]. Communicants, who pre-share symmetric keys, can download and xor specific pages together to derive a one-time pad. The commonality between our approach and Rabin's is in the use of external machines to assist in privacy; the model, reliance on dedicated services, and pre-negotiation of symmetric keys between communicants are among the central differences.

8 Conclusions

Data privacy has become increasingly important in our litigious and online society. This paper introduced a new approach for protecting data privacy from attackers who retroactively obtain, through legal or other means, a user's stored data and private decryption keys. A novel aspect of our approach is the leveraging of the essential properties of modern P2P systems, including churn, complete decentralization, and global distribution under different administrative and political domains. We demonstrated the feasibility of our approach by presenting *Vanish*, a proof-of-concept prototype based on the Vuze global-scale DHT. Vanish causes sensitive information, such as emails, files, or text messages, to irreversibly self-destruct, without any action on the user's part and without any centralized or trusted system. Our measurement and experimental security analysis sheds insight into the robustness of our approach to adversarial attacks.

Our experience also reveals limitations of existing DHTs for Vanish-like applications. In Vuze, for example, the fixed data timeout and large replication factor present challenges for a self-destructing data system. Therefore, one exciting direction of future research is to redesign existing DHTs with our specific privacy applications in mind. Our plan to release the current Vanish system will help to provide us with further valuable experience to inform future DHT designs for privacy applications.

9 Acknowledgements

We offer special thanks to Steve Gribble, Arvind Krishnamurthy, Mark McGovern, Paul Ohm, Michael Piatek, and our anonymous reviewers for their comments on the paper. This work was supported by NSF grants NSF-0846065, NSF-0627367, and NSF-614975, an Alfred P. Sloan Research Fellowship, the Wissner-Slivka Chair, and a gift from Intel Corporation.

References

[1] C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *WPES*, 2007.

[2] Amazon.com. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`, 2008.

[3] Azureus. `http://www.vuze.com/`.

[4] BBC News. US mayor charged in SMS scandal. `http://news.bbc.co.uk/2/hi/americas/7311625.stm`, 2008.

[5] M. Bellare and A. Palacio. Protecting against key exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 16(6), 2006.

[6] M. Bellare and B. Yee. Forward security in private key cryptography. In M. Joye, editor, *CT-RSA 2003*, 2003.

[7] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, 1982.

[8] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security*, 1996.

[9] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[10] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, 2004.

[11] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 453–469, Bruges, Belgium, May 14–18, 2000. Springer-Verlag, Berlin, Germany.

[12] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In B. S. K. Jr., editor, *CRYPTO'97*, 1997.

[13] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT 2003*, 2003.

[14] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of OSDI*, 2002.

[15] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.

[16] T. Condie, V. Kacholia, S. Sankararaman, J. M. Hellerstein, and P. Maniatis. Induced churn as shelter from routing table poisoning. In *Proc. of NDSS*, 2006.

[17] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *3rd USENIX HotSec*, July 2008.

[18] M. Dama. Amazon EC2 scalable processing power. `http://www.maxdama.com/2008/08/amazon-ec2-scalable-processing-power.html`, 2008.

[19] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[20] Disappearing Inc. Disappearing Inc. product page. `http://www.specimenbox.com/di/ab/hwdi.html`, 1999.

[21] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *CT-RSA 2003*, volume 2612, pages 19–32. Springer-Verlag, Berlin, Germany, 2003.

[22] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. A generic construction for intrusion-resilient public-key encryption. In T. Okamoto, editor, *CT-RSA 2004*, volume 2964 of *LNCS*, pages 81–98, San Francisco, CA, USA, Feb. 23–27, 2004. Springer-Verlag, Berlin, Germany.

[23] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *EUROCRYPT 2002*, 2002.

[24] Y. Dodis, A. Sahai, and A. Smith. On perfect and adaptive security in exposure-resilient cryptography. In *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 301–324. Springer-Verlag, Berlin, Germany, 2001.

[25] Y. Dodis and M. Yung. Exposure-resilience for free: The case of hierarchical ID-based encryption. In *IEEE International Security In Storage Workshop*, 2002.

[26] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, 2002.

[27] Emulab. Emulab – network emulation testbed. http://www.emulab.net/, 2008.

[28] J. Falkner, M. Piatek, J. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Internet Measurement Conference*, 2007.

[29] D. Goodin. Your personal data just got permanently cached at the US border. http://www.theregister.co.uk/2008/05/01/electronic_searches_at_us_borders/, 2008.

[30] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of SOSP*, 2003.

[31] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security*, 1996.

[32] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.

[33] U. M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *Journal of Cryptology*, 5:53–66, 1992.

[34] V. Mayer-Schoenberger. Useful Void: the art of forgetting in the age of ubiquitous computing. *Working Paper, John F. Kennedy School of Government, Harvard University*, 2007.

[35] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of Peer-to-Peer Systems*, 2002.

[36] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies Symposium*, July 2008.

[37] D. McCullagh. Feds use keylogger to thwart PGP, Hushmail. news.cnet.com/8301-10784_3-9741357-7.html, 2008.

[38] D. McCullagh. Security guide to customs-proofing your laptop. http://www.news.com/8301-13578_3-9892897-38.html, 2008.

[39] S. K. Nair, M. T. Dashti, B. Crispo, and A. S. Tanenbaum. A hybrid PKI-IBC based ephemerizer system. In *International Information Security Conference*, 2007.

[40] E. Nakashima. Clarity sought on electronic searches. http://www.washingtonpost.com/wp-dyn/content/article/2008/02/06/AR2008020604763.html, 2008.

[41] New York Times. F.B.I. Gained Unauthorized Access to E-Mail. http://www.nytimes.com/2008/02/17/washington/17fisa.html?_r=1&hp=&adxnnl=1&oref=slogin&adxnnlx=1203255399-44ri626iqXg7QNmwzoeRkA, 2008.

[42] News 24. Think before you SMS. http://www.news24.com/News24/Technology/News/0,,2-13-1443_1541201,00.html, 2004.

[43] Office of Public Sector Information. Regulation of Investigatory Powers Act (RIPA), Part III – Investigation of Electronic Data Protected by Encryption etc. http://www.opsi.gov.uk/acts/acts2000/ukpga_20000023_en_8, 2000.

[44] P. Ohm. The Fourth Amendment right to delete. *The Harvard Law Review*, 2005.

[45] PC Magazine. Messages can be forever. http://www.pcmag.com/article2/0,1759,1634544,00.asp, 2004.

[46] R. Perlman. The Ephemerizer: Making data disappear. *Journal of Information System Security*, 1(1), 2005.

[47] R. Perlman. File system design with assured delete. In *Security in Storage Workshop (SISW)*, 2005.

[48] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding: A survey. *Proceedings of the IEEE*, 87(7), 1999.

[49] B. Poettering. "ssss: Shamir's Secret Sharing Scheme". http://point-at-infinity.org/ssss/, 2006.

[50] N. Provos. Encrypting virtual memory. In *USENIX Security*, 2000.

[51] K. P. N. Puttaswamy, H. Zheng, and B. Y. Zhao. Securing structured overlays against identity attacks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2008.

[52] M. O. Rabin. Provably unbreakable hyper-encryption in the limited access model. In *IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, 2005.

[53] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of the Annual Technical Conf.*, 2004.

[54] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of ACM SIGCOMM*, 2005.

[55] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In *17th USENIX Security Symposium*, 2008.

[56] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, 2001.

[57] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, 2002.

[58] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[59] R. Singel. Encrypted e-mail company Hushmail spills to feds. http://blog.wired.com/27bstroke6/2007/11/encrypted-e-mai.html, 2007.

[60] A. Singh, T. W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proc. of INFOCOM*, 2006.

[61] Slashdot. http://tech.slashdot.org/article.pl?sid=09/02/17/2213251&tid=267, 2009.

[62] Spitzer criminal complaint. http://nytimes.com/packages/pdf/nyregion/20080310spitzer-complaint.pdf, 2008.

[63] M. Steiner and E. W. Biersack. Crawling Azureus. Technical Report RR-08-223, 2008.

[64] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, pages 149–160, 2001.

[65] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldszmidt, and T. Wobber. How dynamic are IP addresses? In *Proc. of SIGCOMM*, 2007.

[66] K. Zetter. Tor researcher who exposed embassy e-mail passwords gets raided by Swedish FBI and CIA. http://blog.wired.com/27bstroke6/2007/11/swedish-researc.html, 2007.

Efficient Data Structures for Tamper-Evident Logging

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

Many real-world applications wish to collect tamper-evident logs for forensic purposes. This paper considers the case of an untrusted logger, serving a number of clients who wish to store their events in the log, and kept honest by a number of auditors who will challenge the logger to prove its correct behavior. We propose semantics of tamper-evident logs in terms of this auditing process. The logger must be able to prove that individual logged events are still present, and that the log, as seen now, is consistent with how it was seen in the past. To accomplish this efficiently, we describe a tree-based data structure that can generate such proofs with logarithmic size and space, improving over previous linear constructions. Where a classic hash chain might require an 800 MB trace to prove that a randomly chosen event is in a log with 80 million events, our prototype returns a 3 KB proof with the same semantics. We also present a flexible mechanism for the log server to present authenticated and tamper-evident search results for all events matching a predicate. This can allow large-scale log servers to selectively delete old events, in an agreed-upon fashion, while generating efficient proofs that no inappropriate events were deleted. We describe a prototype implementation and measure its performance on an 80 million event syslog trace at 1,750 events per second using a single CPU core. Performance improves to 10,500 events per second if cryptographic signatures are offloaded, corresponding to 1.1 TB of logging throughput per week.

1 Introduction

There are over 10,000 U.S. regulations that govern the storage and management of data [22, 58]. Many countries have legal, financial, medical, educational and privacy regulations that require businesses to retain a variety of records. Logging systems are therefore in wide use (albeit many without much in the way of security features).

Audit logs are useful for a variety of forensic purposes, such as tracing database tampering [59] or building a versioned filesystem with verifiable audit trails [52]. Tamper-evident logs have also been used to build Byzantine fault-tolerant systems [35] and protocols [15], as well as to detect misbehaving hosts in distributed systems [28].

Ensuring a log's integrity is a critical component in the security of a larger system. Malicious users, including insiders with high-level access and the ability to subvert the logging system, may want to perform unlogged activities or tamper with the recorded history. While tamper-resistance for such a system might be impossible, tamper-detection should be guaranteed in a strong fashion.

A variety of hash data structures have been proposed in the literature for storing data in a tamper-evident fashion, such as trees [34, 49], RSA accumulators [5, 11], skip lists [24], or general authenticated DAGs. These structures have been used to build certificate revocation lists [49], to build tamper-evident graph and geometric searching [25], and authenticated responses to XML queries [19]. All of these store static data, created by a *trusted author* whose signature is used as a root-of-trust for authenticating responses of a lookup queries.

While authenticated data structures have been adapted for dynamic data [2], they continue to assume a trusted author, and thus they have no need to detect inconsistencies across versions. For instance, in SUNDR [36], a trusted network filesystem is implemented on untrusted storage. Although version vectors [16] are used to detect when the server presents forking-inconsistent views to clients, only trusted clients sign updates for the filesystem.

Tamper-evident logs are fundamentally different: An *untrusted* logger is the sole author of the log and is responsible for both building and signing it. A log is a dynamic data structure, with the author signing a stream of commitments, a new commitment each time a new event is added to the log. Each commitment *snapshots* the entire log up to that point. If each signed commitment is the root of an authenticated data structure, well-known authenticated dictionary techniques [62, 42, 20] can detect tampering *within* each snapshot. However, without additional mechanisms to prevent it, an untrusted logger is free to have different snapshots make *inconsistent claims about the past*. To be secure, a tamper-evident log system must both detect tampering within each signed log *and* detect when different instances of the log make inconsistent claims.

Current solutions for detecting when an untrusted server is making inconsistent claims over time require linear space and time. For instance, to prevent undetected tampering, existing tamper evident logs [56, 17, 57] which rely upon a hash chain require auditors examine every intermediate event between snapshots. One proposal [43] for a tamper-evident log was based on a skip list. It has logarithmic lookup times, assuming the log

is known to be internally consistent. However, proving internal consistency requires scanning the full contents of the log. (See Section 3.4 for further analysis of this.)

In the same manner, CATS [63], a network-storage service with strong accountability properties, snapshots the internal state, and only probabilistically detects tampering by auditing a subset of objects for correctness between snapshots. Pavlou and Snodgrass [51] show how to integrate tamper-evidence into a relational database, and can prove the existence of tampering, if suspected. Auditing these systems for consistency is expensive, requiring each auditor visit each snapshot to confirm that any changes between snapshots are authorized.

If an untrusted logger knows that a just-added event or returned commitment will not be audited, then any tampering with the added event or the events fixed by that commitment will be undiscovered, and, by definition, the log is not tamper-evident. To prevent this, *a tamper-evident log requires frequent auditing*. To this end, we propose a tree-based history data structure, logarithmic for all auditing and lookup operations. Events may be added to the log, commitments generated, and audits may be performed independently of one another and at any time. No batching is used. Unlike past designs, we explicitly focus on how tampering will be discovered, through auditing, and we optimize the costs of these audits. Our *history tree* allows loggers to efficiently prove that the sequence of individual logs committed to, over time, make consistent claims about the past.

In Section 2 we present background material and propose semantics for tamper-evident logging. In Section 3 we present the history tree. In Section 4 we describe *Merkle aggregation*, a way to annotate events with attributes which can then be used to perform tamper-evident queries over the log and *safe deletion* of events, allowing unneeded events to be removed in-place, with no additional trusted party, while still being able to prove that no events were improperly purged. Section 5 describes a prototype implementation for tamper-evident logging of syslog data traces. Section 6 discusses approaches for scaling the logger's performance. Related work is presented in Section 7. Future work and conclusions appear in Section 8.

2 Security Model

In this paper, we make the usual cryptographic assumptions that an attacker cannot forge digital signatures or find collisions in cryptographic hash functions. Furthermore we are not concerned with protecting the secrecy of the logged events; this can be addressed with external techniques, most likely some form of encryption [50, 26, 54]. For simplicity, we assume a single monolithic log on a single host computer. Our goal is to detect tampering. It is impractical to prevent the destruction or alteration of digital records that are in the custody of a Byzantine logger. Replication strategies, outside the scope of this paper, can help ensure availability of the digital records [44].

Tamper-evidence requires auditing. If the log is never examined, then tampering cannot be detected. To this end, we divide a logging system into three logical entities—many *clients* which generate events for appending to a log or history, managed on a centralized but totally untrusted *logger*, which is ultimately audited by one or more trusted *auditors*. We assume clients and auditors have very limited storage capacity while loggers are assumed to have unlimited storage. By auditing the published commitments and demanding proofs, auditors can be convinced that the log's integrity has been maintained. At least one auditor is assumed to be incorruptible. In our system, we distinguish between clients and auditors, while a single host could, in fact, perform both roles.

We must trust clients to behave correctly while they are following the event insertion protocol, but we trust clients nowhere else. Of course, a malicious client could insert garbage, but we wish to ensure that an event, once correctly inserted, cannot be undetectably hidden or modified, even if the original client is subsequently colluding with the logger in an attempt to tamper with old data.

To ensure these semantics, an untrusted logger must regularly prove its correct behavior to auditors and clients. *Incremental proofs*, demanded of the logger, prove that current commitment and prior commitment make consistent claims about past events. *Membership proofs* ask the logger to return a particular event from the log along with a proof that the event is consistent with the current commitment. Membership proofs may be demanded by clients after adding events or by auditors verifying that older events remain correctly stored by the logger. These two styles of proofs are sufficient to yield tamper-evidence. As any vanilla lookup operation may be followed by a request for proof, the logger must behave faithfully or risk its misbehavior being discovered.

2.1 Semantics of a tamper evident history

We now formalize our desired semantics for secure histories. Each time an event X is sent to the logger, it assigns an index i and appends it to the log, generating a version-i commitment C_i that depends on all of the events to-date, $X_0 \ldots X_i$. The commitment C_i is bound to its version number i, signed, and published.

Although the stream of histories that a logger commits to $(C_0 \ldots C_i, C_{i+1}, C_{i+2} \ldots)$ are supposed to be mutually-consistent, each commitment fixes an *independent* history. Because histories are not known, a priori, to be consistent with one other, we will use primes ($'$) to distinguish between different histories and the events contained within them. In other words, the events in log C_i (i.e., those committed by commitment C_i) are $X_0 \ldots X_i$

and the events in log C'_j are $X'_0 \ldots X'_j$, and we will need to prove their correspondence.

2.1.1 Membership auditing

Membership auditing is performed both by clients, verifying that new events are correctly inserted, and by auditors, investigating that old events are still present and unaltered. The logger is given an event index i and a commitment C_j, $i \leq j$ and is required to return the ith element in the log, X_i, and a proof that C_j implies X_i is the ith event in the log.

2.1.2 Incremental auditing

While a verified membership proof shows that an event was logged correctly in *some* log, represented by its commitment C_j, additional work is necessary to verify that the sequence of logs committed by the logger is consistent over time. In *incremental auditing*, the logger is given two commitments C_j and C'_k, where $j \leq k$, and is required to prove that the two commitments make consistent claims about past events. A verified incremental proof demonstrates that $X_a = X'_a$ for all $a \in [0, j]$. Once verified, the auditor knows that C_j and C'_k commit to the same shared history, and the auditor can safely discard C_j.

A dishonest logger may attempt to tamper with its history by rolling back the log, creating a new fork on which it inserts new events, and abandoning the old fork. Such tampering will be caught if the logging system satisfies *historical consistency* (see Section 2.3) and by a logger's inability to generate an incremental proof between commitments on different (and inconsistent) forks when challenged.

2.2 Client insertion protocol

Once clients receive commitments from the logger after inserting an event, they must immediately redistribute them to auditors. This prevents the clients from subsequently colluding with the logger to roll back or modify their events. To this end, we need a mechanism, such as a gossip protocol, to distribute the signed commitments from clients to multiple auditors. It's unnecessary for every auditor to audit every commitment, so long as some auditor audits every commitment. (We further discuss tradeoffs with other auditing strategies in Section 3.1.)

In addition, in order to deal with the logger presenting different views of the log to different auditors and clients, auditors must obtain and reconcile commitments received from multiple clients or auditors, perhaps with the gossip protocol mentioned above. Alternatively the logger may publish its commitment in a public fashion so that all auditors receive the same commitment [27]. All that matters is that auditors have access to a diverse collection of commitments and demand incremental proofs to verify that the logger is presenting a consistent view.

2.3 Definition: tamper evident history

We now define a tamper-evident history system as a five-tuple of algorithms:

$H.\textsc{Add}(X) \rightarrow C_j$**.** Given an event X, appends it to the history, returning a new commitment.

$H.\textsc{Incr.Gen}(C_i, C_j) \rightarrow P$**.** Generates an incremental proof between C_i and C_j, where $i \leq j$.

$H.\textsc{Membership.Gen}(i, C_j) \rightarrow (P, X_i)$**.** Generates a membership proof for event i from commitment C_j, where $i \leq j$. Also returns the event, X_i.

$P.\textsc{Incr.Vf}(C'_i, C_j) \rightarrow \{\top, \bot\}$**.** Checks that P proves that C_j fixes every entry fixed by C'_i (where $i \leq j$). Outputs $\top$ if no divergence has been detected.

$P.\textsc{Membership.Vf}(i, C_j, X'_i) \rightarrow \{\top, \bot\}$**.** Checks that P proves that event X'_i is the i'th event in the log defined by C_j (where $i \leq j$). Outputs $\top$ if true.

The first three algorithms run on the logger and are used to append to the log H and to generate *proofs* P. Auditors or clients verify the proofs with algorithms $\{\textsc{Incr.Vf}, \textsc{Membership.Vf}\}$. Ideally, the proof P sent to the auditor is more concise than retransmitting the full history H. Only commitments need to be signed by the logger. Proofs do not require digital signatures; either they demonstrate consistency of the commitments and the contents of an event or they don't. With these five operations, we now define "tamper evidence" as a system satisfying:

Historical Consistency If we have a valid incremental proof between two commitments C_j and C_k, where $j \leq k$, ($P.\textsc{Incr.Vf}(C_j, C_k) \rightarrow \top$), and we have a valid membership proof P' for the event X'_i, where $i \leq j$, in the log fixed by C_j (i.e., $P'.\textsc{Membership.Vf}(i, C_j, X'_i) \rightarrow \top$) and a valid membership proof for X''_i in the log fixed by C_k (i.e., $P''.\textsc{Membership.Vf}(i, C_k, X''_i) \rightarrow \top$), then X'_i must equal X''_i. (In other words, if two commitments commit consistent histories, then they must both fix the same events for their shared past.)

2.4 Other threat models

Forward integrity Classic tamper-evident logging uses a different threat model, forward integrity [4]. The forward integrity threat model has two entities: clients who are fully trusted but have limited storage, and loggers who are assumed to be honest until suffering a Byzantine failure. In this threat model, the logger must be prevented from undetectably tampering with events logged prior to the Byzantine failure, but is allowed to undetectably tamper with events logged after the Byzantine failure.

Although we feel our threat model better characterizes the threats faced by tamper-evident logging, our history

tree and the semantics for tamper-evident logging are applicable to this alternative threat model with only minor changes. Under the semantics of forward-integrity, membership auditing just-added events is unnecessary because tamper-evidence only applies to events occurring before the Byzantine failure. Auditing a just-added event is unneeded if the Byzantine failure hasn't happened and irrelevant afterwards. Incremental auditing is still necessary. A client must incrementally audit received commitments to prevent a logger from tampering with events occurring before a Byzantine failure by rolling back the log and creating a new fork. Membership auditing is required to look up and examine old events in the log.

Itkis [31] has a similar threat model. His design exploited the fact that if a Byzantine logger attempts to roll back its history to before the Byzantine failure, the history must fork into two parallel histories. He proposed a procedure that tested two commitments to detect divergence without online interaction with the logger and proved an $O(n)$ lower bound on the commitment size. We achieve a tighter bound by virtue of the logger cooperating in the generation of these proofs.

Trusted hardware Rather than relying on auditing, an alternative model is to rely on the logger's hardware itself to be tamper-resistant [58, 1]. Naturally, the security of these systems rests on protecting the trusted hardware and the logging system against tampering by an attacker with complete physical access. Although our design could certainly use trusted hardware as an auditor, cryptographic schemes like ours rest on simpler assumptions, namely the logger can and must prove it is operating correctly.

3 History tree

We now present our new data structure for representing a tamper-evident history. We start with a Merkle tree [46], which has a long history of uses for authenticating static data. In a Merkle tree, data is stored at the leaves and the hash at the root is a tamper-evident summary of the contents. Merkle trees support logarithmic path lengths from the root to the leaves, permitting efficient random access. Although Merkle trees are a well-known tamper-evident data structure and our use is straightforward, the novelty in our design is in using a versioned computation of hashes over the Merkle tree to efficiently prove that different log snapshots, represented by Merkle trees, with *distinct* root hashes, make consistent claims about the past.

A filled history tree of depth d is a binary Merkle hash tree, storing 2^d events on the leaves. Interior nodes, $I_{i,r}$ are identified by their index i and layer r. Each leaf node $I_{i,0}$, at layer 0, stores event X_i. Interior node $I_{i,r}$ has left child $I_{i,r-1}$ and right child $I_{i+2^{r-1},r-1}$. (Figures 1 through 3 demonstrate this numbering scheme.) When a tree is not full, subtrees containing no events are represented as $\square$. This can be seen starting in Figure 1, a version-2 tree having three events. Figure 2 shows a version-6 tree, adding four additional events. Although the trees in our figures have a depth of 3 and can store up to 8 leaves, our design clearly extends to trees with greater depth and more leaves.

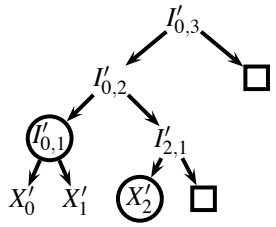


Figure 1: A version 2 history with commitment $C'_2 = I'_{0,3}$.

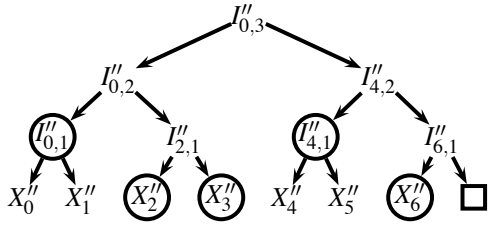


Figure 2: A version 6 history with commitment $C''_6 = I''_{0,3}$.

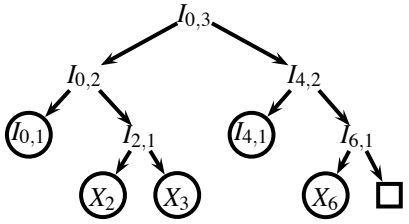


Figure 3: An incremental proof P between a version 2 and version 6 commitment. Hashes for the circled nodes are included in the proof. Other hashes can be derived from their children. Circled nodes in Figures 1 and 2 must be shown to be equal to the corresponding circled nodes here.

Each node in the history tree is *labeled* with a cryptographic hash which, like a Merkle tree, fixes the contents of the subtree rooted at that node. For a leaf node, the label is the hash of the event; for an interior node, the label is the hash of the concatenation of the labels of its children.

An interesting property of the history tree is the ability to efficiently reconstruct old versions or *views* of the tree. Consider the history tree given in Figure 2. The logger could reconstruct C''_2 analogous to the version-2 tree in Figure 1 by pretending that nodes $I''_{4,2}$ and X''_3 were $\square$ and then recomputing the hashes for the interior nodes and the root. If the reconstructed C''_2 matched a previously advertised commitment C'_2, then both trees must have the same contents and commit the same events.

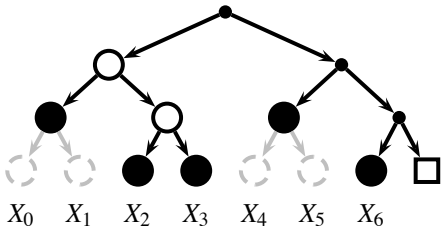


Figure 4: Graphical notation for a history tree analogous to the proof in Figure 3. Solid discs represent hashes included in the proof. Other nodes are not included. Dots and open circles represent values that can be recomputed from the values below them; dots may change as new events are added while open circles will not. Grey circle nodes are unnecessary for the proof.

This forms the intuition of how the logger generates an incremental proof P between two commitments, C'_2 and C''_6. Initially, the auditor only possesses commitments C'_2 and C''_6; it does not know the underlying Merkle trees that these commitments fix. The logger must show that both histories commit the same events, i.e., $X''_0 = X'_0, X''_1 = X'_1$, and $X''_2 = X'_2$. To do this, the logger sends a *pruned tree* P to the auditor, shown in Figure 3. This pruned tree includes just enough of the full history tree to compute the commitments C_2 and C_6. Unnecessary subtrees are *elided* out and replaced with *stubs*. Events can be either included in the tree or replaced by a stub containing their hash. Because an incremental proof involves *three* history trees, the trees committed by C'_2 and C''_6 with unknown contents and the pruned tree P, we distinguish them by using a different number of primes ($'$).

From P, shown in Figure 3, we reconstruct the corresponding root commitment for a version-6 tree, C_6. We recompute the hashes of interior nodes based on the hashes of their children until we compute the hash for node $I_{0,3}$, which will be the commitment C_6. If $C''_6 = C_6$ then the corresponding nodes, circled in Figures 2 and 3, in the pruned tree P and the implicit tree committed by C''_6 must match.

Similarly, from P, shown in Figure 3, we can reconstruct the version-2 commitment C_2 by pretending that the nodes X_3 and $I_{4,2}$ are $\square$ and, as before, recomputing the hashes for interior nodes up to the root. If $C'_2 = C_2$, then the corresponding nodes, circled in Figures 1 and 3, in the pruned tree P and the implicit tree committed by C'_2 must match, or $I'_{0,1} = I_{0,1}$ and $X'_2 = X_2$.

If the events committed by C'_2 and C''_6 are the same as the events committed by P, then they must be equal; we can then conclude that the tree committed by C''_6 is consistent with the tree committed by C'_2. By this we mean that the history trees committed by C'_2 and C''_6 both commit the same events, or $X''_0 = X'_0$, $X''_1 = X'_1$, and $X''_2 = X'_2$, even though the events $X''_0 = X'_0$, $X''_1 = X'_1$, X''_4, and X''_5 are unknown to the auditor.

3.1 Is it safe to skip nodes during an audit?

In the pruned tree in Figure 3, we omit the events fixed by $I_{0,1}$, yet we still preserve the semantics of a tamper-evident log. Even though these earlier events may not be sent to the auditor, they are still fixed by the unchanged hashes above them in the tree. Any attempted tampering will be discovered in future incremental or membership audits of the skipped events. With the history tree, auditors only receive the portions of the history they need to audit the events they have chosen to audit. Skipping events makes it possible to conduct a variety of selective audits and offers more flexibility in designing auditing policies.

Existing tamper-evident log designs based on a classic hash-chain have the form $C_i = H(C_{i-1} \parallel X_i), C_{-1} = \square$ and do not permit events to be skipped. With a hash chain, an incremental or membership proof between two commitments or between an event and a commitment must include *every* intermediate event in the log. In addition, because intermediate events cannot be skipped, each auditor, or client acting as an auditor, must eventually receive every event in the log. Hash chaining schemes, as such, are only feasible with low event volumes or in situations where every auditor is already receiving every event.

When membership proofs are used to investigate old events, the ability to skip nodes can lead to dramatic reductions in proof size. For example, in our prototype described in Section 5, in a log of 80 million events, our history tree can return a complete proof for any randomly chosen event in 3100 bytes. In a hash chain, where intermediate events cannot be skipped, an average of 40 million hashes would be sent.

Auditing strategies In many settings, it is possible that not every auditor will be interested in every logged event. Clients may not be interested in auditing events inserted or commitments received by other clients. One could easily imagine scenarios where a single logger is shared across many organizations, each only incentivized to audit the integrity of its own data. These organizations could run their own auditors, focusing their attention on commitments from their own clients, and only occasionally exchanging commitments with other organizations to ensure no forking has occurred. One can also imagine scenarios where independent accounting firms operate auditing systems that run against their corporate customers' log servers.

The log remains tamper-evident if clients gossip their received commitments from the logger to at least one honest auditor who uses it when demanding an incremental proof. By not requiring that every commitment be audited by every auditor, the total auditing overhead across all auditors can be proportional to the total number of events in the log—far cheaper than the number of events times the number of auditors as we might otherwise require.

$$A^v_{i,0} = \begin{cases} H(0 \,\|\, X_i) & \text{if } v \geq i \end{cases} \quad (1)$$

$$A^v_{i,r} = \begin{cases} H(1 \,\|\, A^v_{i,r-1} \,\|\, \square) & \text{if } v < i + 2^{r-1} \\ H(1 \,\|\, A^v_{i,r-1} \,\|\, A^v_{i+2^{r-1},r-1}) & \text{if } v \geq i + 2^{r-1} \end{cases} \quad (2)$$

$$C_n = A^n_{0,d} \quad (3)$$

$$A^v_{i,r} \equiv \mathrm{FH}_{i,r} \qquad \text{whenever } v \geq i + 2^r - 1 \quad (4)$$

Figure 5: Recurrence for computing hashes.

Skipping nodes offers other time-security tradeoffs. Auditors may conduct audits probabilistically, selecting only a subset of incoming commitments for auditing. If a logger were to regularly tamper with the log, its odds of remaining undetected would become vanishingly small.

3.2 Construction of the history tree

Now that we have an example of how to use a tree-based history, we will formally define its construction and semantics. A version-n history tree stores $n+1$ events, $X_0 \ldots X_n$. Hashes are computed over the history tree in a manner that permits the reconstruction of the hashes of interior nodes of older versions or *views*. We denote the hash on node $I_{i,r}$ by $A^v_{i,r}$ which is parametrized by the node's index, layer and view being computed. A version-v view on a version-n history tree reconstructs the hashes on interior nodes for a version-v history tree that only included events $X_0 \ldots X_v$. When $v = n$, the reconstructed root commitment is C_n. The hashes are computed with the recurrence defined in Figure 5.

A history tree can support arbitrary size logs by increasing the depth when the tree fills (i.e., $n = 2^d - 1$) and defining $d = \lceil \log_2(n+1) \rceil$. The new root, one level up, is created with the old tree as its left child and an empty right child where new events can be added. For simplicity in our illustrations and proofs, we assume a tree with fixed depth d.

Once a given subtree in the history tree is complete and has no more slots to add events, the hash for the root node of that subtree is *frozen* and will not change as future events are added to the log. The logger caches these frozen hashes (i.e., the hashes of frozen nodes) into $\mathrm{FH}_{i,r}$ to avoid the need to recompute them. By exploiting the frozen hash cache, the logger can recompute $A^v_{i,r}$ for any node with at most $O(d)$ operations. In a version-n tree, node $I_{i,r}$ is frozen when $n \geq i + 2^r - 1$. When inserting a new event into the log, $O(1)$ expected case and $O(d)$ worse case nodes will become frozen. (In Figure 1, node $I'_{0,1}$ is frozen. If event X_3 is added, nodes $I'_{2,1}$ and $I'_{0,2}$ will become frozen.)

Now that we have defined the history tree, we will describe the incremental proofs generated by the logger. Figure 4 abstractly illustrates a pruned tree equivalent to the proof given in Figure 3, representing an incremental proof from C_2 to C_6. Dots represent unfrozen nodes whose hashes are computed from their children. Open circles represent frozen nodes which are not included in the proof because their hashes can be recomputed from their children. Solid discs represent frozen nodes whose inclusion is necessary by being leaves or stubs. Grayed out nodes represent elided subtrees that are not included in the pruned tree. From this pruned tree and equations (1)-(4) (shown in Figure 5) we can compute $C_6 = A^6_{0,3}$ *and* a commitment from an earlier version-2 view, $A^2_{0,3}$.

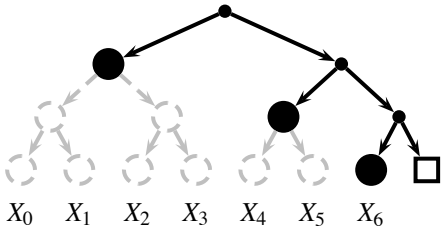


Figure 6: A proof skeleton for a version-6 history tree.

This pruned tree is incrementally built from a *proof skeleton*, seen in Figure 6—the minimum pruned tree of a version-6 tree consisting only of frozen nodes. The proof skeleton for a version-n tree consists of frozen hashes for the left siblings for the path from X_n to the root. From the included hashes and using equations (1)-(4), this proof skeleton suffices to compute $C_6 = A^6_{0,3}$.

From Figure 6 the logger incrementally builds Figure 4 by splitting frozen interior nodes. A node is split by including its children's hashes in the pruned tree instead of itself. By recursively splitting nodes on the path to a leaf, the logger can *include* that leaf in the pruned tree. In this example, we split nodes $I_{0,2}$ and $I_{2,1}$. For each commitment C_i that is to be reconstructable in an incremental proof the pruned tree P must include a path to the event X_i. The same algorithm is used to generate the membership proof for an event X_i.

Given these constraints, we can now define the five history operations in terms of the equations in Figure 5.

$H.\textsc{Add}(X) \to C_n$**.** Event is assigned the next free slot, n. C_n is computed by equations (1)-(4).

$H.\textsc{Incr.Gen}(C_i, C_j) \to P$**.** The pruned tree P is a version-j proof skeleton including a path to X_i.

$H.\textsc{Membership.Gen}(i, C_j) \to (P, X_i)$**.** The pruned tree P is a version-j proof skeleton including a path to X_i.

$P.\textsc{Incr.Vf}(C''_i, C'_j) \to \{\top, \bot\}$**.** From P apply equations (1)-(4) to compute $A^i_{0,d}$ and $A^j_{0,d}$. This can only be done if P includes a path to the leaf X_i. Return $\top$ if $C''_i = A^i_{0,d}$ and $C'_j = A^j_{0,d}$.

P.MEMBERSHIP.VF$(i, C'_j, X'_i) \rightarrow \{\top, \bot\}$. From P apply equations (1)-(4) to compute $A^j_{0,d}$. Also extract X_i from the pruned tree P, which can only be done if P includes a path to event X_i. Return $\top$ if $C'_j = A^j_{0,d}$ and $X_i = X'_i$.

Although incremental and membership proofs have different semantics, they both follow an identical tree structure and can be built and audited by a common implementation. In addition, a single pruned tree P can embed paths to several leaves to satisfy multiple auditing requests.

What is the size of a pruned tree used as a proof? The pruned tree necessary for satisfying a self-contained incremental proof between C_i and C_j or a membership proof for i in C_j requires that the pruned tree include a path to nodes X_i and X_j. This resulting pruned tree contains at most $2d$ frozen nodes, logarithmic in the size of the log.

In a real implementation, the log may have moved on to a later version, k. If the auditor requested an incremental proof between C_i and C_j, the logger would return the latest commitment C_k, and a pruned tree of at most $3d$ nodes, based around a version-k tree including paths to X_i and X_j. More typically, we expect auditors will request an incremental proof between a commitment C_i and the latest commitment. The logger can reply with the latest commitment C_k and pruned tree of at most $2d$ nodes that included a path to X_i.

The frozen hash cache In our description of the history tree, we described the *full representation* when we stated that the logger stores frozen hashes for all frozen interior nodes in the history tree. This cache is redundant whenever a node's hash can be recomputed from its children. We expect that logger implementations, which build pruned trees for audits and queries, will maintain and use the cache to improve efficiency.

When generating membership proofs, incremental proofs, and query lookup results, there is no need for the resulting pruned tree to include redundant hashes on interior nodes when they can be recomputed from their children. We assume that pruned trees used as proofs will use this *minimum representation*, containing frozen hashes only for stubs, to reduce communication costs.

Can overheads be reduced by exploiting redundancy between proofs? If an auditor is in regular communication with the logger, demanding incremental proofs between the previously seen commitment and the latest commitment, there is redundancy between the pruned subtrees on successive queries.

If an auditor previously requested an incremental proof between C_i and C_j and later requests an incremental proof P between C_j and C_n, the two proofs will share hashes on the path to leaf X_j. The logger may send a *partial proof* that omits these common hashes, and only contains the expected $O(\log_2(n-j))$ frozen hashes that are not shared between the paths to X_j and X_n. This devolves to $O(1)$ if a proof is requested after every insertion. The auditor need only cache d frozen hashes to make this work.

Tree history time-stamping service Our history tree can be adapted to implement a round-based time-stamping service. After every round, the logger publishes the last commitment in public medium such as a newspaper. Let C_i be the commitment from the prior round and C_k be the commitment of the round a client requests that its document X_j be timestamped. A client can request a pruned tree including a path to leaves X_i, X_j, X_k. The pruned tree can be verified against the published commitments to prove that X_j was submitted in the round and its order within that round, without the cooperation of the logger.

If a separate history tree is built for each round, our history tree is equivalent to the threaded authentication tree proposed by Buldas et al. [10] for time-stamping systems.

3.3 Storing the log on secondary storage

Our history tree offers a curious property: it can be easily mapped onto write-once append-only storage. Once nodes become frozen, they become immutable, and are thus safe to output. This ordering is predetermined, starting with (X_0), $(X_1, I_{0,1})$, (X_2), $(X_3, I_{2,1}, I_{0,2})$, $(X_4)\ldots$. Parentheses denote the nodes written by each ADD transaction. If nodes within each group are further ordered by their layer in the tree, this order is simply a post-order traversal of the binary tree. Data written in this linear fashion will minimize disk seek overhead, improving the disk's write performance. Given this layout, and assuming all events are the same size on disk, converting from an $(index, layer)$ to the byte index used to store that node takes $O(\log n)$ arithmetic operations, permitting efficient direct access.

In order to handle variable-length events, event data can be stored in a separate write-once append-only *value store*, while the leaves of the history tree contain offsets into the value store where the event contents may be found. Decoupling the history tree from the value store also allows many choices for how events are stored, such as databases, compressed files, or standard flat formats.

3.4 Comparing to other systems

In this section, we evaluate the time and space tradeoffs between our history tree and earlier hash chain and skip list structures. In all three designs, membership proofs have the same structure and size as incremental proofs, and proofs are generated in time proportional to their size.

Maniatis and Baker [43] present a tamper-evident log using a deterministic variant of a skip list [53]. The skip list history is like a hash-chain incorporating extra skip links that hop over many nodes, allowing for logarithmic lookups.

	Hash chain	Skip list	History tree
ADD Time	$O(1)$	$O(1)$	$O(\log_2 n)$
INCR.GEN proof size to C_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
MEMBERSHIP.GEN proof size for X_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
Cache size	-	$O(\log_2 n)$	$O(\log_2 n)$
INCR.GEN partial proof size	-	$O(n-j)$	$O(\log_2(n-j))$
MEMBERSHIP.GEN partial proof size	-	$O(\log_2(n-i))$	$O(\log_2(n-i))$

Table 1: We characterize the time to add an event to the log and the size of full and partial proofs generated in terms of n, the number of events in the log. For partial proofs audits, j denotes the number of events in the log at the time of the last audit and i denotes the index of the event being membership-audited.

In Table 1 we compare the three designs. All three designs have $O(1)$ storage per event and $O(1)$ commitment size. For skip list histories and tree histories, which support partial proofs (described in Section 3.2), we present the cache size and the expected proof sizes in terms of the number of events in the log, n, and the index, j, of the prior contact with the logger or the index i of the event being looked up. Our tree-based history strictly dominates both hash chains and skip lists in proof generation time and proof sizes, particularly when individual clients and auditors only audit a subset of the commitments or when partial proofs are used.

Canonical representation A hash chain history and our history tree have a canonical representation of both the history and of proofs within the history. In particular, from a given commitment C_n, there exists one unique path to each event X_i. When there are multiple paths auditing is more complex because the alternative paths must be checked for consistency with one another, both within a single history, and between the stream of histories $C_i, C_{i+1}, \ldots$ committed by the logger. Extra paths may improve the efficiency of looking up past events, such as in a skip list, or offer more functionality [17], but cannot be trusted by auditors and must be checked.

Maniatis and Baker [43] claim to support logarithmic-sized proofs, however they suffer from this multi-path problem. To verify internal consistency, an auditor with no prior contact with the logger must receive every event in the log in every incremental or membership proof.

Efficiency improves for auditors in regular contact with the logger that use partial proofs and cache $O(\log_2 n)$ state between incremental audits. If an auditor has previously verified the logger's internal consistency up to C_j, the auditor will be able to verify the logger's internal consistency up to a future commitment C_n with the receipt of events $X_{j+1} \ldots X_n$ Once an auditor knows that the skip list is internally consistent the links that allow for logarithmic lookups can be trusted and subsequent membership proofs on old events will run in $O(\log_2 n)$ time. Skip list histories were designed to function in this mode, with each auditor eventually receiving every event in the log.

Auditing is required Hash chains and skip lists only offer a complexity advantage over the history tree when adding new events, but this advantage is fleeting. If the logger knows that a given commitment will never be audited, it is free to tamper with the events fixed by that commitment, and the log is no longer provably tamper evident. Every commitment returned by the logger must have a non-zero chance of being audited and any evaluation of tamper-evident logging must include the costs of this unavoidable auditing. With multiple auditors, auditing overhead is further multiplied. After inserting an event, hash chains and skip lists suffer an $O(n-j)$ disadvantage the moment they do incremental audits between the returned commitment and prior commitments. They cannot reduce this overhead by, for example, only auditing a random subset of commitments.

Even if the threat model is weakened from our always-untrusted logger to the forward-integrity threat model (See Section 2.4), hash chains and skip lists are less efficient than the history tree. Clients can forgo auditing just-added events, but are still required to do incremental audits to prior commitments, which are expensive with hash chains or skip lists.

4 Merkle aggregation

Our history tree permits $O(\log_2 n)$ access to arbitrary events, given their index. In this section, we extend our history tree to support efficient, tamper-evident content searches through a feature we call *Merkle aggregation*, which encodes auxiliary information into the history tree. Merkle aggregation permits the logger to perform authorized purges of the log while detecting unauthorized deletions, a feature we call *safe deletion*.

As an example, imagine that a client flags certain events in the log as "important" when it stores them. In the history tree, the logger propagates these flags to interior nodes, setting the flag whenever either child is flagged. To ensure that the tagged history is tamper-evident, this flag can be incorporated into the hash label of a node and checked during auditing. As clients are assumed to be trusted when inserting into the log, we assume clients will properly annotate their events. Membership auditing will detect if the logger incorrectly stored a leaf with the wrong flag or improperly propagated the flag. Incremental audits would detect tampering if any frozen

node had its flag altered. Now, when an auditor requests a list of only flagged events, the logger can generate that list along with a proof that the list is complete. If there are relatively few "important" events, the query results can skip over large chunks of the history.

To generate a proof that the list of flagged events is complete, the logger traverses the full history tree H, pruning any subtrees without the flag set, and returns a pruned tree P containing only the visited nodes. The auditor can ensure that no flagged nodes were omitted in P by performing its own recursive traversal on P and verifying that every stub is unflagged.

Figure 7 shows the pruned tree for a query against a version-5 history with events X_2 and X_5 flagged. Interior nodes in the path from X_2 and X_5 to the root will also be flagged. For subtrees containing no matching events, such as the parent of X_0 and X_1, we only need to retain the root of the subtree to vouch that its children are unflagged.

4.1 General attributes

Boolean flags are only one way we may flag log events for later queries. Rather than enumerate every possible variation, we abstract an aggregation strategy over attributes into a 3-tuple, $(\tau, \oplus, \Gamma)$. τ represents the type of attribute or attributes that an event has. $\oplus$ is a deterministic function used to compute the attributes on an interior node in the history tree by *aggregating* the attributes of the node's children. Γ is a deterministic function that maps an event to its attributes. In our example of client-flagged events, the aggregation strategy is $(\tau := \text{BOOL}, \oplus := \vee, \Gamma(x) := x.isFlagged)$.

For example, in a banking application, an attribute could be the dollar value of a transaction, aggregated with the MAX function, permitting queries to find all transactions over a particular dollar value and detect if the logger tampers with the results. This corresponds to $(\tau := \text{INT}, \oplus := \text{MAX}, \Gamma(x) := x.value)$. Or, consider events having internal timestamps, generated by the client, arriving at the logger out of order. If we attribute each node in the tree with the earliest and latest timestamp found among its children, we can now query the logger for all nodes within a given time range, regardless of the order of event arrival.

There are at least three different ways to implement keyword searching across logs using Merkle aggregation. If the number of keywords is fixed in advance, then the attribute τ for events can be a bit-vector or sparse bit-vector combined with $\oplus := \vee$. If the number of keywords is unknown, but likely to be small, τ can be a sorted list of keywords, with $\oplus := \cup$ (set union). If the number of keywords is unknown and potentially unbounded, then a Bloom filter [8] may be used to represent them, with τ being a bit-vector and $\oplus := \vee$. Of course, the Bloom filter would then have the potential of returning false positives to a query, but there would be no false negatives.

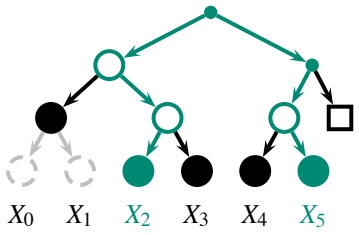


Figure 7: Demonstration of Merkle aggregation with some events flagged as important (highlighted). Frozen nodes that would be included in a query are represented as solid discs.

Merkle aggregation is extremely flexible because Γ can be *any* deterministic computable function. However, once a log has been created, $(\tau, \oplus, \Gamma)$ are fixed for that log, and the set of queries that can be made is restricted based on the aggregation strategy chosen. In Section 5 we describe how we were able to apply these concepts to the metadata used in Syslog logs.

4.2 Formal description

To make attributes tamper-evident in history trees, we modify the computation of hashes over the tree to include them. Each node now has a hash label denoted by $A^v_{i,r}.H$ and an annotation denoted by $A^v_{i,r}.A$ for storing attributes. Together these form the node data that is attached to each node in the history tree. Note that the hash label of node, $A^v_{i,r}.H$, does *not* fix its own attributes, $A^v_{i,r}.A$. Instead, we define a *subtree authenticator* $A^v_{i,r}.* = H(A^v_{i,r}.H \parallel A^v_{i,r}.A)$ that fixes the attributes and hash of a node, and recursively fixes every hash and attribute in its subtree. Frozen hashes $\text{FH}_{i,r}.A$ and $\text{FH}_{i,r}.H$ and $\text{FH}_{i,r}.*$ are defined analogously to the non-Merkle-aggregation case.

We could have defined this recursion in several different ways. This representation allows us to elide unwanted subtrees with a small stub, containing one hash and one set of attributes, while exposing the attributes in a way that makes it possible to locally detect if the attributes were improperly aggregated.

Our new mechanism for computing hash and aggregates for a node is given in equations (5)-(10) in Figure 8. There is a strong correspondence between this recurrence and the previous one in Figure 5. Equations (6) and (7) extract the hash and attributes of an event, analogous to equation (1). Equation (9) handles aggregation of attributes between a node and its children. Equation (8) computes the hash of a node in terms of the subtree authenticators of its children.

INCR.GEN and MEMBERSHIP.GEN operate the same as with an ordinary history tree, except that wherever a frozen hash was included in the proof ($\text{FH}_{i,r}$), we now include both the hash of the node, $\text{FH}_{i,r}.H$, and its attributes $\text{FH}_{i,r}.A$. Both are required for recomputing $A^v_{i,r}.A$ and $A^v_{i,r}.H$ for the parent node. ADD, INCR.VF,

$$A^v_{i,r}.* = H(A^v_{i,r}.H \,\|\, A^v_{i,r}.A) \tag{5}$$

$$A^v_{i,0}.H = \begin{cases} H(0 \,\|\, X_i) & \text{if } v \geq i \end{cases} \tag{6}$$

$$A^v_{i,0}.A = \begin{cases} \Gamma(X_i) & \text{if } v \geq i \end{cases} \tag{7}$$

$$A^v_{i,r}.H = \begin{cases} H(1 \,\|\, A^v_{i,r-1}.* \,\|\, \square) & \text{if } v < i + 2^{r-1} \\ H(1 \,\|\, A^v_{i,r-1}.* \,\|\, A^v_{i+2^{r-1},r-1}.*) & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{8}$$

$$A^v_{i,r}.A = \begin{cases} A^v_{i,r-1}.A & \text{if } v < i + 2^{r-1} \\ A^v_{i,r-1}.A \oplus A^v_{i+2^{r-1},r-1}.A & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{9}$$

$$C_n = A^n_{0,d}.* \tag{10}$$

Figure 8: Hash computations for Merkle aggregation

and MEMBERSHIP.VF are the same as before except for using the equations (5)-(10) for computing hashes and propagating attributes. Merkle aggregation inflates the storage and proof sizes by a factor of $(A+B)/A$ where A is the size of a hash and B is the size of the attributes.

4.2.1 Queries over attributes

In Merkle aggregation queries, we permit query results to contain false positives, i.e., events that do not match the query Q. Extra false positive events in the result only impact performance, not correctness, as they may be filtered by the auditor. We forbid false negatives; every event matching Q will be included in the result.

Unfortunately, Merkle aggregation queries can only match attributes, not events. Consequently, we must conservatively transform a query Q over events into a predicate Q^Γ over attributes and require that it be *stable*, with the following properties: If Q matches an event then Q^Γ matches the attributes of that event (i.e., $\forall_x\ Q(x) \Rightarrow Q^\Gamma(\Gamma(x))$). Furthermore, if Q^Γ is true for either child of a node, it must be true for the node itself (i.e., $\forall_{x,y}\ Q^\Gamma(x) \vee Q^\Gamma(y) \Rightarrow Q^\Gamma(x \oplus y)$ and $\forall_x\ Q^\Gamma(x) \vee Q^\Gamma(\square) \Rightarrow Q^\Gamma(x \oplus \square)$).

Stable predicates can falsely match nodes or events for two reasons: events' attributes may match Q^Γ without the events matching Q, or nodes may occur where $(Q^\Gamma(x) \vee Q^\Gamma(y))$ is false, but $Q^\Gamma(x \oplus y)$ is true. We call a predicate Q *exact* if there can be no false matches. This occurs when $Q(x) \Leftrightarrow Q^\Gamma(\Gamma(x))$ and $Q^\Gamma(x) \vee Q^\Gamma(y) \Leftrightarrow Q^\Gamma(x \oplus y)$. Exact queries are more efficient because a query result does not include falsely matching events and the corresponding pruned tree proving the correctness of the query result does not require extra nodes.

Given these properties, we can now define the additional operations for performing authenticated queries on the log for events matching a predicate Q^Γ.

$H.\text{QUERY}(C_j, Q^\Gamma) \to P$ Given a predicate Q^Γ over attributes τ, returns a pruned tree where every elided subtrees does not match Q^Γ.

$P.\text{QUERY.VF}(C'_j, Q^\Gamma) \to \{\top, \bot\}$ Checks the pruned tree P and returns $\top$ if every stub in P does not match Q^Γ and the reconstructed commitment C_j is the same as C'_j.

Building a pruned tree containing all events matching a predicate Q^Γ is similar to building the pruned trees for membership or incremental auditing. The logger starts with a proof skeleton then recursively traverses it, splitting interior nodes when $Q^\Gamma(FH_{i,r}.A)$ is true. Because the predicate Q^Γ is stable, no event in any elided subtree can match the predicate. If there are t events matching the predicate Q^Γ, the pruned tree is of size at most $O((1+t)\log_2 n)$ (i.e., t leaves with $\log_2 n$ interior tree nodes on the paths to the root).

To verify that P includes all events matching Q^Γ, the auditor does a recursive traversal over P. If the auditor finds an interior stub where $Q^\Gamma(FH_{i,r}.A)$ is true, the verification fails because the auditor found a node that was supposed to have been split. (Unfrozen nodes will always be split as they compose the proof skeleton and only occur on the path from X_j to the root.) The auditor must also verify that pruned tree P commits the same events as the commitment C'_j by reconstructing the root commitment C_j using the equations (5)-(10) and checking that $C_j = C'_j$.

As with an ordinary history tree, a Merkle aggregating tree requires auditing for tamper-detection. If an event is never audited, then there is no guarantee that its attributes have been properly included. Also, a dishonest logger or client could deliberately insert false log entries whose attributes are aggregated up the tree to the root, causing garbage results to be included in queries. Even so, if Q is stable, a malicious logger cannot hide matching events from query results without detection.

4.3 Applications

Safe deletion Merkle aggregation can be used for expiring old and obsolete events that do not satisfy some predicate and prove that no other events were deleted inappropriately. While Merkle aggregation queries prove that no matching event is excluded from a query result, safe deletion requires the contrapositive: proving to an auditor that each purged event was legitimately purged because it did not match the predicate.

Let $Q(x)$ be a stable query that is true for all events that the logger must keep. Let $Q^\Gamma(x)$ be the corresponding predicate over attributes. The logger stores a pruned tree that includes all nodes and leaf events where $Q^\Gamma(x)$ is true. The remaining nodes may be elided and replaced with stubs. When a logger cannot generate a path to a previously deleted event X_i, it instead supplies a pruned tree that includes a path to an ancestor node A of X_i where $Q^\Gamma(A)$ is false. Because Q is stable, if $Q^\Gamma(A)$ is false, then $Q^\Gamma(\Gamma(X_i))$ and $Q(X_i)$ must also be false.

Safe deletion and auditing policies must take into account that if a subtree containing events $X_i \ldots X_j$ is purged, the logger is unable to generate incremental or membership proofs involving commitments $C_i \ldots C_j$. The auditing policy must require that any audits using those commitments be performed before the corresponding events are deleted, which may be as simple as requiring that clients periodically request an incremental proof to a later or long-lived commitment.

Safe deletion will not save space when using the append-only storage described in Section 3.3. However, if data-destruction policies require destroying a subset of events in the log, safe deletion may be used to prove that no unauthorized log events were destroyed.

"Private" search Merkle aggregation enables a weak variant of private information retrieval [14], permitting clients to have privacy for the specific contents of their events. To aggregate the attributes of an event, the logger only needs the attributes of an event, $\Gamma(X_i)$, not the event itself. To verify that aggregation is done correctly also only requires the attributes of an event. If clients encrypt their events and digitally sign their public attributes, auditors may verify that aggregation is done correctly while clients preserve their event privacy from the logger and other clients and auditors.

Bloom filters, in addition to providing a compact and approximate way to represent the presence or absence of a large number of keywords, can also enable private indexing (see, e.g., Goh [23]). The logger has no idea what the individual keywords are within the Bloom filter; many keywords could map to the same bit. This allows for private keywords that are still protected by the integrity mechanisms of the tree.

5 Syslog prototype implementation

Syslog is the standard Unix-based logging system [38], storing events with many attributes. To demonstrate the effectiveness of our history tree, we built an implementation capable of storing and searching syslog events. Using events from syslog traces, captured from our departmental servers, we evaluated the storage and performance costs of tamper-evident logging and secure deletion.

Each syslog event includes a timestamp, the host generating the event, one of 24 *facilities* or subsystem that generated the event, one of 8 logging *levels*, and the *message*. Most events also include a *tag* indicating the program generating the event. Solutions for authentication, management, and reliable delivery of syslog events over the network have already been proposed [48] and are in the process of being standardized [32], but none of this work addresses the logging semantics that we wish to provide.

Our prototype implementation was written in a hybrid of Python 2.5.2 and C++ and was benchmarked on an Intel Core 2 Duo 2.4GHz CPU with 4GB of RAM in 64-bit mode under Linux. Our present implementation is single-threaded, so the second CPU core is underutilized. Our implementation uses SHA-1 hashes and 1024-bit DSA signatures, borrowed from the OpenSSL library.

In our implementation, we use the array-based post-order traversal representation discussed in Section 3.3. The value store and history tree are stored in separate write-once append-only files and mapped into memory. Nodes in the history tree use a fixed number of bytes, permitting direct access. Generating membership and incremental proofs requires RAM proportional to the size of the proof, which is logarithmic in the number of events in the log. Merkle aggregation query result sizes are presently limited to those which can fit in RAM, approximately 4 million events.

The storage overheads of our tamper-evident history tree are modest. Our prototype stores five attributes for each event. Tags and host names are encoded as 2-of-32 bit Bloom filters. Facilities and hosts are encoded as bit-vectors. To permit range queries to find every event in a particular range of time, an interval is used to encode the message timestamp. All together, there are twenty bytes of attributes and twenty bytes for a SHA-1 hash for each node in the history tree. Leaves have an additional twelve bytes to store the offset and length of the event contents in the value store.

We ran a number of simulations of our prototype to determine the processing time and space overheads of the history tree. To this end, we collected a trace of four million events from thirteen of our departmental server hosts over 106 hours. We observed 9 facilities, 6 levels, and 52 distinct tags. 88.1% of the events are from the mail server and 11.5% are from 98,743 failed ssh connection attempts. Only .393% of the log lines are from other sources. In testing our history tree, we replay this trace 20 times to insert 80 million events. Our syslog trace, after the replay, occupies 14.0 GB, while the history tree adds an additional 13.6 GB.

5.1 Performance of the logger

The logger is the only centralized host in our design and may be a bottleneck. The performance of a real world logger will depend on the auditing policy and relative frequency between inserting events and requesting audits. Rather than summarize the performance of the logger for one particular auditing policy, we benchmark the costs of the various tasks performed by the logger.

Our captured syslog traces averaged only ten events per second. Our prototype can insert events at a rate of 1,750 events per second, including DSA signature generation. Inserting an event requires four steps, shown in Table 2, with the final step, signing the resulting commitment, responsible for most of the processing time. Throughput

Step	Task	% of CPU	Rate (events/sec)
A	Parse syslog message	2.4%	81,000
B	Insert event into log	2.6%	66,000
C	Generate commitment	11.8%	15,000
D	Sign commitment	83.3%	2,100
	Membership proofs (with locality)	-	8,600
	Membership proofs (no locality)	-	32

Table 2: Performance of the logger in each of the four steps required to insert an event and sign the resulting commitment and in generating membership proofs. Rates are given assuming nothing other than the specified step is being performed.

would increase to 10,500 events per second if the DSA signatures were computed elsewhere (e.g., leveraging multiple CPU cores). (Section 6 discusses scalability in more detail.) This corresponds to 1.9MB/sec of uncompressed syslog data (1.1 TB per week).

We also measured the rate at which our prototype can generate membership and incremental proofs. The size of an incremental proof between two commitments depends upon the distance between the two commitments. As the distance varies from around two to two million events, the size of a self-contained proof varies from 1200 bytes to 2500 bytes. The speed for generating these proofs varies from 10,500 proofs/sec to 18,000 proofs/sec, with shorter distances having smaller proof sizes and faster performance than longer distances. For both incremental and membership proofs, compressing by gzip [18] halves the size of the proofs, but also halves the rate at which proofs can be generated.

After inserting 80 million events into the history tree, the history tree and value store require 27 GB, several times larger than our test machine's RAM capacity. Table 2 presents our results for two membership auditing scenarios. In our first scenario we requested membership proofs for random events chosen among the most recent 5 million events inserted. Our prototype generated 8,600 self-contained membership proofs per second, averaging 2,400 bytes each. In this high-locality scenario, the most recent 5 million events were already sitting in RAM. Our second scenario examined the situation when audit requests had low locality by requesting membership proofs for random events anywhere in the log. The logger's performance was limited to our disk's seek latency. Proof size averaged 3,100 bytes and performance degraded to 32 membership proofs per second. (We discuss how this might be overcome in Section 6.2.)

To test the scalability of the history tree, we benchmarked insert performance and auditing performance on our original 4 million event syslog event trace, without replication, and the 80 million event trace after 20x replication. Event insertion and incremental auditing are roughly 10% slower on the larger log.

5.2 Performance of auditors and clients

The history tree places few demands upon auditors or clients. Auditors and clients must verify the logger's commitment signatures and must verify the correctness of pruned tree replies to auditing requests. Our machine can verify 1,900 DSA-1024 signatures per second. Our current tree parser is written in Python and is rather slow. It can only parse 480 pruned trees per second. Once the pruned tree has been parsed, our machine can verify 9,000 incremental or membership proofs per second. Presently, one auditor cannot verify proofs as fast as the logger can generate them, but auditors can clearly operate independently of one another, in parallel, allowing for exceptional scaling, if desired.

5.3 Merkle aggregation results

In this subsection, we describe the benefits of Merkle aggregation in generating query results and in safe deletion. In our experiments, due to limitations of our implementation in generating large pruned trees, our Merkle aggregation experiments used the smaller four million event log.

We used 86 different predicates to investigate the benefits of safe deletion and the overheads of Merkle aggregation queries. We used 52 predicates, each matching one tag, 13 predicates, each matching one host, 9 predicates, each matching one facility, 6 predicates, one matching each level, and 6 predicates, each matching the k highest logging levels.

The predicates matching tags and hosts use Bloom filters, are *inexact*, and may have false positives. This causes 34 of the 65 Bloom filter query results to include more nodes than our "worst case" expectation for exact predicates. By using larger Bloom filters, we reduce the chances of spurious matches. When a 4-of-64 Bloom filter is used for tags and hostnames, pruned trees resulting from search queries average 15% fewer nodes, at the cost of an extra 64 bits of attributes for each node in the history tree. In a real implementation, the exact parameters of the Bloom filter would best be tuned to match a sample of the events being logged.

Merkle aggregation and safe deletion Safe deletion allows the purging of unwanted events from the log. Auditors define a stable predicate over the attributes of events indicating which events must be kept, and the logger keeps a pruned tree of only those matching events. In our first test, we simulated the deletion of all events except those from a particular host. The pruned tree was generated in 14 seconds, containing 1.92% of the events in the full log and serialized to 2.29% of the size of the full tree. Although 98.08% of the events were purged, the logger was only able to purge 95.1% of the nodes in the

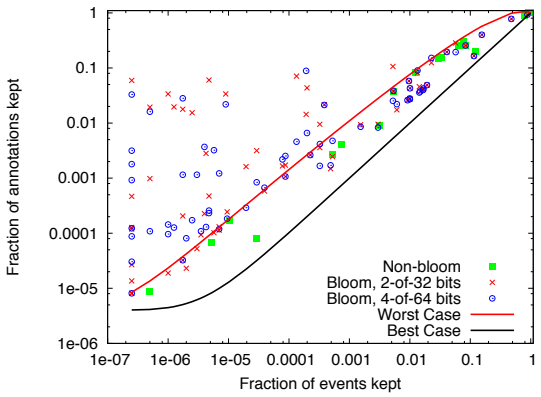


Figure 9: Safe deletion overhead. For a variety of queries, we plot the fraction of hashes and attributes kept after deletion versus the fraction of events kept.

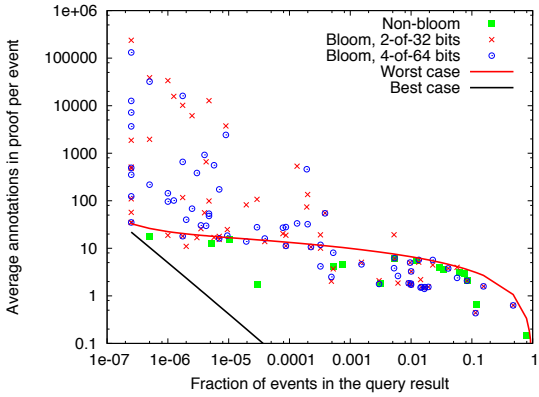


Figure 10: Query overhead per event. We plot the ratio between the number of hashes and matching events in the result of each query versus the fraction of events matching the query.

history tree because the logger must keep the hash label and attributes for the root nodes of elided subtrees.

When measuring the size of a pruned history tree generated by safe deletion, we assume the logger caches hashes and attributes for all interior nodes in order to be able to quickly generate proofs. For each predicate, we measure the *kept ratio*, the number of interior node or stubs in a pruned tree of all nodes matching the predicate divided by the number of interior nodes in the full history tree. In Figure 9 for each predicate we plot the kept ratio versus the fraction of events matching the predicate. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. Our Bloom-filter queries do worse than the "worst-case" bound because Bloom filter matches are inexact and will thus trigger false positive matches on interior nodes, forcing them to be kept in the resulting pruned tree. Although many Bloom filters did far worse than the "worst-case," among the Bloom filters that matched fewer than 1% of the events in the log, the logger is still able to purge over 90% of the nodes in the history tree and often did much better than that.

Merkle aggregation and authenticated query results In our second test, we examine the overheads for Merkle aggregation query lookup results. When the logger generates the results to a query, the resulting pruned tree will contain both matching events and history tree overhead, in the form of hashes and attributes for any stubs. For each predicate, we measure the *query overhead ratio*—the number of stubs and interior nodes in a pruned tree divided by the number of events in the pruned tree. In Figure 10 we plot the query overhead ratio versus the fraction of events matching the query for each of our 86 predicates. This plot shows, for each event matching a predicate, proportionally how much extra overhead is incurred, per event, for authentication information. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. With exact predicates, the overhead of authenticated query results is very modest, and again, inexact Bloom filter queries will sometimes do worse than the "worst case."

6 Scaling a tamper-evident log

In this section, we discuss techniques to improve the insert throughput of the history tree by using concurrency, and to improve the auditing throughput with replication. We also discuss a technique to amortize the overhead of a digital signature over several events.

6.1 Faster inserts via concurrency

Our tamper-evident log offers many opportunities to leverage concurrency to increase throughput. Perhaps the simplest approach is to offload signature generation. From Table 2, signatures account for over 80% of the runtime cost of an insert. Signatures are not included in any other hashes and there are no interdependencies between signature computations. Furthermore, signing a commitment does not require knowing anything other than the root commitment of the history tree. Consequently, it's easy to offload signature computations onto additional CPU cores, additional hosts, or hardware crypto accelerators to improve throughput.

It is possible for a logger to also generate commitments concurrently. If we examine Table 2, parsing and inserting events in the log is about two times faster than generating commitments. Like signatures, commitments have no interdependencies on one other; they depend only on the history tree contents. As soon as event X_j is inserted into the tree and $O(1)$ frozen hashes are computed and stored,

a new event may be immediately logged. Computing the commitment C_j only requires read-only access to the history tree, allowing it to be computed concurrently by another CPU core without interfering with subsequent events. By using shared memory and taking advantage of the append-only write-once semantics of the history tree, we would expect concurrency overhead to be low.

We have experimentally verified the maximum rate at which our prototype implementation, described in Section 5, can insert syslog events into the log at 38,000 events per second using only one CPU core on commodity hardware. This is the maximum throughput our hardware could potentially support. In this mode we assume that digital signatures, commitment generation, and audit requests are delegated to additional CPU cores or hosts. With multiple hosts, each host must build a replica of the history tree which can be done at least as fast as our maximum insert rate of 38,000 events per second. Additional CPU cores on these hosts can then be used for generating commitments or handling audit requests.

For some applications, 38,000 events per second may still not be fast enough. Scaling beyond this would require fragmenting the event insertion and storage tasks across multiple logs. To break interdependencies between them, the fundamental history tree data structure we presently use would need to evolve, perhaps into disjoint logs that occasionally entangle with one another as in timeline entanglement [43]. Designing and evaluating such a structure is future work.

6.2 Logs larger than RAM

For exceptionally large audits or queries, where the working set size does not fit into RAM, we observed that throughput was limited to disk seek latency. Similar issues occur in any database query system that uses secondary storage, and the same software and hardware techniques used by databases to speed up queries may be used, including faster or higher throughput storage systems or partitioning the data and storing it in-memory across a cluster of machines. A single large query can then be issued to the cluster node managing each sub-tree. The results would then be merged before transmitting the results to the auditor. Because each sub-tree would fit in its host's RAM, sub-queries would run quickly.

6.3 Signing batches of events

When large computer clusters are unavailable and the performance cost of DSA signatures is the limiting factor in the logger's throughput, we may improve performance of the logger by allowing multiple updates to be handled with one signature computation.

Normally, when a client requests an event X to be inserted, the logger assigns it an index i, generates the commitment C_i, signs it, and returns the result. If the logger has insufficient CPU to sign every commitment, the logger could instead delay returning C_i until it has a signature for some later commitment C_j ($j \geq i$). This later signed commitment could then be sent to the client expecting an earlier one. To ensure that the event X_i in the log committed by C_j was X, the client may request a membership proof from commitment C_j to event i and verify that $X_i = X$. This is safe due to the tamper-evidence of our structure. If the logger were ever to later sign a C_i inconsistent with C_j, it would fail an incremental proof.

In our prototype, inserting events into the log is twenty times faster than generating and signing commitments. The logger may amortize the costs of generating a signed commitment over many inserted events. The number of events per signed commitment could vary dynamically with the load on the logger. Under light load, the logger could sign every commitment and insert 1,750 events per second. With increasing load, the logger might sign one in every 16 commitments to obtain an estimated insert rate of 17,000 events per second. Clients will still receive signed commitments within a fraction of a second, but several clients can now receive the same commitment. Note that this analysis only considers the maximum insert rate for the log and does not include the costs of replying to audits. The overall performance improvements depend on how often clients request incremental and membership proofs.

7 Related work

There has been recent interest in creating append-only databases for regulatory compliance. These databases permit the ability to access old versions and trace tampering [51]. A variety of different data structures are used, including a B-tree [64] and a full text index [47]. The security of these systems depends on a write-once semantics of the underlying storage that cannot be independently verified by a remote auditor.

Forward-secure digital signature schemes [3] or stream authentication [21] can be used for signing commitments in our scheme or any other logging scheme. Entries in the log may be encrypted by clients for privacy. Kelsey and Schneier [57] have the logger encrypt entries with a key destroyed after use, preventing an attacker from reading past log entries. A hash function is iterated to generate the encryption keys. The initial hash is sent to a trusted auditor so that it may decrypt events. Logcrypt [29] extends this to public key cryptography.

Ma and Tsudik [41] consider tamper-evident logs built using forward-secure sequential aggregating signature schemes [39, 40]. Their design is round-based. Within each round, the logger evolves its signature, combining a new event with the existing signature to generate a new signature, and also evolves the authentication key. At the end of a round, the final signature can authenticate any event inserted.

Davis et. al. [17] permits keyword searching in a log by trusting the logger to build parallel hash chains for each keyword. Techniques have also been designed for keyword searching encrypted logs [60, 61]. A tamper-evident store for voting machines has been proposed, based on append-only signatures [33], but the signature sizes grow with the number of signed messages [6].

Many timestamping services have been proposed in the literature. Haber and Stornetta [27] introduce a time-stamping service based on hash chains, which influenced the design of Surety, a commercial timestamping service that publishes their head commitment in a newspaper once a week. Chronos is a digital timestamping service inspired by a skip list, but with a hashing structure similar to our history tree [7]. This and other timestamping designs [9, 10] are round-based. In each round, the logger collects a set of events and stores the events within that round in a tree, skip list, or DAG. At the end of the round the logger publicly broadcasts (e.g., in a newspaper) the commitment for that round. Clients then obtain a logarithmically-sized, tamper-evident proof that their events are stored within that round and are consistent with the published commitment. Efficient algorithms have been constructed for outputting time stamp authentication information for successive events within a round in a streaming fashion, with minimal storage on the server [37]. Unlike these systems, our history tree allows events to be added to the log, commitments generated, and audits to be performed at any time.

Maniatis and Baker [43] introduced the idea of *timeline entanglement*, where every participant in a distributed system maintains a log. Every time a message is received, it is added to the log, and every message transmitted contains the hash of the log head. This process spreads commitments throughout the network, making it harder for malicious nodes to diverge from the canonical timeline without there being evidence somewhere that could be used in an audit to detect tampering. Auditorium [55] uses this property to create a shared "bulletin board" that can detect tampering even when $N-1$ systems are faulty.

Secure aggregation has been investigated as a distributed protocol in sensor networks for computing sums, medians, and other aggregate values when the host doing the aggregation is not trusted. Techniques include trading off approximate results in return for sublinear communication complexity [12], or using MAC codes to detect one-hop errors in computing aggregates [30]. Other aggregation protocols have been based around hash tree structures similar to the ones we developed for Merkle aggregation. These structures combine aggregation and cryptographic hashing, and include distributed sensor-network aggregation protocols for computing authenticated sums [13] and generic aggregation [45]. The sensor network aggregation protocols interactively generate a secure aggregate of a set of measurements. In Merkle aggregation, we use intermediate aggregates as a tool for performing efficient queries. Also, our Merkle aggregation construction is more efficient than these designs, requiring fewer cryptographic hashes to verify an event.

8 Conclusions

In this work we have shown that regular and continous auditing is a critical operation for any tamper-evident log system, for without auditing, clients cannot detect if a Byzantine logger is misbehaving by not logging events, removing unaudited events, or forking the log. From this requirement we have developed a new tamper-evident log design, based on a new Merkle tree data structure that permits a logger to produce concise proofs of its correct behavior. Our system eliminates any need to trust the logger, instead allowing clients and auditors of the logger to efficiently verify its correct behavior with only a constant amount of local state. By sharing commitments among clients and auditors, our design is resistant even to sophisticated forking or rollback attacks, even in cases where a client might change its mind and try to repudiate events that it had logged earlier.

We also proposed Merkle aggregation, a flexible mechanism for encoding auxiliary attributes into a Merkle tree that allows these attributes to be aggregated from the leaves up to the root of the tree in a verifiable fashion. This technique permits a wide range of efficient, tamper-evident queries, as well as enabling verifiable, safe deletion of "expired" events from the log.

Our prototype implementation supports thousands of events per second, and can easily scale to very large logs. We also demonstrated the effectiveness of Bloom filters to enable a broad range of queries. By virtue of its concise proofs and scalable design, our techniques can be applied in a variety of domains where high volumes of logged events might otherwise preclude the use of tamper-evident logs.

Acknowledgements

The authors gratefully acknowledge Farinaz Koushanfar, Daniel Sandler, and Moshe Vardi for many helpful comments and discussions on this project. The authors also thank the anonymous referees and Micah Sherr, our shepherd, for their assistance. This work was supported, in part, by NSF grants CNS-0524211 and CNS-0509297.

References

[1] ACCORSI, R., AND HOHL, A. Delegating secure logging in pervasive computing systems. In *Security in Pervasive Computing* (York, UK, Apr. 2006), pp. 58–72.

[2] ANAGNOSTOPOULOS, A., GOODRICH, M. T., AND TAMASSIA, R. Persistent authenticated dictionaries and their applications. In *International Conference on*

Information Security (ISC) (Seoul, Korea, Dec. 2001), pp. 379–393.

[3] BELLARE, M., AND MINER, S. K. A forward-secure digital signature scheme. In *CRYPTO '99* (Santa Barbara, CA, Aug. 1999), pp. 431–448.

[4] BELLARE, M., AND YEE, B. S. Forward integrity for secure audit logs. Tech. rep., University of California at San Diego, Nov. 1997.

[5] BENALOH, J., AND DE MARE, M. One-way accumulators: a decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '93)* (Lofthus, Norway, May 1993), pp. 274–285.

[6] BETHENCOURT, J., BONEH, D., AND WATERS, B. Cryptographic methods for storing ballots on a voting machine. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2007).

[7] BLIBECH, K., AND GABILLON, A. CHRONOS: An authenticated dictionary based on skip lists for timestamping systems. In *Workshop on Secure Web Services* (Fairfax, VA, Nov. 2005), pp. 84–90.

[8] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[9] BULDAS, A., LAUD, P., LIPMAA, H., AND WILLEMSON, J. Time-stamping with binary linking schemes. In *CRYPTO '98* (Santa Barbara, CA, Aug. 1998), pp. 486–501.

[10] BULDAS, A., LIPMAA, H., AND SCHOENMAKERS, B. Optimally efficient accountable time-stamping. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)* (Melbourne, Victoria, Australia, Jan. 2000), pp. 293–305.

[11] CAMENISCH, J., AND LYSYANSKAYA, A. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO '02* (Santa Barbara, CA, Aug. 2002), pp. 61–76.

[12] CHAN, H., PERRIG, A., PRZYDATEK, B., AND SONG, D. SIA: Secure information aggregation in sensor networks. *Journal Computer Security 15*, 1 (2007), 69–102.

[13] CHAN, H., PERRIG, A., AND SONG, D. Secure hierarchical in-network aggregation in sensor networks. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, VA, Oct. 2006), pp. 278–287.

[14] CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. Private information retrieval. In *Annual Symposium on Foundations of Computer Science* (Milwaukee, WI, Oct. 1995), pp. 41–50.

[15] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested append-only memory: Making adversaries stick to their word. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 189–204.

[16] D. S. PARKER, J., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering 9*, 3 (1983), 240–247.

[17] DAVIS, D., MONROSE, F., AND REITER, M. K. Time-scoped searching of encrypted audit logs. In *Information and Communications Security Conference* (Malaga, Spain, Oct. 2004), pp. 532–545.

[18] DEUTSCH, P. Gzip file format specification version 4.3. RFC 1952, May 1996. http://www.ietf.org/rfc/rfc1952.txt.

[19] DEVANBU, P., GERTZ, M., KWONG, A., MARTEL, C., NUCKOLLS, G., AND STUBBLEBINE, S. G. Flexible authentication of XML documents. *Journal of Computer Security 12*, 6 (2004), 841–864.

[20] DEVANBU, P., GERTZ, M., MARTEL, C., AND STUBBLEBINE, S. G. Authentic data publication over the internet. *Journal Computer Security 11*, 3 (2003), 291–314.

[21] GENNARO, R., AND ROHATGI, P. How to sign digital streams. In *CRYPTO '97* (Santa Barbara, CA, Aug. 1997), pp. 180–197.

[22] GERR, P. A., BABINEAU, B., AND GORDON, P. C. Compliance: The effect on information management and the storage industry. The Enterprise Storage Group, May 2003. http://searchstorage.techtarget.com/tip/0,289483,sid5_gci906152,00.html.

[23] GOH, E.-J. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/ See also http://eujingoh.com/papers/secureindex/.

[24] GOODRICH, M., TAMASSIA, R., AND SCHWERIN, A. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II (DISCEX II)* (Anaheim, CA, June 2001), pp. 68–82.

[25] GOODRICH, M. T., TAMASSIA, R., TRIANDOPOULOS, N., AND COHEN, R. F. Authenticated data structures for graph and geometric searching. In *Topics in Cryptology, The Cryptographers' Track at the RSA Conference (CT-RSA)* (San Francisco, CA, Apr. 2003), pp. 295–313.

[26] GOYAL, V., PANDEY, O., SAHAI, A., AND WATERS, B. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, Virginia, Oct. 2006), pp. 89–98.

[27] HABER, S., AND STORNETTA, W. S. How to time-stamp a digital document. In *CRYPTO '98* (Santa Barbara, CA, 1990), pp. 437–455.

[28] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *SOSP '07* (Stevenson, WA, Oct. 2007).

[29] HOLT, J. E. Logcrypt: Forward security and public verification for secure audit logs. In *Australasian Workshops on Grid Computing and E-research* (Hobart, Tasmania, Australia, 2006).

[30] HU, L., AND EVANS, D. Secure aggregation for wireless networks. In *Symposium on Applications and the Internet Workshops (SAINT)* (Orlando, FL, July 2003), p. 384.

[31] ITKIS, G. Cryptographic tamper evidence. In *ACM Conference on Computer and Communications Security (CCS '03)* (Washington D.C., Oct. 2003), pp. 355–364.

[32] KELSEY, J., CALLAS, J., AND CLEMM, A. Signed Syslog messages. http://tools.ietf.org/id/draft-ietf-syslog-sign-23.txt (work in progress), Sept. 2007.

[33] KILTZ, E., MITYAGIN, A., PANJWANI, S., AND RAGHAVAN, B. Append-only signatures. In *International Colloquium on Automata, Languages and Programming* (Lisboa, Portugal, July 2005).

[34] KOCHER, P. C. On certificate revocation and validation. In *International Conference on Financial Cryptography*

(FC '98) (Anguilla, British West Indies, Feb. 1998), pp. 172–177.

[35] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 45–58.

[36] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Operating Systems Design & Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[37] LIPMAA, H. On optimal hash tree traversal for interval time-stamping. In *Proceedings of the 5th International Conference on Information Security (ISC02)* (Seoul, Korea, Nov. 2002), pp. 357–371.

[38] LONVICK, C. The BSD Syslog protocol. RFC 3164, Aug. 2001. http://www.ietf.org/rfc/rfc3164.txt.

[39] MA, D. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS'08)* (Tokyo, Japan, Mar. 2008), pp. 341–352.

[40] MA, D., AND TSUDIK, G. Forward-secure sequential aggregate authentication. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Oakland, CA, May 2007), IEEE Computer Society, pp. 86–91.

[41] MA, D., AND TSUDIK, G. A new approach to secure logging. *Transactions on Storage 5*, 1 (2009), 1–21.

[42] MANIATIS, P., AND BAKER, M. Enabling the archival storage of signed documents. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Monterey, CA, 2002).

[43] MANIATIS, P., AND BAKER, M. Secure history preservation through timeline entanglement. In *USENIX Security Symposium* (San Francisco, CA, Aug. 2002).

[44] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[45] MANULIS, M., AND SCHWENK, J. Provably secure framework for information aggregation in sensor networks. In *Computational Science and Its Applications (ICCSA)* (Kuala Lumpur, Malaysia, Aug. 2007), pp. 603–621.

[46] MERKLE, R. C. A digital signature based on a conventional encryption function. In *CRYPTO '88* (1988), pp. 369–378.

[47] MITRA, S., HSU, W. W., AND WINSLETT, M. Trustworthy keyword search for regulatory-compliant records retention. In *International Conference on Very Large Databases (VLDB)* (Seoul, Korea, Sept. 2006), pp. 1001–1012.

[48] MONTEIRO, S. D. S., AND ERBACHER, R. F. Exemplifying attack identification and analysis in a novel forensically viable Syslog model. In *Workshop on Systematic Approaches to Digital Forensic Engineering* (Oakland, CA, May 2008), pp. 57–68.

[49] NAOR, M., AND NISSIM, K. Certificate revocation and certificate update. In *USENIX Security Symposium* (San Antonio, TX, Jan. 1998).

[50] OSTROVSKY, R., SAHAI, A., AND WATERS, B. Attribute-based encryption with non-monotonic access structures. In *ACM Conference on Computer and Communications Security (CCS '07)* (Alexandria, VA, Oct. 2007), pp. 195–203.

[51] PAVLOU, K., AND SNODGRASS, R. T. Forensic analysis of database tampering. In *ACM SIGMOD International Conference on Management of Data* (Chicago, IL, June 2006), pp. 109–120.

[52] PETERSON, Z. N. J., BURNS, R., ATENIESE, G., AND BONO, S. Design and implementation of verifiable audit trails for a versioning file system. In *USENIX Conference on File and Storage Technologies* (San Jose, CA, Feb. 2007).

[53] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.

[54] SAHAI, A., AND WATERS, B. Fuzzy identity based encryption. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '05)* (May 2005), vol. 3494, pp. 457 – 473.

[55] SANDLER, D., AND WALLACH, D. S. Casting votes in the Auditorium. In *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)* (Boston, MA, Aug. 2007).

[56] SCHNEIER, B., AND KELSEY, J. Automatic event-stream notarization using digital signatures. In *Security Protocols Workshop* (Cambridge, UK, Apr. 1996), pp. 155–169.

[57] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security 1*, 3 (1999).

[58] SION, R. Strong WORM. In *International Conference on Distributed Computing Systems* (Beijing, China, May 2008), pp. 69–76.

[59] SNODGRASS, R. T., YAO, S. S., AND COLLBERG, C. Tamper detection in audit logs. In *Conference on Very Large Data Bases (VLDB)* (Toronto, Canada, Aug. 2004), pp. 504–515.

[60] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy* (Berkeley, CA, May 2000), pp. 44–55.

[61] WATERS, B. R., BALFANZ, D., DURFEE, G., AND SMETTERS, D. K. Building an encrypted and searchable audit log. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2004).

[62] WEATHERSPOON, H., WELLS, C., AND KUBIATOWICZ, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Future Directions in Distributed Computing* (2003), vol. 2584 of *Lecture Notes in Computer Science*, pp. 142–147.

[63] YUMEREFENDI, A. R., AND CHASE, J. S. Strong accountability for network storage. *ACM Transactions on Storage 3*, 3 (2007).

[64] ZHU, Q., AND HSU, W. W. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *ACM SIGMOD International Conference on Management of Data* (Baltimore, MD, June 2005), pp. 395–406.

VPriv: Protecting Privacy in Location-Based Vehicular Services

Raluca Ada Popa and Hari Balakrishnan
Massachusetts Institute of Technology
Email: {ralucap,hari}@mit.edu

Andrew J. Blumberg
Stanford University
Email: blumberg@math.stanford.edu

Abstract

A variety of location-based vehicular services are currently being woven into the national transportation infrastructure in many countries. These include usage- or congestion-based road pricing, traffic law enforcement, traffic monitoring, "pay-as-you-go" insurance, and vehicle safety systems. Although such applications promise clear benefits, there are significant potential violations of the *location privacy* of drivers under standard implementations (i.e., GPS monitoring of cars as they drive, surveillance cameras, and toll transponders).

In this paper, we develop and evaluate *VPriv*, a system that can be used by several such applications without violating the location privacy of drivers. The starting point is the observation that in many applications, some centralized server needs to compute a function of a user's *path*—a list of time-position tuples. VPriv provides two components: 1) the first practical protocol to compute path functions for various kinds of tolling, speed and delay estimation, and insurance calculations in a way that does not reveal anything more than the result of the function to the server, and 2) an out-of-band enforcement mechanism using random spot checks that allows the server and application to handle misbehaving users. Our implementation and experimental evaluation of VPriv shows that a modest infrastructure of a few multi-core PCs can easily serve 1 million cars. Using analysis and simulation based on real vehicular data collected over one year from the CarTel project's testbed of 27 taxis running in the Boston area, we demonstrate that VPriv is resistant to a range of possible attacks.

1 Introduction

Over the next few years, location-based vehicular services using a combination of in-car devices and roadside surveillance systems will become a standard feature of the transportation infrastructure in many countries. Already, there is a burgeoning array of applications of such technology, including electronic toll collection, automated traffic law enforcement, traffic statistic collection, insurance pricing using measured driving behavior, vehicle safety systems, and so on.

These services promise substantial improvements to the efficiency of the transportation network as well as to the daily experience of drivers. Electronic toll collection reduces bottlenecks at toll plazas, and more sophisticated forms of congestion tolling and usage pricing (e.g., the London congestion tolling system [24]) reduce traffic at peak times and generate revenue for transit improvements. Although the efficacy of automated traffic enforcement (e.g., stop-light cameras) is controversial, many municipalities are exploring the possibility that it will improve compliance with traffic laws and reduce accidents. Rapid collection and analysis of traffic statistics can guide drivers to choose optimal routes and allows for rational analysis of the benefits of specific allocations of transportation investments. Some insurance companies (e.g. [21]) are now testing or even deploying "pay-as-you-go" insurance programs in which insurance premiums are adjusted using information about driving behavior collected by GPS-equipped in-car devices.

Unfortunately, along with the tremendous promise of these services come very serious threats to the *location privacy* of drivers (see Section 3 for a precise definition). For instance, some current implementations of these services involve pervasive tracking—toll transponder transmitting client/account ID, license-plate cameras, mandatory in-car GPS [32], and insurance "black boxes" that monitor location and other driving information—with the data aggregated centrally by various government and corporate entities.

Furthermore, as a pragmatic matter, the widespread deployment and adoption of traffic monitoring is greatly impaired by public concern about privacy issues. A sizable impediment to further electronic tolling penetration in the San Francisco Bay Area is the refusal of a significant minority of drivers to install the devices due to

privacy concerns [31]. Privacy worries also affect the willingness of drivers to participate in the collection of traffic statistics.

This paper proposes *VPriv*, a practical system to protect a user's locational privacy while efficiently supporting a range of location-based vehicular services. VPriv supports applications that compute functions over the *paths* traveled by individual cars. A path is simply a sequence of *points*, where each point has a random time-varying identifier, a timestamp, and a position. Usage-based tolling, delay and speed estimation, as well as pay-as-you-go calculations can all be computed given the paths of each driver.

VPriv has two components. The first component is an *efficient protocol for tolling and speed or delay estimation that protects the location privacy of the drivers*. This protocol, which belongs to the general family of secure multi-party computations, guarantees that a joint computation between server and client can proceed correctly without revealing the private data of the parties involved. The result is that each driver (car) is guaranteed that no other information about his paths can be inferred from the computation, other than what is revealed by the result of the computed function. The idea of using multi-party secure computation in the vehicular setting is inspired from previous work [2, 3, 30]; however, these papers use multi-party computations as a black box, relying on general reductions from the literature. Unfortunately, these are extremely slow and complex, at least three orders of magnitude slower than our implementation in our experiments (see Section 8.2), which makes them unpractical.

Our main contribution here is the first *practically efficient* design, software implementation, and experimental evaluation of multi-party secure protocols for functions computed over driving paths. Our protocols exploit the specificity of cost functions over path time-location tuples: the path functions we are interested in consist of sums of costs of tuples, and we use homomorphic encryption [29] to allow the server to compute such sums using encrypted data.

The second component of VPriv addresses a significant concern: *making VPriv robust to physical attacks*. Although we can prove security against "cryptographic attacks" using the mathematical properties of our protocols, it is very difficult to protect against physical attacks in this fashion (e.g., drivers turning off their devices). However, one of the interesting aspects of the problem is that the embedding in a social and physical context provides a framework for discovering misbehavior. We propose and analyze a method using sporadic random spot-checks of vehicle locations that *are* linked to the actual identity of the driver. This scheme is general and independent of the function to be computed because it checks that *the argument* (driver paths) to the secure two-party protocol is highly likely to be correct. Our analysis shows that this goal can be achieved with a small number of such checks, making this enforcement method inexpensive and minimally invasive.

We have implemented VPriv in C++ (and also Javascript for a browser-based demonstration). Our measurements show that the protocol runs in 100 seconds per car on a standard computer. We estimate that 30 cores of 2.4GHz speed, connected over a 100 Megabits/s link, can easily handle 1 million cars. Thus, the infrastructure required to handle an entire state's vehicular population is relatively modest. Our code is available at `http://cartel.csail.mit.edu/#vpriv`.

2 Related work

VPriv is inspired by recent work on designing cryptographic protocols for vehicular applications [2, 3, 30]. These works also discuss using random vehicle identifiers combined with secure multi-party computation or zero-knowledge proofs to perform various vehicular computations. However, these papers employ multi-party computations as a black box, relying on general results from the literature for reducing arbitrary functions to secure protocols [34]. Such protocols tend to be very complex and slow. The state of the art "general purpose" compiler for secure function evaluation, Fairplay [26], produces implementations which run more than three orders of magnitude more slowly than the VPriv protocol, and scale very poorly with the number of participating drivers (see Section 8.2). Given present hardware constraints, general purpose solutions for implementing secure computations are simply not viable for this kind of application. A key contribution of this paper is to present a protocol for the *specific* class of cost functions on time-location pairs, which maintains privacy and is efficient enough to be run on practical devices and suitable for deployment.

Electronic tolling and public transit fare collection were some of the early application areas for anonymous electronic cash. Satisfactory solutions to certain classes of road-pricing problems (e.g., cordon-based tolling) can be developed using electronic cash algorithms in concert with anonymous credentials [6, 25, 1]. There has been a substantial amount of work on practical protocols for these problems so that they run efficiently on small devices (e.g., [5]). Physical attacks based on the details of the implementation and the associated bureaucratic structures remain a persistent problem, however [13]. We explicitly attempt to address such attacks in VPriv. Our "spot check" methodology provides a novel approach to validating user participation in the cryptographic protocols, and we prove its efficiency empirically. Furthermore, unlike VPriv, the electronic cash

approach is significantly less suitable for more sophisticated road pricing applications, and does not apply at all to the broader class of vehicular location-based services such as "pay-as-you-go" insurance, automated traffic law enforcement, and aggregate traffic statistic collection.

There has also been a great deal of related work on protecting location privacy and anonymity while collecting vehicular data (e.g., traffic flow data) [18, 22, 16]. The focus of this work is different from ours, although it can be used in conjunction. It analyzes potential privacy violations associated with the side channels present in anonymized location databases (e.g., they conclude that it is possible to infer to what driver some GPS traces belong in regions of low density).

Using spatial analogues of the notion of *k-anonymity* [33], some work focused on using a trusted server to spatially and temporally distort locational services [15, 10]. In addition, there has been a good deal of work on using a trusted server to distort or degrade data before releasing it. An interesting class of solutions to these problems were presented in the papers [19, 17], involving "cloaking" the data using spatial and temporal subsampling techniques. In addition, these papers [17, 19] developed tools to quantify the degree of mixing of cars on a road needed to assure anonymity (notably the "time to confusion" metric). However, these solutions treat a different problem than VPriv, because most of them assume a trusted server and a non-adversarial setting, in which the user and server do not deviate from the protocol, unlike in the case of tolling or law enforcement. Furthermore, for many of the protocols we are interested in, it is not always possible to provide time-location tuples for only a subset of the space.

Nonetheless, the work in these papers complements our protocol nicely. Since VPriv does produce an anonymized location database, the analysis in [17] about designing "path upload" points that adequately preserve privacy provides a method for placing tolling regions and "spot checks" which do not violate the location privacy of users. See Section 9 for further discussion of this point.

3 Model

In this section, we describe the framework underlying our scheme, goals, and threat model. The framework captures a broad class of vehicular location-based services.

3.1 Framework

The participants in the system are *drivers*, *cars* and *a server*. Drivers operate cars, cars are equipped with transponders that transmit information to the server, and drivers also run *client software* which enacts the cryptographic protocol on their behalf.

For any given problem (tolling, traffic statistics estimation, insurance calculations, etc.), there is one logical server and many drivers with their cars. The server computes some function f for any given car; f takes the path of the car generated during an *interaction interval* as its argument. To compute f, the server must collect the set of points corresponding to the path traveled by the car during the desired interaction interval. Each point is a tuple with three fields: $\langle$`tag, time, location`$\rangle$.

While driving, each car's transponder generates a collection of such tuples and sends them to the server. The server computes f using the set of $\langle$`time,location`$\rangle$ pairs. If location privacy were not a concern, the `tag` could uniquely identify the car. In such a case, the server could aggregate all the tuples having the same tag and know the path of the car. Thus, in our case, these tags will be chosen at random so that they cannot be connected to an individual car. However, the driver's client application will give the server a *cryptographic commitment* to these tags (described in Sections 4.1, 5): in our protocol, this commitment binds the driver to the particular tags and hence the result of f (e.g., the tolling cost) without revealing the tags to the server.

We are interested in developing protocols that preserve location privacy for three important functions:

1. *Usage-based tolls:* The server assesses a path-dependent toll on the car. The toll is some function of the time and positions of the car, known to both the driver and server. For example, we might have a toll that sets a particular price per mile on any given road, changing that price with time of day. We call this form of tolling a *path toll*; VPriv also supports a *point toll*, where a toll is charged whenever a vehicle goes past a certain point.
2. *Automated speeding tickets:* The server detects violations of speed restrictions: for instance, did the car ever travel at greater than 65 MPH? More generally, the server may wish to detect violations of speed limits which vary across roads and are time-dependent.
3. *"Pay-as-you-go" insurance premiums:* The server computes a "safety score" based on the car's path to determine insurance premiums. Specifically, the server computes some function of the time, positions, and speed of the car. For example, we might wish to assess higher premiums on cars that persistently drive close to the speed limit, or are operated predominantly late at night.

These applications can be treated as essentially similar examples of the basic problem of computing a localized cost function of the car's path represented as points. By localized we mean that the function can be decom-

posed as a sum of costs associated to a specific point or small number of specific points that are close together in space-time. In fact, our general framework can be applied to any function over path tuples because of the general result that every polynomially computable function has a secure multi-party protocol [12, 34]. However, as discussed in Section 8.2, these general results lead to impractical implementations: instead, we devise efficient protocols by exploiting the specific form of the cost functions.

In our model, each car's transponder (transponder may be tampered with) obtains the point tuples as it drives and delivers them to the server. These tasks can be performed in several ways, depending on the infrastructure and resources available. For example, tuples can be generated as follows:

- A *GPS* device provides location and time, and the car's transponder prepares the tuples.
- *Roadside devices* sense passing cars, communicate with a car's transponder to receive a tag, and create a tuple by attaching time information and the fixed location of the roadside device.

Each car generates tuples periodically; depending on the specific application, either at random intervals (e.g., roughly every 30 seconds) or potentially based on location as well, for example at each intersection if the car has GPS capability. The tuples can be delivered rapidly (e.g., via roadside devices, the cellular network, or available WiFi [9]) or they can be batched until the end of the day or of the month. Section 9 describes how to avoid leaking private information when transmitting such packets to the server.

Our protocol is independent of the way these tuples are created and sent to the server, requiring only that tuples need to reach the server before the function computation. This abstract model is flexible and covers many practical systems, including in-car device systems (such as CarTel [20]), toll transponder systems such as E-ZPass [14], and roadside surveillance systems.

3.2 Threat model

Many of the applications of VPriv are adversarial, in that both the driver and the operator of the server may have strong financial incentives to misbehave. VPriv is designed to resist five types of attacks:

1. The driver attempts to cheat by using a modified client application during the function computation protocol to change the result of the function.
2. The driver attempts to cheat physically, by having the car's transponder upload incorrect tuples (providing incorrect inputs to the function computation protocol):
   (a) The driver turns off or selectively disables the in-car transponder, so the car uploads no data or only a subset of the actual path data.
   (b) The transponder uploads synthetic data.
   (c) The transponder eavesdrops on another car and attempts to masquerade as that car.
3. The server guesses the path of the car from the uploaded tuples.
4. The server attempts to cheat during the function computation protocol to change the result of the function or obtain information about the path of the car.
5. Some intermediate router synthesizes false packets or systematically changes packets between the car's transponder and the server.

All these attacks are counteracted in our scheme as discussed in Section 9. Note however that in the main discussion of the protocol, for ease of exposition we treat the server as a passive adversary; we assume that the server attempts to violate the privacy of the driver by inferring private data but correctly implements the protocol (e.g. does not claim the driver failed a verification test, when she did not). We believe this is a reasonable assumption since the server is likely to belong to an organization (e.g., the government or an insurance company) which is unlikely to engage in active attacks. However, as we discuss in Section 9, the protocol can be made resilient to a fully malicious server as well with very few modifications.

3.3 Design goals

We have the following goals for the protocol between the driver and the server, which allows the server to compute a function over a private path.

Correctness. For the car C with path P_C, the server computes the correct value of $f(P_C)$.

Location privacy. We formalize our notion of location privacy in this paper as follows:

Definition 1 ***(Location privacy)*** *Let*

- $\mathbb{S}$ *denote the server's database consisting of* $\langle$*tag,time,location*$\rangle$ *tuples.*
- $\mathbb{S}'$ *denote the database generated from* $\mathbb{S}$ *by removing the tag associated to each tuple: for every tuple* $\langle$*tag, location, time*$\rangle \in \mathbb{S}$*, there is a tuple* $\langle$*location, time*$\rangle \in \mathbb{S}'$*.*
- C *be an arbitrary car.*
- $\mathcal{V}$ *denote all the information available to the server in VPriv ("the server's view"). This comprises the information sent by* C *to the server while executing the protocol (including the result of the function computation) and any other information owned or computed by the server during the computation of* f*(path of* C*), (which includes* $\mathbb{S}$*).*

- *$\mathcal{V}'$ denote all the information contained in $\mathbb{S}'$, the result of applying f on C, and any other side channels present in the raw database $\mathbb{S}'$.*

The computation of f(path of C) preserves the locational privacy *of C if the server's information about C's tuples is insignificantly larger in $\mathcal{V}$ than in $\mathcal{V}'$.*

Here the "insignificant amount" refers to an amount of information that cannot be exploited by a computationally bounded machine. For instance, the encryption of a text typically offers some insignificant amount of information about the text. This notion can be formalized using simulators, as is standard for this kind of cryptographic guarantee. Such a mathematical definition and proof is left for an extended version of our paper.

Informally, this definition says that the privacy guarantees of VPriv are the same as those of a system in which the server stores only tag-free path points $\langle$`time,location`$\rangle$ without any identifying information and receives (from an oracle) the result of the function (without running any protocol). Note that this definition means that any side channels present in the raw data of $\mathbb{S}$ itself will remain in our protocols; for instance, if one somehow knows that only a single car drives on certain roads at a particular time, then that car's privacy will be violated. See Section 9 for further discussion of this issue.

Efficiency. The protocol must be sufficiently efficient so as to be feasible to run on inexpensive in-car devices. This goal can be hard to achieve; modern cryptographic protocols can be computationally intensive.

Note that we do not aim to hide the result of the function; rather, we want to compute this result without revealing private information. In some cases, such as tolling, the result may reveal information about the path of the driver. For example, a certain toll cost may be possible only by a combination of certain items. However, if the toll period is large enough, there may be multiple combinations of tolls that add to the result. Also, finding such a combination is equivalent to the subset-sum problem, which is NP-complete.

4 Architecture

This section gives an overview of the VPriv system and its components. There are three software components: the client application, which runs on the client's computer, a transponder device attached to the car, and the server software attached to a tuple database. The only requirements on the transponder are that it store a list of random tags and generate tuples as described in Section 3.1. The client application is generally assumed to be executed on the driver's home computer or mobile device like a smart-phone.

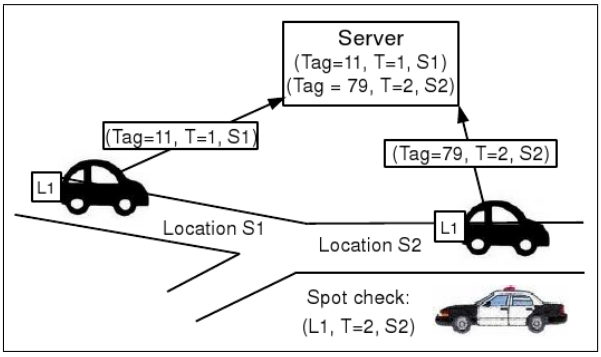


Figure 1: Driving phase overview: A car with license plate L1 is traveling from Location S1 at time 1 to Location S2 at time 2 when it undergoes a spot check. It uploads path tuples to the server.

The protocol consists of the following phases:

1. Registration. From time to time—say, upon renewing a car's registration or driver license—the driver must identify herself to the server by presenting a license or registration information. At that time, the client application generates a set of random tags that will be used in the protocol. We assume that these are indistinguishable from random by a computationally bounded adversary. The tags are also transferred to the car's transponder, but *not* given to the server. The client application then cryptographically produces *commitments* to these random tags. We describe the details of computing these commitments in Sections 4.1 and 5. The client application will provide the ciphertext of the commitments to the server and these *will* be bound to the driver's identity; however, they do not reveal any information about the actual tags under cryptographic assumptions.

2. Driving. As the car is driven, the transponder gathers time-location tuples and uploads them to the server. Each path tuple is unique because the random tag is never reused (or reused only in a precisely constrained fashion, see Section 5). The server *does not know* which car uploaded a certain tuple. To ensure that the transponder abides by the protocol, VPriv also uses sporadic random spot checks that observe the physical locations of cars, as described in Section 6. At a high level, this process generates tuples consisting of the actual license plate number, time, and location of observation. Since these spot checks record license plate information, the server knows which car they belong to. During the next phase, the client application will have to prove that the tuples uploaded by the car's transponder are consistent with these spot checks. Figure 1 illustrates the driving phase.

3. Reconciliation. This stage happens at the end of each interaction interval (e.g., at the end of the month, when a driver pays a tolling bill) and computes the function f. The client authenticates itself via a web con-

nection to the server. He does not need to transfer any information from the transponder to the computer (unless the tuples can be corrupted or lost on their way to the server and the client needs to check that they are all there). It is enough if his computer knows the initial tags (from registration). If the car had undergone a spot check, the client application has to prove that the tuples uploaded are consistent with the spot checks before proceeding (as explained in Section 6). Then, the client application initiates the function computation. The server has received tuples from the driver's car, generated in the driving phase. However, the server has also received similar tuples from many other cars and *does not know* which ones belong to a specific car. Based on this server database of tuples as well as the driver's commitment information from registration, the server and the client application conduct a cryptographic protocol in which:

- The client computes the desired function on the car's path, the path being the *private input*.
- Using a zero-knowledge proof, the client application proves to the server that the result of the function is correct, by answering correctly a series of challenges posed by the server *without revealing the driver's tags*.

The reconciliation can be done transparently to the user the client software; from the perspective of the user, he only needs to perform an online payment.

To implement this protocol, VPriv uses a set of modern cryptographic tools: a homomorphic commitment scheme and random function families. We provide a brief overview of these tools below. The experienced reader may skip to Section 5, where we provide efficient realizations that exploit details of our restricted problem setting.

4.1 Overview of cryptographic mechanisms

A **commitment scheme** [4] consists of two algorithms, *Commit* and *Reveal* or *Decommit*. Assume that Alice wants to commit to a value v to Bob. In general terms, Alice wants to provide a ciphertext to Bob from which he cannot gain any information about v. However, Alice needs to be bound to the value of v. This means that, later when she wants to reveal v to Bob, she cannot provide a different value, $v' \neq v$, which matches the same ciphertext. Specifically, she computes *Commit*$(v) \rightarrow (c(v), d(v))$, where $c(v)$ is the resulting ciphertext and $d(v)$ is a decommitment key with the following properties:

- Bob cannot feasibly gain any information from c.
- Alice cannot feasibly provide $v' \neq v$ such that *Commit*$(v') \rightarrow (c(v), d')$, for some d'.

$COST$	Path tolling cost computed by the client and reported to the server.
$c(x), d(x)$	The **c**iphertext and **d**ecommitment value resulting from committing to value x. That is, *Commit*$(x) = (c(x), d(x))$.
v_i	The random tags used by the **v**ehicle's transponder. A subset of these will be used while driving.
(s_i, t_i)	A pair formed of a random tag uploaded at the **s**erver and the **t**oll cost the server associates with it. $\{s_i\}$ is the set of all random tags the server received within a tolling period with $t_i > 0$.

Figure 2: Notation.

We say that Alice reveals v to Bob if she provides v and $d(v)$, the decommitment value, to Bob, who already holds $c(v)$. Note that c and d are not functions applied to v; they are values resulting when computing *Commit*(v) and stored for when v is revealed.

We use a **homomorphic commitment** scheme (such as the one introduced by Pedersen [29]), in which performing an arithmetic operation on the ciphertexts corresponds to some arithmetic operation on the plaintext. For instance, a commitment scheme that has the property that $c(v) \cdot c(v') = c(v + v')$ is homomorphic. Here, the decommitment key of the sum of the plaintexts is the sum of the decommitment keys $d(v + v') = d(v) + d(v')$.

A **secure multi-party computation** [34] is a protocol in which several parties hold private data and engage in a protocol in which they compute the result of a function on their private data. At the end of the protocol, the correct result is obtained and none of the participants can learn the private information of any other beyond what can be inferred from the result of the function. In this paper, we designed a variant of a secure two-party protocol. One party is the car/driver whose private data is the driving path, and the other is the server, which has no private data. A **zero-knowledge proof** [12] is a related concept that involves proving the truth of a statement without revealing any other information.

A **pseudorandom function family** [27] is a collection of functions $\{f_k\} : D \rightarrow R$ with domain D and range R, indexed by k. If one chooses k at random, for all $v \in D$, $f_k(v)$ can be computed efficiently (that is, in polynomial time) and f_k is indistinguishable from a function with random output for each input.

5 Protocols

This section presents a detailed description of the specific interactive protocol for our applications, making precise

the preceding informal description. For concreteness, we describe the protocol first in the case of the tolling application; the minor variations necessary to implement the speeding ticket and insurance premium applications are presented subsequently.

5.1 Tolling protocol

We first introduce the notation in Figure 2. For clarity, we present the protocol in a schematic manner in Figure 3. For simplicity, the protocol is illustrated for only one round. For multiple rounds, we need *a different random function for each round*. (The reason is that if the same random function is used across rounds, the server could guess the tuples of the driver by posing a $b = 0$ and a $b = 1$ challenge.) The registration phase is the same for multiple rounds, with the exception that multiple random functions are chosen in Step (a) and Steps (b) and (c) are executed for each random function.

This protocol is a case of two party-secure computation (the car is a malicious party with private data and the server is an honest but curious party) that takes the form of zero-knowledge proof: the car first computes the tolling cost and then it proves to the server that the result is correct. Intuitively, the idea of the protocol is that the client provides the server an encrypted version of her tags on which the server can compute the tolling cost in ciphertext. The server has a way of verifying that the ciphertext provided by the client is correct. The privacy property comes from the fact that the server can perform only one of the two operations at the same time: either check that the ciphertext is computed correctly, or compute the tolling cost on the vehicle tags using the ciphertext. Performing both means figuring out the driver's tuples.

These verifications and computations occur within a round, and there are multiple rounds. During each round, the server has a probability of at least $1/2$ to detect whether the client provided an incorrect COST, as argued in the proof below. The round protocol should be repeated s times, until the server has enough confidence in the correctness of the result. After s rounds, the probability of detecting a misbehaving client is at least $1-(1/2)^s$, which decreases exponentially. Thus, for $s = 10$, the client is detected with 99.9% probability. The number of rounds is fixed and during registration the client selects a pseudorandom function f_k for each round and provides a set of commitments for each round.

Note that this protocol also reveals the number of tolling tuples of the car because the server knows the size of the intersection (i.e. the number of matching encryptions $f_k(v_i) = f_k(s_j)$ in iv) for $b = 1$). We do not regard this as a significant problem, since the very fact that a particular amount was paid may reveal this number (especially for cases where the tolls are about equal). However, if desired, we can handle this problem by uploading some "junk tuples". These tuples still use valid driver tags, but the location or time can be an indication to the server that they are junk and thus the server assigns a zero cost. These tuples will be included in the tolling protocol when the server will see them encrypted and will not know how many junk tuples are in the intersection of server and driver tuples and thus will not know how many actual tolling tuples the driver has. Further details of this scheme are not treated here due to space considerations.

First, it is clear that if the client is honest, the server will accept the tolling cost.

Theorem 1 *If the server responds with "ACCEPT", the protocol in Figure 3 results in the correct tolling cost and respects the driver's location privacy.*

Proof: Assume that the client has provided an incorrect tolling cost in step 3b. Note first that all decommitment keys provided to the server must be correct; otherwise the server would have detected this when checking that the commitment was computed correctly. Then, at least one of the following data provided by the client provides has to be incorrect:

- The encryption of the pairs (s_j, t_j) obtained from the server. For instance, the car could have removed some entries with high cost so that the server computes a lower total cost in step iv).
- The computation of the total toll $COST$. That is, $COST \neq \sum_{v_i=s_j} t_j$. For example, the car may have reported a smaller cost.

For if both are correct, the tolling cost computed must be correct.

During each round, the server chooses to test one of these two conditions with a probability of $1/2$. Thus, if the tolling cost is incorrect, the server will detect the misbehavior with a probability of at least $1/2$. As discussed, the detection probability increases exponentially in the number of rounds.

For location privacy, we prove that the server gains no significant additional information about the car's data other than the tolling cost and the number of tuples involved in the cost (and see above for how to avoid the latter). Let us examine the information the server receives from the client:

Step (1c): The commitments $c(k)$ and $c(f_k(v_i))$ do not reveal information by the definition of a commitment scheme.

Step (i): $c(t_j)$ does not reveal information by the definition of a commitment scheme. By the definition of the pseudorandom function, $f_k(s_i)$ looks random. After

1. **Registration phase:**
   (a) Each client chooses random vehicle tags, v_i, and a random function, f_k (one per round), by choosing k at random.
   (b) Encrypts the selected vehicle tags by computing $f_k(v_i), \forall i$, commits to the random function by computing $c(k)$, commits to the encrypted vehicle tags by computing $c(f_k(v_i))$, and stores the associated decommitment keys, $(d(k), d(f_k(v_i)))$.
   (c) Send $c(k)$ and $c(f_k(v_i)), \forall i$ to the server. This will prevent the car from using different tags.
2. **Driving phase:** The car produces path tuples using the random tags, v_i, and sends them to the server.
3. **Reconciliation phase:**
   (a) The server computes the associated tolling cost, t_j, for each random tag s_j received at the server in the last period based on the location and time where it was observed and sends (s_j, t_j) to the client only if $t_j > 0$.
   (b) The client computes the tolling cost $COST = \sum_{v_i = s_j} t_j$ and sends it to the server.
   (c) The **round protocol** (client proves that $COST$ is correct) begins:

Client	Server
(i) Shuffle at random the pairs (s_j, t_j) obtained from the server. Encrypt s_j according to the chosen f_k random function by computing $f_k(s_j), \forall j$. Compute $c(t_j)$ and store the associated decommitments.	
Send to server $f_k(s_j)$ and $c(t_j)\ , \forall j\ \rightarrow$	
	(ii) The server picks a bit b at random. If $b = 0$, challenge the client to verify that the ciphertext provided is correct; else ($b = 1$) challenge the client to verify that the total cost based on the received ciphertext matches $COST$.
	$\leftarrow$ Challenge random bit b
(iii) If $b = 0$, the client sends k and the set of (s_j, t_j) in the shuffled order to the server and proves that these are the values she committed to in step (i) by providing $d(k)$ and $d(t_j)$. If $b = 1$, the client sends the ciphertexts of all v_i ($f_k(v_i)$) and proves that these are the values she committed to during registration by providing $d(f_k(v_i))$. The client also computes the intersection of her and the server's tags, $I = \{v_i, \forall\ i\} \cap \{s_j, \forall\ j\}$. Let $T = \{t_j : s_j \in I\}$ be the set of associated tolls to s_j in the intersection. Note that $\sum_T t_j$ represents the total tolling cost the client has to pay. By the homomorphic property discussed in Section 4.1, the product of the commitments to these tolls t_j, $\prod_{t_j \in T} c(t_j)$, is a ciphertext of the total tolling cost whose decommitment key is $D = \sum_{t_j \in T} d(t_j)$. The server will compute the sum of these costs in ciphertext in order to verify that $COST$ is correct; the client needs to provide D for this verification.	
If $b = 0$, $d(k), d(t_i)$ else $D, d(f_k(v_i))\ \rightarrow$	
	(iv) If $b = 0$, the server verifies that all pairs (s_j, t_j) have been correctly shuffled, encrypted with f_k, and committed. This verifies that the client computed the ciphertext correctly. If $b = 1$, the server computes $\prod_{j:\exists\ i,\ f_k(v_i) = f_k(s_j)} c(t_j)$. As discussed, this yields a ciphertext of the total tolling cost and the server verifies if it is a commitment to $COST$ using D. If all checks succeed, the server *accepts* the tolling cost, else it *denies* it.

Figure 3: VPriv's protocol for computing the path tolling cost (small modifications of this basic protocol work for the other applications). The arrows indicate data flow.

the client shuffles at random the pairs (s_j, t_j), the server cannot tell which $f_k(s_j)$ corresponds to which s_j. Without such shuffling, even if the s_j is encrypted, the server would still know that the j-th ciphertext corresponds to the j-th plaintext. This will break privacy in Step (iv) for $b = 1$ when the server compares the ciphertext of s_j to the ciphertext of v_j.

Step (iii): If $b = 0$, the client will reveal k and t_j and no further information from the client will be sent to the server in this round. Thus, the values of $f_k(v_i)$ remain committed so the server has no other information about v_i other than these committed values, which do not leak information. If $b = 1$, the client reveals $f_k(v_i)$. However, since k is not revealed, the server does not know which pseudorandom function was used and due to the pseudorandom function property, the server cannot find

v_i. Providing D only provides decommitment to the sum of the tolls which is the result of the function, and no additional information is leaked (i.e., in the case of the Pedersen scheme).

Information across rounds: A different pseudorandom function is used during every round so the information from one round cannot be used in the next round. Furthermore, the commitment to the same value in different rounds will be different and look random.

Therefore, we support our definition of location privacy because the road pricing protocol does not leak any additional information about whom the tuple tags belong to and the cars generated the tags randomly. □

The protocol is linear in the number of tuples the car commits to during registration and the number of tuples received from the server in step 3a. It is easy to modify slightly the protocol to reduce the number of tuples that need to be downloaded as discussed in Section 7.

Point tolls (replacement of tollbooths). The predominant existing method of assessing road tolls comes from point-tolling; in such schemes, tolls are assessed at particular points, or linked to entrance/exit pairs. The latter is commonly used to charge for distance traveled on public highways. Such tolling schemes are easily handled by our protocol; tuples are generated corresponding to the tolling points. Tolls that depend on the entrance/exit pairs can be handled by uploading a pair of tuples with the same tag; we discuss this refinement in detail for computation of speed below in Section 5.2. The tolling points can be "virtual", or alternatively an implementation can utilize the existing E-Zpass infrastructure:

- The transponder knows a list of places where tuples need to be generated, or simply generates a tuple per intersection using GPS information.
- An (existing) roadside router infrastructure at tolling places can signal cars when to generate tuples.

Other tolls. Another useful toll function is charging cars for driving in certain regions. For example, cars can be charged for driving in the lower Manhattan core, which is frequently congested. One can modify the tolling cost protocol such that the server assigns a cost of 1 to every tuple inside the perimeter of this region. If the result of the function is positive, it means that the client was in the specific region.

5.2 Speeding tickets

In this application, we wish to detect and charge a driver who travels above some fixed speed limit L. For simplicity, we will initially assume that the speed limit is the same for all roads, but it is straightforward to extend the solution to varying speed limits. this constraint. The idea is to cast speed detection as a tolling problem, as follows.

We modify the driving phase to require that the car uses each random vehicle tag v_i twice; thus the car will upload pairs of linked path tuples. The server can compute the speed from a pair of linked tuples, and so during the reconciliation phase, the server assigns a cost t_i to each linked pair: if the speed computed from the pair is $> L$, the cost is non-zero, else it is zero. Now the reconciliation phase proceeds as discussed above. The spot check challenge during the reconciliation phase now requires verification that a consistent pair of tuples was generated, but is otherwise the same. If it deemed useful that the car reveal information about *where* the speeding violation occurred, the server can set the cost t_i for a violating pair to be a unique identifier for that speeding incident.

Note that this protocol leaves "gaps" in coverage during which speeding violations are not detected. Since these occur every other upload period, it is hard to imagine a realistic driver exploiting this. Likely, the driver will be travelling over the speed limit for the duration of several tuple creations. However, if this is deemed to be a concern for a given application, a variant can be used in which the period of changing tuples is divided and linked pairs are interleaved so that the whole time range is covered: $\ldots v_2\ v_1\ v_3\ v_2\ v_4\ v_3\ v_5\ v_4\ v_6\ v_5 \ldots$

The computational costs of this protocol are analogous to the costs of the tolling protocol and so the experimental analysis of that protocol applies in this case as well. There is a potential concern about additional side channels in the server's database associated with the use of linked tuples. Although the driver has the same guarantees as in the tolling application that her participation in the protocol does not reveal any information beyond the value of the function, the server has additional raw information in the form of the linkage. The positional information leaked in the linked tuple model is roughly the same as in the tolling model with twice the time interval between successive path tuples. Varying speed limits on different roads can be accommodated by having the prices t_i incorporate location.

5.3 Insurance premium computation

In this application, we wish to assign a "safety score" to a driver based on some function of their path which assesses their accident risk for purposes of setting insurance premiums. For example, the safety score might reflect the fraction of total driving time that is spent driving above 45 MPH at night. Or the safety score might be a count of incidents of violation of local speed limits.

As in the speeding ticket example, it is straightforward to compute these sorts of quantities from the variant of the protocol in which we require repeated use of a vehicle identifier v_i on successive tuples. If only a function

of speed and position is required, in fact the exact framework of the speeding ticket example will suffice.

6 Enforcement

The cryptographic protocol described in Section 5 ensures that a driver cannot lie about the result of the function to be computed given some private inputs to the function (the path tuples). However, when implementing such a protocol in a real setting, we need to ensure that the inputs to the function are correct. For example, the driver can turn off the transponder device on a toll road. The server will have no path tuples from that car on this road. The driver can then successfully participate in the protocol and compute the tolling cost only for the roads where the transponder was on and prove to the server that the cost was "correct".

In this section, we present a general enforcement scheme that deals with security problems of this nature. The enforcement scheme applies to any function computed over a car's path data.

The enforcement scheme needs to be able to detect a variety of driver misbehaviors such as using tags other than the ones committed to during registration, sending incorrect path tuples by modifying the time and location fields, failing to send path tuples, etc. To this end, we employ an end-to-end approach using sporadic *random spot checks*. We assume that at random places on the road, unknown to the drivers, there will be physical observations of a path tuple $\langle$`license plate,time,location`$\rangle$. We show in Section 8 that such spot checks can be infrequent (and thus do not affect driver privacy), while being effective.

The essential point is that the spot check tuples are connected to the car's physical identifier, the license plate. For instance, such a spot check could be performed by secret cameras that are able to take pictures of the license plates. At the end of the day or month, an officer could extract license plate, time and location information or this task could be automated. Alternatively, using the existing surveillance infrastructure, spot checks can be carried out by roving police cars that secretly record the car information. This is similar to today's "speed traps" and the detection probability should be the same for the same number of spot checks.

The data from the spot check is then used to validate the entries in the server database. In the reconciliation phase of the protocol from Section 5, the driver is also required to prove that she uploaded a tuple that is sufficiently close to the one observed during the spot check (and verify that the tag used in this tuple was one of the tags committed to during registration). Precisely, given a spot check tuple (t_c, ℓ_c), the driver must prove she generated a tuple (t, ℓ) such that $|t - t_c| < \Omega_1$ and $|\ell - \ell_c| < (\Omega_2)|t - t_c|$, where Ω_1 is a threshold related to the tuple production frequency and Ω_2 is a threshold related to the maximum rate of travel.

This proof can be performed in zero knowledge, although since the spot check reveals the car's location at that point, this is not necessary. The driver can just present as a proof the tuple it uploaded at that location. If the driver did not upload such a tuple at the server around the observation time and place, she will not be able to claim that another driver's tuple belongs to his due to the commitment check. The server may allow a threshold number of tuples to be missing in the database to make up for accidental errors. Before starting the protocol, a driver can check if all his tuples were received at the server and upload any missing ones.

Intuitively, we consider that the risk of being caught tampering with the protocol is akin to the current risk of being caught driving without a license plate or speeding. It is also from this perspective that we regard the privacy violation associated with the spot check method: the augmented protocol by construction reveals the location of the car at the spot check points. However, as we will show in Section 8, the number of spot checks needed to detect misbehaving drivers with high probability is very small. This means that the privacy violation is limited, and the burden on the server (or rather, whoever runs the server) of doing the spot checks is manageable.

The spot check enforcement is feasible for organizations that can afford widespread deployment of such spot checks; in practice, this would be restricted principally to governmental entities. For some applications such as insurance protocols, this assumption is unrealistic (although depending on the nature of insurance regulation in the region in question it may be the case that insurance companies could benefit from governmental infrastructure).

In this case, the protocol can be enforced by requiring auditable tamper-evident transponders. The transponder should run correctly the driving phase with tuples from registration. Correctness during the reconciliation phase is ensured by the cryptographic protocol. The insurance company can periodically check if the transponder has been tampered with (and penalize the driver if necessary). To handle the fact that the driver can temporarily disable or remove the transponder, the insurance company can check the mileage recorded by the transponder against that of the odometer, for example during annual state inspections.

7 Implementation

We implemented the road pricing protocol in C++ (577 lines on the server side and 582 on the client side). It consists of two modules, the client and the server. The

source code is available at `http://cartel.csail.mit.edu/#vpriv`. We implemented the tolling protocol from Figure 3, where we used the Pedersen commitment scheme [29] and the random function family in [27], and a typical security parameter (key size) of 128 bits (for more security, one could use a larger key size although considering the large number of commitments produced by the client, breaking a significant fraction of them is unlikely). The implementation runs the registration and reconciliation phases one after the other for one client and the server. Note that the protocol for each client is independent of the one for any other client so a logical server (which can be formed of multi-core or multiple commodity machines) could run the protocol for multiple clients in parallel.

7.1 Downloading a subset of the server's database

In the protocols described above, the client downloads the entire set of tags (along with their associated costs) from the server. When there are many clients and correspondingly the set of tags is large, this might impose unreasonable costs in terms of bandwidth and running time. In this section we discuss variants of the protocol in which these costs are reduced, at some loss of privacy.

Specifically, making a client's tags unknown among the tags of all users may not be necessary. For example, one might decide that a client's privacy would still be adequately protected if her tags cannot be distinguished in a collection of one thousand other clients' tags. Using this observation, we can trade off privacy for improved performance.

In the revised protocol, the client downloads only a subset of the total list of tags. For correctness, the client needs to prove that all of her tags are among the ones downloaded. Let the number of encrypted tags provided to the server during registration be n; the first $m \leq n$ of these tags have been used in the last reconciliation period. Assume the driver informs the server of m. Any misreporting regarding m can be discovered by the enforcement scheme (because any tags committed to during registration but not included in the first m will not verify the spot check). When step (iv) is executed for $b = 1$, the server also checks that all the first m tuples are included in the set s_i; that is $\{f_k(v_i)|i \leq m\} \in \{f_k(s_j)|\forall j\}$.

There are many ways in which the client could specify the subset of tags to download from the server. For instance, one way is to ask the server for some ranges of tags. For example, if the field of tags is between 0 and $(2^{128}-1)/2^{128}$, and the client has a tag of value around 0.5673, she can ask for all the tuples with tags in the range $[0.5672, 0.5674]$. The client can ask for an interval for each of her tags as well as for some junk intervals. The client's tag should be in a random position in the requested interval. Provided that the car tags are random, in an interval of length ΔI, if there are $total$ tags, there will be about $\Delta I \cdot total$ tags.

Alternatively, during registration clients could be assigned random "tag subsets" which are then subsequently used to download clusters of tags; the number of clients per tag subset can be adjusted to achieve the desired efficiency/ privacy characteristics. The tag subset could be enforced by having the clients pick random tags with a certain prefix. Clients living in the same area would belong to the same tag subset. In this way, a driver's privacy comes from the fact that the server will not know whether the driver's tuples belong to him or to any other driver from that region (beyond any side information).

8 Evaluation

In this section we evaluate the protocols proposed. We first evaluate the implementation of the road pricing protocol. We then analyze the effectiveness of the enforcement scheme using theoretical analysis in Section 8.3.1 and with real data traces in Section 8.3.2.

We evaluated the C++ implementation by varying the number of random vehicle tags, the total number of tags seen at the server, and the number of rounds. In a real setting, these numbers will depend on the duration of the reconciliation period and the desired probability of detecting a misbehaving client. We pick random tags seen by the server and associate random costs with them. In our experiments, the server and the clients are located on the same computer, so network delays are not considered or evaluated. We believe that the network delay should not be an overhead because we can see that there are about two round trips per round. Also, the number of tuples downloaded by a client from the server should be reasonable because the client only downloads a subset of these tuples as discussed in Section 7. We are concerned primarily with measuring the cryptographic overhead.

8.1 Execution time

Figures 4, 5, and 6 show the performance results on a dual-core processor with 2.0 GHz and 1 GByte of RAM. Memory usage was rarely above 1%. The execution time for a challenge bit of 0 was typically twice as long as the one for a challenge type of 1. The running time reported is the total of the registration and reconciliation times for the server and client, averaged over multiple runs.

The graphs show an approximately linear dependency of the execution time on the parameters chosen. This

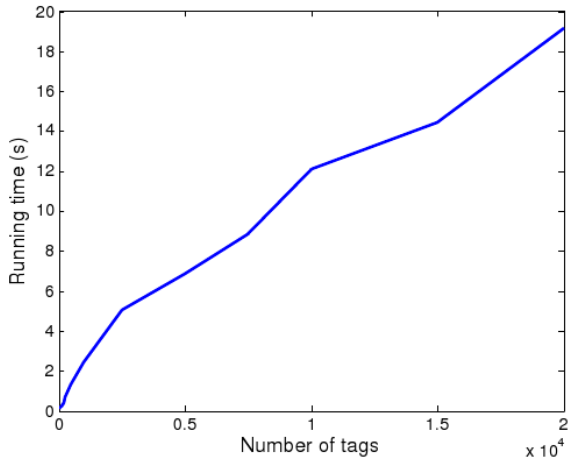


Figure 4: The running time of the road pricing protocol as a function of the number of tags generated during registration for one round.

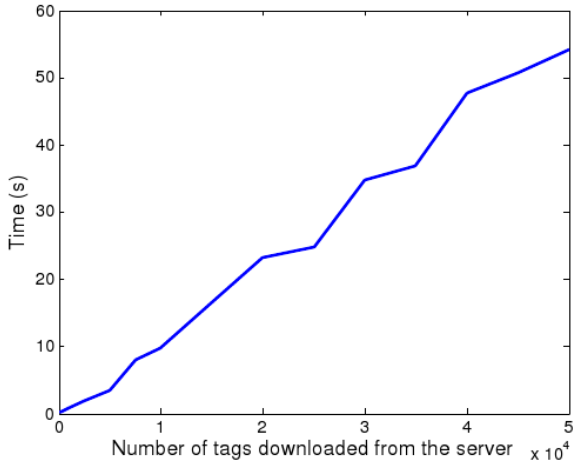


Figure 5: The running time of the road pricing protocol as a function of the number of tuples downloaded from the server during the reconciliation phase for one round.

result makes sense because all the steps of the protocol have linear complexity in these parameters.

In our experiments, we generated a random tag on average once every minute, using that tag for all the tuples collected during that minute. This interval is adjustable; the 1 minute seems reasonable given the 43 MPH average speed [28]. The average number of miles per car per year in the US is $14,500$ miles and 55 min per day ([28]), which means that each month sees about ≈ 28 hours of driving per car. Picking a new tag once per minute leads to $28 \times 60 = 1680$ tags per car per month (one month is the reconciliation period that makes sense for our applications). So a car will use about 2000 tags per month.

We consider that downloading $10,000$ tuples from the server offers good privacy, while increasing efficiency (note that these are only tuples with non-zero tolling cost). The reason is as follows. A person roughly drives through less than 50 toll roads per month. Assuming no side channels, the probability of guessing which tuples

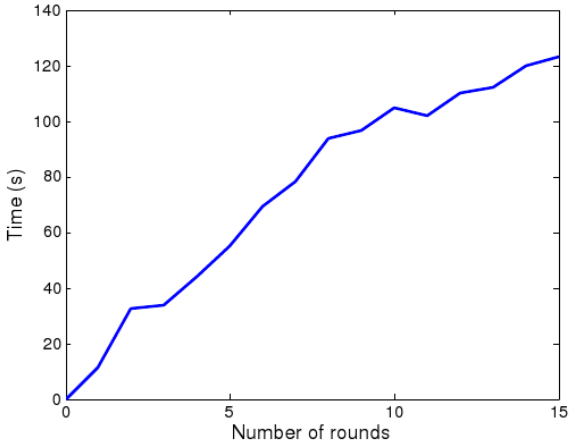


Figure 6: The running time of the road pricing protocol as a function of the number of rounds used in the protocol. The number of tags the car uses is 2000 and the number of tuples downloaded from the server is 10000.

belong to a car in this setting is $1/\binom{10000}{50}$, which is very small. Even if some of the traffic patterns of some drivers are known, the 50 tuples of the driver would be mixed in with the other 10000.

If the protocol uses 10 rounds (corresponding to a detection probability of 99.9%), the running time will be about $10 \cdot 10 = 100$ seconds, according to Figure 6. This is a very reasonable latency for a task that is done once per month and it is orders of magnitude less than the latency of the generic protocol [2] evaluated below. The server's work is typically less than half of the aggregate work, that is, 50 seconds. Downloading $10,000$ tuples (each about 50 bytes) at a rate of $10Mb/s$ yields an additional delay of 4 seconds. Therefore, one similar core could handle 30 days per month times 86400 seconds per day divided by 54 seconds per car = 51840 cars per month. Even if bandwidth does not scale linearly with the number of cores, the latency due to bandwidth utilization is still one order of magnitude less than the one for computation; even if it adds up and cannot be parallelized, the needed number of cores is still within the same order of magnitude. Also, several computers can be placed in different parts of the network in order to parallelize the use of wide-area bandwidth. Since the downloaded content for drivers in the same area is the same, a proxy in certain regions will decrease bandwidth usage significantly. Hence, for 1 million cars, one needs $10^6/51840 \approx 21 < 30$ similar cores; this computation suggests our protocol is feasible for real deployment. (We assumed equal workloads per core because each core serves about 50000 users so the variance among cores is made small.)

8.2 Comparison to Fairplay

Fairplay [26] is a general-purpose compiler for producing secure two-party protocols that implement arbitrary functions. It generates circuits using Yao's classic work on secure two-party computation [34]. We implemented a simplified version of the tolling protocol in Fairplay. The driver has a set of tuples and the server simply computes the sum of the costs of some of these tuples. We made such simplifications because the Fairplay protocol was prohibitively slow with a more similar protocol to ours. Also, in our implementation, the Fairplay server has no private state (to match our setting in which the private state is only on the client). We found that the performance and resource consumption of Fairplay were untenable for very small-sized instances of this problem. The Fairplay program ran out of 1 GB of heap space for a server database of only 75 tags, and compiling and running the protocol in such a case required over 5 minutes. In comparison, our protocol runs with about $10,000$ tuples downloaded from the server in 100s, which yields a difference in performance of *three orders of magnitude*. In addition, the oblivious circuit generated in this case was over 5 MB, and the scaling (both for memory and latency) appeared to be worse than linear in the number of tuples. There have been various refinements to aspects of Fairplay since its introduction which significantly improve its performance and bandwidth requirements; notably, the use of ordered binary decision diagrams [23]. However, the performance improvements associated with this work are less than an order of magnitude at best, and so do not substantially change the general conclusion that the general-purpose implementation of the relevant protocol is orders of magnitude slower than VPriv. This unfeasibility of using existing general frameworks required us to invent our own protocol for cost functions over path tuples that is efficient and provides the same security guarantees as the general protocols.

8.3 Enforcement effectiveness

We now analyze the effectiveness of the enforcement scheme both analytically and using trace-driven experiments. We would like to show that the time a motorist can drive illegally and the number of required spot checks are small. We will see that the probability to detect a misbehaving driver grows exponentially in the number of spot checks, making the number of spot checks logarithmic in the desired detection probability. This result is attractive from the dual perspectives of implementation cost and privacy preservation.

8.3.1 Analytical evaluation

We perform a probabilistic analysis of the time a motorist can drive illegally as well as the number of spot checks required. Let p be the probability that a driver undergoes a spot check in a one-minute interval (or similarly, driving through a segment). Let m be the number of minutes until a driver is detected with a desired probability. The number of spot checks a driver undergoes is a binomial random variable with parameters (p, m), pm being its expected value.

The probability that a misbehaving driver undergoes at least one spot check in m minutes is

$$\Pr[\text{spot check}] = 1 - (1-p)^m. \qquad (1)$$

Figure 7 shows the number of minutes a misbehaving driver will be able to drive before it will be observed with high probability. This time decreases exponentially in the probability of a spot check in each minute. Take the example of $p = 1/500$. In this case, each car has an expected time of 500 minutes (8.3h) of driving until it undergoes a spot check and will be observed with 95% probability after about 598 min (< 10 hours) of driving, which means that overwhelmingly likely the driver will not be able to complete a driving period of a month without being detected.

However, a practical application does not need to ensure that cars upload tuples on all the roads. In the road pricing example, it is only necessary to ensure that cars upload tuples on toll roads. Since the number of toll points is usually only a fraction of all the roads, a much smaller number of spot checks will suffice. For example, if we have a spot check at one tenth of the tolling roads, after 29 minutes, each driver will undergo a spot check with 95% probability.

Furthermore, if the penalty for failing the spot check test is high, a small number of spot checks would suffice because even a small probability of detecting each driver would eliminate the incentive to cheat for many drivers. In order to ensure compliance by rational agents, we simply need to ensure that the penalty associated with noncompliance, β, is such that $\beta(\Pr[\text{penalization}]) > \alpha$, where α is the total toll that could possibly be accumulated over the time period. Of course, evidence from randomized law enforcement suggests strongly that independent of β, $\Pr[\text{penalization}]$ needs to be appreciable (that is, a driver must have confidence that they *will* be caught if they persist in flouting the compliance requirements) [8].

If there is concern about the possibility of tuples lost in transit from client to server, our protocol can be augmented with an anonymized interaction in which a client checks to see if all of her tuples are included in the server's database (the client can perform this check af-

ter downloading the desired tuples from the server and before the spot check reconciliation and zero-knowledge protocol). Alternatively, the client might simply blindly upload duplicates of all her tuples at various points throughout the month to ensure redundant inclusion in the database. Note that it is essential that this interaction should be desynchronized from the reconciliation process in order to prevent linkage and associated privacy violation.

Nevertheless, even if we allow for a threshold t of tuples to be lost before penalizing a driver, the probability of detection is still exponential in the driving time $1 - \sum_{i=0}^{t} \binom{m}{i} p^i (1-p)^{m-i} \geq 1 - e^{\frac{-(t-mp)^2}{2mp}}$, where the last inequality uses Chernoff bounds.

8.3.2 Experimental evaluation

We now evaluate the effectiveness of the enforcement scheme using a trace-driven experimental evaluation. We obtained real traces from the CarTel project testbed [20], containing the paths of 27 limousine drivers mostly in the Boston area, though extending to other MA, NH, RI, and CT areas, during a one-year period (2008). Each car drives many hours every day. The cars carry GPS sensors that record location and time. We match the locations against the Navteq map database. The traces consist of tuples of the form (car tag, segment tag, time) generated at intervals with a mean of 20 seconds. Each segment represents a continuous piece of road between two intersections (one road usually consists of many segments).

We model each spot check as being performed by a police car standing by the side of a road segment. The idea is to place such police cars on certain road segments, to replay the traces, and verify how many cars would be spot-checked.

We do not claim that our data is representative of the driving patterns of most motorists. However, these are the best real data traces we could obtain with driver, time, and location information. We believe that such data is still informative; one might argue that a limousine's path is an aggregation of the paths of the different individuals that took the vehicles in one day.

It is important to place spot checks randomly to prevent misbehaving drivers from knowing the location of the spot checks and consequently to behave correctly only in that area. One solution is to examine traffic patterns and to determine the most frequently travelled roads. Then, spot checks would be placed with higher probability on popular roads and with lower probability on less popular roads. This scheme may not observe a malicious client driving through very sparsely travelled places; however, such clients may spend fuel and time resources by driving through these roads and which most likely do not even have tolls. More sophisticated placement schemes are possible; here, we are primarily concerned with showing the ability to observe most traffic with remarkably few spot checks.

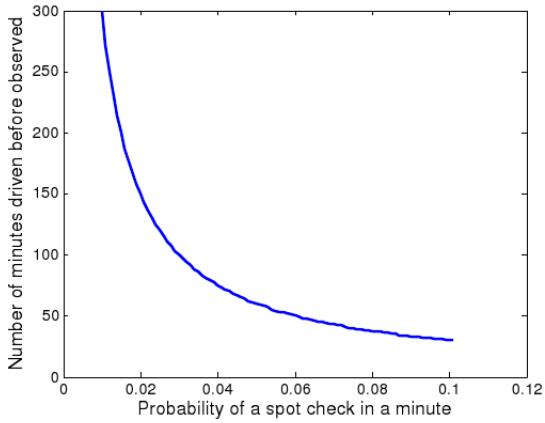


Figure 7: The time a motorist can drive illegally before it undergoes a spot check with a probability 95% for various values of p, the probability a driver undergoes a spot check in a minute.

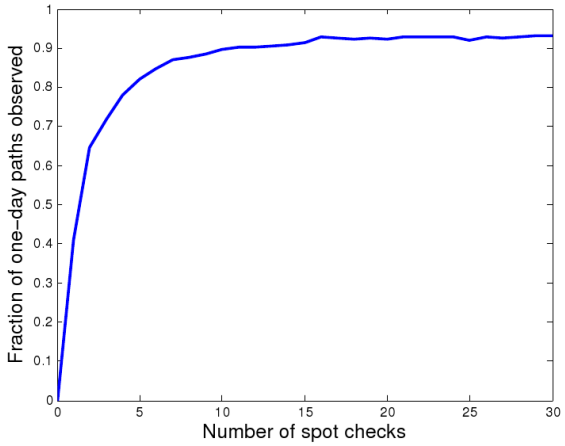


Figure 8: The fraction of one-day paths observed out of a total of 4826 one-day paths as a function of the total number of police cars placed.

Consider the following experiment: we use the traces from a month as a training phase and the traces from the next month as a testing phase, for each month except for the last one. The first month is used to determine the first 1% (≈ 300) popular sites. We choose an increasing number of police cars to be placed randomly at some of these sites. Then, in the testing phase we examine how many drivers are observed in the next month. We perform this experiment for an increasing number of police cars and for each experiment we average the results over fifty runs. In order to have a large sample, we consider the paths of a driver in two different days as the paths of two different drivers. This yields 4826 different one-day traces.

Figure 8 illustrates the data obtained. In few places, the graph is not perfectly monotonic and this is due to randomization: we are placing few spot checks in some

of the 300 locations. Even if in some cases we place a spot check more than in others, due to randomization, the spot checks may be placed in an unfavorable position and observe less paths. The reason is that the 300 spot check vary significantly in popularity. From the shape of the graph, we can see that the fraction of paths observed increases very fast at the beginning; this is explained by the exponential behavior discussed in Section 8.3.1. After 10 spot checks have been placed, the fraction of paths observed grows much slower. This is because we are only placing spot checks at 1% of the segments traveled by the limousine drivers. Some one-day paths may not be included at all in this set of paths. Overall, we can see that this algorithm requires a relatively small number of police cars, namely 20, to observe $\approx 90\%$ of the 4826 one-day paths.

Our data unfortunately does not reflect the paths of the entire population of a city and we could not find such extensive trace data. A natural question to ask would be how many police cars would be needed for a large city. We speculate that this number is larger than the number of drivers by a sublinear factor in the size of the population; according to the discussion in Section 8.3.1, the number of spot checks increases logarithmically in the probability of detection of each driver and thus the percentage of drivers observed.

9 Security analysis

In this section, we discuss the resistance of our protocol to the various attacks outlined in Section 3.2.

Client and intermediate router attacks. Provided that the client's tuples are successfully and honestly uploaded at the server, the analysis of Section 5 shows that the client cannot cheat about the result of the function. To ensure that the tuples arrive uncorrupted, the client should encrypt tuples with the public key of the server. To deal with dropped or forged tuples, the drivers should make sure that all their tuples are included in the subset of tuples downloaded from the server during the function computation. If some tuples are missing, the client can upload them to the server. These measures overcome any misbehavior on the part of intermediate routers.

The spot check method (backed with an appropriate penalty) is a strong disincentive for client misbehavior. An attractive feature of the spot check scheme is that it protects against attacks involving bad tuple uploads by drivers. For example, drivers cannot turn off their transponders because they will fail the spot check test; they will not be able to provide a consistent tuple. Similarly, drivers cannot use invalid tags (synthetic or copied from another driver), because the client will then not pass the spot checks; the driver did not commit to such tags during registration.

If two drivers agree to use the same tags (and commit to them in registration), they will both be responsible for the result of the function (i.e., they will pay the sum of the tolling amounts for both of them).

Server misbehavior. Provided that the server honestly carries out the protocol, the analysis of Section 5 shows that it cannot obtain any additional information from the cryptographic protocol. A concern could be that the server attempts to track the tuples a car sends by using network information (e.g., IP address). Well-studied solutions from the network privacy and anonymization literature can be used here, such as Tor [7], or onion routing [11]. The client can avoid any timing coincidence by sending these tuples in separate packets (perhaps even at some intervals of time) towards the end of the driving period, when other people are sending such tuples.

Another issue is the presence of side channels in the anonymized tuple database. As discussed in Section 2, a number of papers have demonstrated that in low-density regions it is possible to reconstruct paths with some accuracy from anonymized traces [18, 22, 16]. As formalized in Definition 1, our goal in this paper was to present a protocol that avoids leaking any additional information beyond what can be deduced from the anonymized database. The obvious way to prevent this kind of attack is to restrict the protocol so that tuples are uploaded (and spot checks are conducted) only in areas of high traffic density. An excellent framework for analyzing potential privacy violations has been developed in [19, 17], which use a *time to confusion* metric that measures how long it takes an identified vehicle to mix back into traffic. In [17], this is used to design traffic information upload protocols with exclusion areas and spacing constraints so as to reduce location privacy loss.

Recall that in Section 5, we assumed that the server is a passive adversary: it is trusted not to change the result of the function, although it tries to obtain private information. A malicious server might dishonestly provide tuples to the driver or compute the function f wrongly. With a few changes to the protocol, however, VPriv can be made resilient to such attacks.

- The function f is made public. In Figure 3, step 3a), the server computes the tolls associated to each tuple. A malicious server can attach any cost to each tuple, and to counteract this, we require that the tolling function is public. Thus, the client can compute the cost of each tuple in a verifiable way.
- For all the client commitments sent to the server, the client must also provide to the server a signed hash of the ciphertext. This will prevent the server from changing the client's ciphertext because he cannot forge the client's signature.
- When the server sends the client the subset of tuples in Step 3a, the server needs to send a signed hash of these values as well. Then, the server cannot change

his mind about the tuples provided.

- The server needs to prove to a separate entity that the client misbehaved during enforcement before penalizing it (eg. insurance companies must show the tamper-evident device).

Note that it is very unlikely that the server could drop or modify the tuples of a specific driver because the server does not know which ones belong to the driver and would need to drop or modify a large, detectable number of tuples. If the server rejects the challenge information of the client in Step iv) when it is correct, then the client can prove to another person that its response to the challenge is correct.

10 Conclusion

In this paper, we presented VPriv, a practical system to protect a driver's location privacy while efficiently supporting a range of location-based vehicular services. VPriv combined cryptographic protocols to protect the location privacy of the driver with a spot check enforcement method. A central focus of our work was to ensure that VPriv satisfies pragmatic goals: we wanted VPriv to be efficient enough to run on stock hardware, to be sufficiently flexible so as to support a variety of location-based applications, to be implementable with many different physical setups. and to resist a wide array of physical attacks. We verified through analytical results and simulation using real vehicular data that VPriv realized these goals.

Acknowledgments. This work was supported in part by the National Science Foundation under grants CNS-0716273 and CNS-0520032. We thank the members of the CarTel project, especially Jakob Eriksson, Sejoon Lim, and Sam Madden for the vehicle traces, and Seth Riney of PlanetTran. David Andersen, Sharon Goldberg, Ramki Gummadi, Sachin Katti, Petros Maniatis, and the members of the PMG group at MIT have provided many useful comments on this paper. We thank Robin Chase and Roy Russell for helpful discussions.

References

[1] BANGERTER, E., CAMENISCH, J., AND LYSYANSKAYA, A. A cryptographic framework for the controlled release of certified data. In *Security Protocols Workshop* (2004).

[2] BLUMBERG, A., AND CHASE, R. Congestion pricing that respects "driver privacy". In *ITSC* (2005).

[3] BLUMBERG, A., KEELER, L., AND SHELAT, A. Automated traffic enforcement which respects driver privacy. In *ITSC* (2004).

[4] BRASSARD, G., CHAUM, D., AND CREPEAU, C. Minimum disclosure proofs of knowledge. In *JCSS, 37, pp. 156-189* (1988).

[5] CAMENISCH, J., HOHENBERGER, S., AND LYSYANSKAYA, A. Balancing accountability and privacy using e-cash. In *SCN* (2006).

[6] CHAUM, D. Security without identification: transaction systems to make big brother obsolete. In *CACM 28(10)* (1985).

[7] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Sec. Symp., USENIX Association* (2004).

[8] EIDE, E., RUBIN, P. H., AND SHEPHERD, J. *Economics of crime*. Now Publishers, 2006.

[9] ERIKSSON, J., BALAKRISHNAN, H., AND MADDEN, S. Cabernet: Vehicular content delivery using wifi. In *MOBICOM* (2008).

[10] GEDIK, B., AND LIU, L. Location privacy in mobile systems: A personalized anonymization model. In *25th IEEE ICDCS* (2005).

[11] GOLDSCHLAG, D., REED, M., AND SYVERSON, P. Onion routing for anonymous and private internet connections. In *CACM, 42(2)* (1999).

[12] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems. In *Proceedings of 17th Symposium on the Theory of Computation, Providence, Rhode Island.* (1985).

[13] GOODIN, D. Microscope-wielding boffins crack tube smartcard.

[14] GROUP, E.-Z. I. E-zpass.

[15] GRUTESER, M., AND GRUNWALD, D. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM MobiSys* (2003).

[16] GRUTESER, M., AND HOH, B. On the anonymity of periodic location samples. In *Pervasive* (2005).

[17] HOH, B., GRUTESER, M., HERRING, R., BAN, J., WORK, D., HERRERA, J.-C., BAYEN, A., ANNAVARAM, M., AND JACOBSON, Q. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Mobisys* (2008).

[18] HOH, B., GRUTESER, M., XIONG, H., AND ALRABADY, A. Enhancing security and privacy in trafc-monitoring systems. In *IEEE Pervasive Computing, 5(4):38-46* (2006).

[19] HOH, B., GRUTESER, M., XIONG, H., AND ALRABADY, A. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *ACM CCS* (2007).

[20] HULL, B., BYCHKOVSKY, V., CHEN, K., GORACZKO, M., MIU, A., SHIH, E., ZHANG, Y., BALAKRISHNAN, H., AND MADDEN, S. Cartel: A distributed mobile sensor computing system. In *ACM SenSys* (2006).

[21] INSURANCE, A. Mile meter.

[22] KRUMM, J. Inference attacks on location tracks. In *Pervasive* (2007).

[23] L. KRUGER, E. GOH, S. J., AND BONEH, D. Secure function evaluation with ordered binary decision diagrams. In *ACM CCS* (2006).

[24] LITMAN, T. London congestion pricing, 2006.

[25] LYSYANSKAYA, A., RIVEST, R., SAHAI, A., , AND WOLF, S. *Pseudonym systems*. Springer, 2000.

[26] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay - a secure two-party computation system. In *USENIX Sec. Symp., USENIX Association* (2004).

[27] NAOR, M., AND REINGOLD, O. Number-theoretic constructions of efficient pseudo-random functions. In *Journal of the ACM, Volume 51, Issue 2, p. 231-262* (March 2004).

[28] OF TRANSPORTATION STATISTICS, B. National household travel survey daily travel quick facts.

[29] PEDERSEN, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Springer-Verlag* (1998).

[30] RASS, S., FUCHS, S., SCHAFFER, M., AND KYAMAKYA, K. How to protect privacy in floating car data systems. In *Proceedings of the fifth ACM international workshop on VehiculAr Inter-NETworking* (2008).

[31] RILEY, P. The tolls of privacy: An underestimated roadblock for electronic toll collection usage. In *Third International Conference on Legal, Security + Privacy Issues in IT* (2008).

[32] SALLADAY, R. Dmv chief backs tax by mile. In *Los Angeles Times* (November 16, 2004).

[33] SWEENEY, L. k-anonymity: A model for protecting privacy. In *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems v.10 n.5* (2002).

[34] YAO, A. C. Protocols for secure computations (extended abstract). In *FOCS* (1982: 160-164).