Mitigating Risk while Complying with Data Retention Laws

Luis Vargas
University of Florida
Gainesville, Florida
lfvargas14@ufl.edu

Gyan Hazarika
University of Florida
Gainesville, Florida
ghazarika@ufl.edu

Rachel Culpepper
University of Richmond
Richmond, Virginia
rachel.culpepper@richmond.edu

Kevin R.B. Butler
University of Florida
Gainesville, Florida
butler@ufl.edu

Thomas Shrimpton
University of Florida
Gainesville, Florida
teshrim@ufl.edu

Doug Szajda
University of Richmond
Richmond, Virginia
dszajda@richmond.edu

Patrick Traynor
University of Florida
Gainesville, Florida
traynor@ufl.edu

ABSTRACT

Data breaches represent a significant threat to organizations. While the general problem of protecting data has received much attention, one large (and growing) class has not – data that must be kept due to mandatory retention laws. Such data is often of little use to an organization, is rarely accessed, and represents a significant potential liability, yet cannot be discarded. Protecting such data entails an unusual combination of practical constraints (such as providing verification to a party that may be unknown) and thus requires functionality that is not well addressed by traditional cryptographic primitives. We propose to mitigate the risk to such data through a new system called *Dragchute*, which creates a time window during which locked data cannot be accessed by *anyone*. Based on a verifiable non-interactive, non-parallelizable, time-delay key escrow mechanism, Dragchute is novel in that it *requires* that no cryptographic material capable of providing early access to the data be retained, yet provides verification for multiple properties. We define a base construction for Dragchute, show possible extensions that help meet additional verification requirements, and characterize its performance. Our results show that Dragchute systems offer verifiable, customizable, computational protection against data exposure for encryption costs similar to traditional methods (e.g., less than 6% overhead compared to AEAD). We thus show that Dragchute systems provide a critical new means for protecting data that must be retained long term due to mandatory retention laws.

CCS CONCEPTS

• **Security and privacy → Cryptography**; **Database and storage security**;

KEYWORDS

Applied Cryptography; Time-Lock Cryptography; Data Breaches

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243800

ACM Reference Format:
Luis Vargas, Gyan Hazarika, Rachel Culpepper, Kevin R.B. Butler, Thomas Shrimpton, Doug Szajda, and Patrick Traynor. 2018. Mitigating Risk while Complying with Data Retention Laws. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243800

1 INTRODUCTION

Large scale data breaches have become regular events. Incidents ranging from recent attacks on Target [42] and Equifax [24] to the theft of millions of current and former employee profiles from the US Office of Personnel Management (OPM) [54] demonstrate that even large organizations with professional IT security staff remain susceptible to compromise. Such breaches have resulted in billions of dollars of losses to the compromised entities [39] and also caused losses to the consumers and employees whose data was stolen.

Though the general problem of breaches has received much consideration (e.g., papers on access control [11, 63] and database encryption [29, 68]), one large and important class of data has received relatively little — data that is rarely accessed and has little utility to the organizations that own it, but must be retained due to mandatory data retention laws. For example, regulations such as the Sarbanes-Oxley Act [2] require publicly traded companies to retain vast amounts of data for years; effectively in perpetuity, in some cases. Protecting such data requires functionality that is not well addressed by traditional cryptographic primitives.

As an example, advanced persistent threats (APT) raise the possibility that an adversary capable of accessing the data may also compromise the machines on which secret keys are stored. This would render protected data immediately recoverable. Thus, desirable mechanisms for our setting should store no secrets that would lead to rapid exposure of plaintext data following a breach.

Mandatory retention laws may also require organizations to provide evidence of "good-faith" effort to comply. This evidence should sufficiently convince regulators that (minimally) the privacy of the data is maintained, that the underlying plaintext is authentic, and that an authorized party will be able to obtain this plaintext within a reasonable period of time after it is requested. Desirable mechanisms should offer such evidence, preferably in a timely fashion, and without having to fully expose the plaintext data to machines that may unknowingly be compromised.

Compliance with data retention laws may further require that the plaintext data be made available to multiple authorized parties

at some point(s) in the future (e.g., courts in various jurisdictions), and these parties may not be explicitly named at the time the data is placed in storage. Thus, desirable mechanisms should not assume knowledge of recipient identities, or require interactive setup ceremonies between the data-holder and the authorized recipients. In fact, for operational and legal reasons, it would be optimal if no interaction was required beyond the authorized party requesting the data, and the data-holder handing it over.

Furthermore, any mechanism employed to comply with data-retention laws should rely minimally, if at all, on trusted third parties. As the subject data must often be stored for years, reliance on third parties raises serious concerns about the longevity and stability of these parties, as well as a commitment to long-term trust. Use of third parties also introduces complications with respect to liability. Lastly, employing third parties, e.g., for key escrow, raises considerable privacy concerns [5].

Atfi rst glance, existing cryptographic primitives such as time-lock cryptography [14, 18, 23, 53, 61], secret sharing [66], and encapsulated key escrow [8, 9, 49], appear promising. But none meet *all* of the requirements mentioned above. Time-lock cryptographic primitives can be used to prevent immediate access to data for an adversary who has obtained the necessary keys, but they either lack verification mechanisms or provide verification that is limited in scope and requires significant interaction. Secret sharing can make the acquisition of keys more difficult for the adversary, but if the threshold number of shares is stored in-house, then this provides no additional protection; if not, the storing entity is beholden to external share holders and third-party concerns return.

Encapsulated key escrow (EKE) techniques, in which keys are stored in a manner such that they cannot be accessed until a specified future time, appear especially promising. These schemes allow for the escrow of keys with a variety of levels and types of verification. Unfortunately, many EKE protocols require interaction, or significant third-party intervention, as part of their verification procedures. More concerning, many are vulnerable to parallel attacks via well-resourced adversaries [8, 9].

Meeting the Challenges. We propose, instantiate, and test a new encapsulated key-escrow system that stores no secrets, resists parallel attacks, and provides evidence of multiple aspects of compliance. Moreover, most of this evidence can be provided non-interactively. We call our specific realization of these ideas the *Dragchute* system.[1] Unlike traditional encryption-based protection mechanisms, where an information asymmetry is assumed for security (specifically, knowledge of cryptographic keys), security of Dragchute rests solely on the "hardness" of time. Access to data protected with our system is delayed *for all parties* for a specifiable amount of time, regardless of the (realistic) computing resources available. A practical impact of this is to allow the compromised party a window of opportunity to proactively mitigate the damage that might be caused by the data being divulged. Moreover, it slows an adversary's ability to comb through massive amounts of data. We stress that because time, rather than information, is required to decrypt a Dragchute-protected ciphertext, the system ensures that the adversary will not be able to quickly access the underlying plaintext *regardless of the data the adversary may obtain* during a breach.

Built in large part from primitives that have well understood security properties, the system is supported by an intuitively understandable argument regarding how it protects data. It admits non-interactive verification on multiple dimensions that is resistant to parallel attacks and requires minimal, and in many cases no, third party intervention. Our base scheme guarantees the privacy and integrity of data integrity of the supporting cryptographic structures, and provides assurance that these structures have been correctly constructed and are bound to specific ciphertext(s). It does not, however, provide protection against an adversary that completely replaces ciphertext and the data structures that escrow the key(s) with its own "honestly produced" ciphertext and data structures.

Should one require this additional level of protection, we provide an extension of the base protocol that does protect against such an adversary. We also provide an extension that allows one to prove, in succinct non-interactive zero-knowledge, that the escrow data contains the keys required to unlock the corresponding ciphertext. These extensions are orthogonal in that they can be used by themselves or in tandem as context may require.

Our Contributions To summarize, the specific contributions of this paper are:

- **Introduction of Dragchute:** We describe a new means for protecting data at rest. Dragchute systems are designed to protect data for a set period of time post-breach, thereby increasing the time between breach events and data compromise.
- **Evaluation of a Candidate Dragchute Construction:** We design, implement, and conduct a performance analysis on a proof-of-concept system for parallel-attack-resistant encapsulated key escrow with non-interactive verifiability. Our system uses time-locked commitments [14] as a core component. However, we assume that any and all information that may lead to earlier-than-planned opening of the commitment *by any party*, is destroyed immediate after the data is stored.
  Our analysis, conducted using a total of 686 days of compute time, demonstrates that Dragchute ciphertexts requiring approximately one month to decrypt can be achieved at roughly the cost of executing a standard Authenticated Encryption with Associated Data (AEAD) scheme (less than 6% overhead).
- **Non-Interactive Zero-Knowledge Proof and Compliance:** We provide a protocol by which a party implementing our construction can, *without interaction*, provide evidence of compliance (as opposed to obstruction) with legal requests for protected data without storing any long term secrets that would compromise the data itself.

We note that Dragchute systems are widely applicable — according to a 2011 report by the McKinsey Institute [48], rougly 27% of the data stored by businesses in 2009 is ascribed to sectors significantly impacted by data retention laws. These include the banking,fi nancial services, medical, legal, and professional services industries. Smallerfi rms (e.g., solo tax, law, and accounting firms) in particular face significant hardship in meeting data retention mandates. Dragchute may also be useful in protecting data on non-fixed assets (e.g., laptops, USB drives and other backups). This

[1] The name is meant to be evocative of a device used to slow down very fast-moving objects.

is especially true against adversaries who may be able to extract encryption keys given physical access to compromised devices [41].

The remainder of this paper is organized as follows: In Section 2, we provide an overview of related work; Section 3 provides our adversarial model and assumptions; Section 4 provides an overview of Dragchute; Section 5 describes our core operations; Section 6 shows how we embody a Dragchute scheme and details our candidate construction; Section 7 offers an implementation of our proposed system and a performance analysis; Section 8 discusses data retention laws; and Section 9 provides concluding remarks.

2 RELATED WORK

Data breaches cause significant damages [56]. This incentivizes both commercial and government entities to develop methods for protecting data and amortizing potential losses [69, 73]. Access control is one such approach [11, 38, 63]. Though useful, unforeseen attackers (e.g., insiders, outsiders with knowledge of an exploitable vulnerability) are able to bypass such protection measures.

Another class of solution involves encryption followed by mechanisms for efficiently accessing the encrypted data (e.g., [7, 29, 67, 68]). For example, Song et. al. [68] provide probabilistic algorithms for searching encrypted data. Their methods allow for hidden searches and query isolation, and provide provable secrecy. However, as with any traditional encryption scheme, data can be compromised if an adversary gains access to stored secret keys[2]. Moreover, even if such keys are protected via physical or logical isolation, using the keys often requires that they be handled (briefly) in the clear, creating a window of vulnerability. Our solution avoids these problems by requiring that no key material be retained.

Other solutions attempt to securely protect (or delete) data by splitting (e.g., secret sharing), where data or keys are divided into blocks and dispersed throughout the network or storage medium [70, 71, 78]. Data is recovered by obtaining a sufficient number of blocks/key shares. The best known of these schemes is the Vanish system [35], which uses ephemerizers [50, 55] and distributed hash tables (DHT) to delete data by distributing shares of an encryption key throughout a peer-to-peer network. The intent is that the natural churn of the table should eventually delete the partial keys over time. While an important step forward, such systems often fail for operational reasons. For instance, adversaries have been able to obtain enough partial keys from the DHT long after the expiration date [77]. Though systems have been built tofi x this problem [34], they are not targeted for data that requires long retention periods.

Researchers have also attempted to delay access to resources using computational puzzles. Sample problem domains include DDoS [20, 25, 44, 75], spam [28], and practical cryptocurriencies [51]. Unfortunately, the often parallelizable nature of these puzzles makes their use in real systems challenging [47]. Efforts to improve the fairness of such puzzles against adversaries with varying compute resources include memory-hard [12] and non-parallelizable [14, 22, 33, 49, 57, 59, 61, 72] puzzles. None of these techniques have been investigated in the context of data breaches, nor in the case where cryptographic keys are compromised by the adversary. Several are, in addition, interactive or rely on a trusted third party. One [59] provides a system whereby only the intended recipient can gain access to the message after solution of the puzzle. None meets all of the requirements necessary for the forced-retention data context.

There is also a large amount of work on key escrow techniques, roughly beginning with the work on encapsulated key escrow by Bellare and Goldwasser [8]. Their core primitive for data escrow involves encrypting data with a standard symmetric cipher, then splitting the key into two portions, one of which is made public. Parties wishing to access the full key must exhaustively search through the keyspace by concatenating potential hidden key portions with the known key portion. Auxiliary information is provided in order to allow the identification of the proper key when tested. The exhaustive search is easily parallelized, and thus not practical in the current setting. Subsequent escrow schemes [9, 17, 18, 23, 26, 43] rely heavily on trusted servers and/or interaction.

[2]There are numerous reports of such instances. [46, 60, 62, 64] are recent examples.

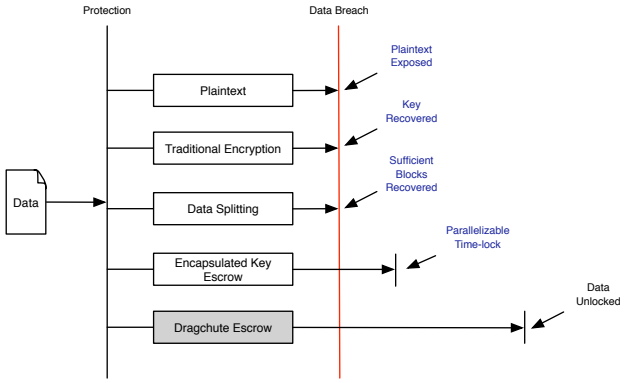

Figure 1: Data can be protected via multiple means. However, multiple operational weaknesses can (and regularly do) render these ineffective. Data protected with Dragchute remains confidential for afi xed time interval even after the data *and keys* have been obtained by the adversary.

3 SECURITY MODEL AND SYSTEM GOALS

The goal of our method is to provide improved protection for historical data that is legally required to be stored, but is of little use to the party storing it. The protection offered by Dragchute systems is agnostic to data type so long that the data itself is rarely accessed, and that when access is necessary, immediate availability is not a requirement. Fundamentally, Dragchute improves protection (either as a stand-alone method, or as augmenting a traditional system) by extending the time between data breach and data access. We thus assume that for the data under consideration there is significant value in this delayed access (e.g., by allowing revocation and/or proactive monitoring).

We assume as well that key management can fail. Figure 1 shows some possible mechanisms to protect a data object stored within a targeted domain (e.g., an enterprise, a government agency, etc.). While an administrator may apply various techniques to protect that data, operational realities may nullify those protections against sophisticated adversaries, e.g., encryption keys are often kept in memory or on disk. Accordingly, our viewpoint is that the adversary can access *all* data held by the target. A Dragchute system is meant

to extend the time between data being captured by the adversary, and the protected content being disclosed to it.

We assume throughout that the party who creates the Dragchute ciphertext and the party to whom the ciphertext is subsequently attested (e.g., a law-enforcement body) are honest. We additionally assume that adversarial parties are able to compromise arbitrary targets via one or more attack vectors (e.g., misconfiguration, vulnerability, insider access) and gain access to that target's sensitive data. We assume an adversary has full knowledge of the schemes protecting the data, and potentially access to substantial computing resources (e.g., supercomputers, botnet). An adversary may be active, attempting to modify or delete ciphertexts and related Dragchute specific data. Dragchute systems are designed to *detect* such activity; the problem of preventing such deletions and modifications is outside the scope of this paper. In any case, Dragchute primitives are designed to fail (and signal failure) when modifications are detected, thereby denying access to protected data.

For simplicity, we assume that all data is leaked as soon as the target is compromised, and that the target itself is only aware of the breach sometime after the data has been stolen. How and when the target becomes aware of the breach is a problem that is orthogonal to the one we consider, falling under the well-studied area of intrusion detection. We assume that appropriate intrusion detection mechanisms will be deployed in conjunction with DE.

The specifics of such measures are determined by administrators of the data, and are of significance to Dragchute systems. In particular, our system is designed to *mitigate*, as opposed to eliminate, the risks associated with forced-retention data. This mitigation takes the form of delayed access to the data for an adversary who manages to breach data storage. Though forced-retention data may need to be stored for years, it is neither practical nor desirable for any mechanism to deny access to it, for all parties, for years at a time. Rather, the system is designed to delay access for a time period chosen at the discretion of the administrator (we envision one or two months). As such, an APT that is able to breach data storage and remain undetected on the system for a period longer than the configured delay may be able to obtain some plaintext data. It is thus crucial that deployed intrusion detection mechanisms are appropriate for the configured Dragchute time delay.

We note also that though a long term APT data can potentially obtain protected data, the amount of such data will be limited, as obtaining larger volumes will require solving several non-parallelizable computation puzzles. In most contexts, the inability for authorized parties to quickly access large volumes of their own data would be problematic. This system is intended, however, to protect data that is stored long-term, of little use to the organization storing it, and expected to be accessed rarely, if at all. Thus the inability to extract large volumes of data quickly is not problematic.

Informally, the goal of a Dragchute system is to provide traditional cryptographic notions of confidentiality, against parallelized and probabilistic adversarial algorithms, *for the period of time required to perform a specific (and configurable) computation.*

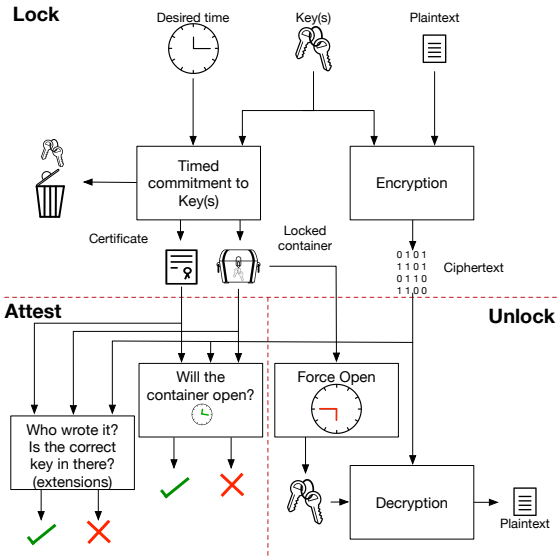

Figure 2: At a high level, Dragchute encrypts afi le, locks information about the key in a time commitment, and then proceeds to discard the key.

4 INTUITION FOR DRAGCHUTE ENCRYPTION

Figure 2 provides a high level overview of our realization of Dragchute. The top portion shows the "Lock" functionality. The plaintext data is encrypted with one or more keys to produce a ciphertext. We use a secure authenticated encryption scheme for this; specifically an encrypt-then-MAC [10, 52] style scheme. We also compute a timed commitment to the encryption keys, outputting a "Locked container" and a "Certificate". *The keys themselves are discarded.* In the base scheme, the Certificate enables efficient attestation (say, to a law-enforcement body) that the container will unlock correctly, reveal exactly what was committed, and corresponds to the ciphertext. Our instantiation of timed-commitment borrows much from a scheme developed by Boneh and Naor [14], in which one commits to a string by XORing it with the tail end of a long, computationally random keystream that is computed by sequential squaring. In the absence of appropriate trap-door information, all of which is discarded (along with the keys) as part of the locking process, it is widely believed that this keystream can only be computed by completing an inherently sequential computation. The desired time required to generate this keysteam, and hence reveal the keys, is an input to the locking function. We stress that all that remains after locking is the ciphertext, the locked container, and the certificate.

The lower-right portion of Figure 2 shows the Unlock functionality. On input of the locked container and the ciphertext, Unlock recovers the encryption keys from the locked container by "forcing" it to open. That is, it carries out the sequential computation to recreate the keystream. Once in possession of the encryption keys, it decrypts the ciphertext to return the original plaintext. Because we use an authenticated encryption scheme in our instantiation, if the recovered keys or the ciphertext have been modified, decryption

will fail and indicate this. Note that since the computation required by Unlock is inherently sequential, even adversaries with substantial access to resources (e.g., botnets, nation-states) will be limited to the compute power of a single core per Unlock instantiation.

The lower-left portion of Figure 2 shows the "Attest" functionality. Provided the certificate, the locked container, and the ciphertext, Attest (quickly) signals whether or not the locked container was correctly constructed and whether it corresponds to the ciphertext. More specifically, our attestation mechanismfi rst verifies, in non-interactive zero-knowledge, that the cipher stream was correctly computed. Thus any committed values, regardless of unlock time, are uniquely determined, in the sense that attempts to later unlock a value different from the one committed will be detected. Attest then checks that a hash of the ciphertext matches a hash that has been committed in the container. In the extended Dragchute schemes detailed below, Attest can additionally verify that the container and ciphertext authenticate, thus detecting adversarial attempts at forging complete (and correct) container/ciphertext pairs, and/or assure quickly, via a zkSNARK [16, 36], that the preimage of a committed hash value is the key that decrypts the corresponding ciphertext. The latter precludes a committer from later claiming that the wrong key was accidentally committed.

We note that even in the augmented Dragchute scheme, there can be no guarantee that the resulting plaintext is *the* information legal authorities might be seeking. Even data retention laws designed to address fraud do not guarantee that the data provided to authorities is what is expected. Rather, they discourage falsifying data via the threat of significantfi nes and/or jail time. For instance, a tax preparer who attests that a document is a client's tax return and then, upon unlocking it, fails to produce that document would likely be held liable for failure to produce the document/perjury. The practical purpose, then, of our attestation mechanism is to provide additional evidence that the committer is assisting, rather than obstructing, in the efforts to obtain the data, and to bind the committer/commit to a specific ciphertext. In a context in which keys are not escrowed, this might be of little value. But the significant delay between when the container/ciphertext pair is presented to the requesting authority, and when the contents of the container are revealed, suggest that quick verification of cooperation is of practical value. Though attestation cannot assure that the plaintext is the requested material, should the (eventually) decrypted ciphertext fail to provide that material, attestation provides authorities with evidence of obstruction. In that sense, it suggests, as well as incentivizes, cooperation, effectively by communicating that the escrowing organization is willing to be "on the hook" for the plaintext protected by the container/ciphertext pair.

5 THE DRAGCHUTE SYSTEM

The central cryptographic primitive of the base Dragchute system is a non-interactive verifiable time-delay key escrow mechanism that requires no trusted third party and admits verification on multiple dimensions. The idea is to use this mechanism to protect ciphertexts generated using a traditional semantically-secure encryption scheme by escrowing the key(s) in such a way that even honest parties must invest considerable work in order to recover them.

Our escrow mechanism consists of four algorithms, Generate, Lock, Unlock, and Attest, whose syntax is as follows.

- The randomized *parameter-generation* algorithm Generate takes as input a security parameter k and returns a pair (pp, sp) of public and secret *locking parameters*. We write $(pp, sp) \leftarrow\!\!\$\ \mathsf{Generate}(1^k)$ for this operation.
- The randomized *locking* algorithm Lock takes as input public and secret parameters pp and sp, a string S, and an integer complexity parameter $t > 0$. It returns a ciphertext C, a *locked container* X, and a *witness* T. We write $(C, X, T) \leftarrow\!\!\$\ \mathsf{Lock}(pp, sp, S, t)$ for this operation.
- The deterministic *unlocking* algorithm Unlock takes as input pp, t, C, X and returns a string S or the distinguished (non-string) symbol $\perp$. We write $S \leftarrow \mathsf{Unlock}(pp, t, C, X)$ for this operation.
- The deterministic *property attestation* algorithm Attest takes as input pp, t, X, T, C and returns a bit b. We write $b \leftarrow \mathsf{Attest}(pp, t, X, T, C)$ for this operation.

For correctness, we require that for all k, S, and t,

$$\Pr[(pp, sp) \leftarrow\!\!\$\ \mathsf{Generate}(1^k); (C, X, T) \leftarrow\!\!\$\ \mathsf{Lock}(pp, sp, S, t) : S \leftarrow \mathsf{Unlock}(pp, t, C, X)] = 1$$

where the probability is over the indicated random choices.

5.1 Discussion of the Syntax

Intuitively, the Lock algorithm is meant to provide a ciphertext C for the plaintext string S that, like traditional encryption, is efficiently decrypted by anyone holding the secret parameters sp. Unlike traditional encryption, Unlock *does not* take the secret parameters sp as input. Therefore, we define the locking algorithm to return the locked container X, which locks (escrows) the secret parameters. Our correctness condition demands that knowledge of honestly generated C, X (and knowledge of the corresponding t and public parameters pp) suffices to recover the plaintext S. Our working *assumption* is that t determines[3] the number of computational steps to unlock the secret parameters contained within X, and that one cannot recover the plaintext directly from C faster than that.

Also unlike traditional encryption, Lock is required to return a witness T, which is used by Attest (when access to data is requested by an appropriate authority) to provide quick verification, the exact nature of which depends on the specific levels and forms necessary for a particular domain (e.g., one can reasonably expect that the storage of forced-retention data will be handled differently, and have different requirements, for a one lawyer lawfi rm and a large multinational organization). Our Attest abstraction is intended to capture mechanisms that provide honest parties who are going to run Unlock with evidence that supports the assumption that the work will yield the correct data. The correctness condition already provides that Unlock should reveal what was hidden via Lock. Checking this, however, takes a long time. Attest quickly allows such parties to feel that the commitment was built faithfully.

As the intention of Attest is to provide, without the need for completing the lengthy unlock of the container, whatever verification is deemed appropriate, it cannot be precisely defined until those requirements have been determined. In our base scheme, a

[3] In our realization of a Dragchute system, the number of computational steps required to unlock the secret is 2^t.

successful output from Attest (i.e., return value 1) provides verification, via a non-interactive zero-knowledge proof (a component of witness T) that the container X has been correctly constructed, (equivalently, that the correctness condition on Lock, Unlock holds), that X corresponds to ciphertext C, and that neither X, T, nor C have been partially altered in place. In our scheme extended to protect against the injection (by an unauthorized party) of data, the successful return of Attest assures in addition that X, C, and T are genuine. Finally, in our scheme extended via a zkSNARK, Attest additionally quickly verifies that the preimage of a committed hash value is the (traditional) key necessary to decrypt C. This latter does not prove that the key itself is contained in X. It does, however, eliminate the potential for a committer to argue, upon an execution of Unlock that fails to produce the necessary key(s), that the wrong keys were accidentally placed in the container. In short, in the full scheme, if $\mathsf{Attest}(pp, t, X, T, C)$ returns 1, all parties should be convinced that X has been faithfully and correctly constructed: it is "well formed" (Unlock will succeed in revealing what has been committed), it corresponds to a non-fraudulent ciphertext C, it contains the key that decrypts C, and any attempt to change what has been committed, prior to completion of Unlock, will be detected.

6 REALIZING DRAGCHUTE

Now that we have introduced our core primitive, we explain how to instantiate it. We make use of standard cryptographic primitives (e.g., AEAD schemes, signature schemes, hash functions), as well as modified versions of timed-commitment scheme and NIZKP.

6.1 Dragchute Timed-Commitment Scheme

As timed-commitment schemes are somewhat outside of the mainstream of cryptography, we give a brief review of them before moving on to our system realization. A *timed-commitment scheme* is a four-tuple of algorithms CS = (Csgen, Commit, Open, Fopen). The randomized *commitment-key generator*, Csgen takes as input a security parameter and returns a commitment key ck; we write $ck \leftarrow\!\$\ \mathsf{Csgen}(1^k)$. The randomized *commitment algorithm* Commit takes the key ck, a vector of strings $\mathcal{S} = (S_1, S_2, \ldots, S_m)$, and a vector of integer time parameters $\mathcal{T} = (t_1, t_2, \ldots, t_m)$. It returns a commitment-decommitment pair, and we write $(\gamma, \delta) \leftarrow\!\$\ \mathsf{Commit}_{ck}(\mathcal{S}, \mathcal{T})$ for this operation. The deterministic *opening algorithm* Open takes as input $\mathcal{T}$, an integer $i \in [|\mathcal{T}|]$, γ, and δ, and outputs the string S_i in return. (Our scheme does not actually make use of this algorithm, as described below.) The deterministic *forced-opening algorithm* Fopen takes as input $\mathcal{T}$, an integer $i \in [|\mathcal{T}|]$, and γ, and outputs S_i in return. We write $S_i \leftarrow \mathsf{Fopen}(\mathcal{T}, i, \gamma)$ for this.

For correctness we require that for all security parameters k, all ck, all $\mathcal{S}, \mathcal{T}$ such that $|\mathcal{S}| = |\mathcal{T}|$, and all $i \in [|\mathcal{T}|]$, when $(\gamma, \delta) \leftarrow\!\$\ \mathsf{Commit}_{ck}(\mathcal{S}, \mathcal{T})$, we have

$$\mathsf{Fopen}(\mathcal{T}, i, \gamma) = \mathsf{Open}(\mathcal{T}, i, \gamma, \delta) = S_i$$

with probability 1 over the coins of Commit.

We have defined timed-commitments to allow for vector-valued inputs $\mathcal{S}, \mathcal{T}$ to support our ultimate constructions. However, when we talk about single-element $\mathcal{S}, \mathcal{T}$ we will replace them with S, t to signal this, and to better align with previous work on timed-commitments. We will also dispense with the integer index i provided to Open and Fopen.

Loosely speaking, if the party committing to $\mathcal{S}$ refuses to decommit, the forced-opening algorithm provides a way to reveal each S_i within a number of computational steps that is a specified function of t_i. Normally the committing party keeps δ *secret* until it is requested, and knowledge of δ allows for $\mathsf{Open}(\mathcal{T}, i, \gamma, \delta)$ to be efficient. In our setting, *we cannot assume any secrets*; thus δ will be discarded, and all openings will be forced openings.

Our timed-commitment scheme is an adaptation of one proposed by Boneh and Naor (BN) [14], which makes use of sequential squaring, a process long believed to be robust against parallelized attacks. The method,first introduced by Rivest, Shamir, and Wagner (RSW) [61] to achieve time-locked encryption, works as follows. Let p, q be large primes, set $N = pq$, and sample $g \leftarrow\!\$\ \mathbb{Z}_N^* \setminus \{1\}$. Let t be an appropriately chosen time parameter, and let $u = g^{2^{2^t}} \bmod N$. Note that though t may be large, given p and q one can quickly compute u, byfirst computing $e = 2^{2^t} \bmod \phi(N)$, where ϕ denotes Euler's totient function, followed by $u = g^e \bmod N$. Moreover, if one discards p, q (and assuming that it is computationally infeasible to compute $\phi(N)$) it is believed that the most efficient method to recompute u is by computing the successive squares $g^2, g^4, g^8, \ldots, g^{2^{2^t}}$ (all quantities modulo N). This process requires 2^t squarings, which may require hours, weeks, or months, depending on the value of t. The quantity u can be used for (say) encryption of a value X by returning the ciphertext $\langle (g, t, N), (X + u) \bmod N \rangle$.

In our case, as in the Boneh-Naor timed-commitment scheme, the value u serves as the anchor for a Blum-Blum-Shub (BBS) pseudorandom sequence [13]. Commitments require XORing with the tail end of such a sequence. Secrecy and integrity of the commitment are guaranteed by the properties of quadratic residues in appropriate groups — secrecy due to the difficulty of computing square roots, integrity due to specific uniqueness properties associated with square roots of quadratic residues.

The Boneh-Naor scheme consists of a commit mechanism together with a zero-knowledge proof that verifies the construction of the commit. Neither suffices in our setting because each requires that secrets be retained. Their zero-knowledge proof, for example, requires that the prover retain the order of the element g that serves as the base for the sequential squaring operation, which in turn generates the bits in the BBS blinding sequence. Unfortunately, $\mathrm{ord}(g)$ is a trapdoor, as for any natural number x, and any RSA modulus N, $g^x \bmod N = g^{(x \bmod \mathrm{ord}(g))} \bmod N$. Thus an adversary possessing $\mathrm{ord}(g)$ can unlock the committed string without completing the required sequential squarings. Additionally problematic in the current setting is that their associated zero-knowledge proof is interactive.

Our scheme, though based on Boneh-Naor's Commit operation, lets us discard *all* cryptographically sensitive material and eliminate interaction.

The Construction. Let K be a message to be committed (e.g., a cryptographic key or concatenation of multiple such cryptographically private parameters).

As stated earlier, a timed-commitment scheme consists of four algorithms: Csgen, Commit, Open, and Fopen. In our method, invocation of Csgen(1^k) results in the generation of the commit key ck as follows. Choose $N = pq$, the product of k-bit strong primes p and q, such that $p = 2\hat{p} + 1$ and $q = 2\hat{q} + 1$, where $\hat{p}$ and $\hat{q}$ are themselves prime. An immediate consequence is that both p and q are Blum primes (e.g., equivalent to 3 mod 4). Pseudorandomly choose $h \in \mathbb{Z}_N^* \setminus \{1\}$, and define $g \in \mathbb{Z}_N^*$ by $g = h^{2^k}$. This operation has the effect of guaranteeing that 2 is not a factor of ord(g) (a non-trivial, but relatively straightforward argument in group theory demonstrates this). A consequence is that ord(g) must divide $\hat{p}\hat{q}$, because ord(g) must divide $|\mathbb{Z}_N^*| = \phi(N) = 2^2\hat{p}\hat{q}$. Since ord($g$) can not be 1, this leaves only $\hat{p}$, $\hat{q}$, and $\hat{p}\hat{q}$ as possibilities, guaranteeing that ord(g) is large. Csgen returns the commit key $ck = (N, g)$.

On invocation of $\text{Commit}_{ck}(K, t)$, the values g and N are used to generate a BBS pseudorandom sequence. Specifically, let $u = g^{2^{2^t}}$, and define the sequence B by $B_i = \text{lsb}(g^{2^i} \bmod N)$, for $i = 0, 1, \ldots, 2^t$. That is, B is the sequence of parity bits generated by successively squaring g. Finally, assume that the message K has length j, with bits indexed from 1 to j, and define the string M of length j by $M_i = K_i \oplus B_{2^t - i}$. Note that the order in which the quadratic residues are used is opposite of the order in which they appear in the sequence of successive squares. That is, K_1 is XORed with the least significant bit of a square root of u. The next bit K_2 is XORed with a square root of the square root of u, and so on.

This process, starting at u and taking successive square roots, turns out to be well defined. Stated another way, it is clear that given the base g and parameter t, the sequence of successive squares is uniquely determined and will end at u. But the blinding sequence is uniquely determined as well by the value u, despite the fact that quadratic residues in $\mathbb{Z}_N^*$ can have multiple square roots. For the product of two Blum primes N, every quadratic residue in $\mathbb{Z}_N^*$ has four square roots, exactly one of which is itself a quadratic residue (see, e.g., Katz and Lindell [45]). Since by definition every element in our sequence is a quadratic residue, this means that starting at u and taking successive square roots (that are themselves quadratic residues) is a well-defined process that will create the same sequence that would result from starting at g and successively squaring. As a result, given the value u, the bits that make up the blinding string B are uniquely determined.

What makes this commit method attractive is that for the committer who knows p and q, the string M is easily created — one finds $B_{2^t - j}$ byfi rst computing $e = 2^{2^t - j} \bmod \phi(N)$, then computing $B_{2^t - j} = g^e \bmod N$. Finishing M from this point requires only j successive squares. Alternatively, one could start at u and compute square roots using any algorithm that exists for computing square roots modulo N when the factorization of N is known. On the other hand, lacking p and q, the only practical means of computing B is by completing the required successive squares starting from g. In particular, a result of Rabin [58] asserts that lacking the factorization of N,fi nding square roots of elements of $\mathbb{QR}_N$, the subgroup quadratic residues in $\mathbb{Z}_N^*$, is at least as difficult as factoring.

Commit ends by returning (γ, δ), where $\gamma = (N, g, t, M)$ and $\delta = (p, q)$. The method is easily extended in the natural way to the general situation ($\text{Commit}_{ck}(\mathcal{S}, \mathcal{T})$) involving vectors of messages and their corresponding time parameters, in which case γ has the form $\gamma = (N, g, \mathcal{T}, \mathcal{M})$ for vectors $\mathcal{T}$ of time parameters and $\mathcal{M}$ of blinded strings.

Though we do not use algorithm Open in our system (and in fact discard δ once the Lock algorithm returns), its construction is straightforward: given t, γ, and δ, one can easily and quickly extract message K. Algorithm Fopen is also straightforward, though not nearly as quick: given t and γ, the quickest path to unlocking K is by computing the successive squares necessary to learn the blinding string B.

6.2 Dragchute Base NIZKP

In addition to the Commit scheme above, our base Dragchute scheme includes a *non-interactive* zero-knowledge proof (NIZKP) that demonstrates to a verifier that the Commit process has been properly executed. Specifically, the proof verifies that u has been constructed according to the required method, effectively binding the committer to whatever is revealed when the commit is opened, and providing the verifier with assurance that completing the required compute task will unlock the commitment. Our proof modifies the Boneh-Naor ZKP (itself a variant of a classic ZKP of Chaum and Pedersen [21]) such that no cryptographically sensitive material need be retained and interaction is eliminated. For the latter, we employ the Fiat and Shamir [32] method for removing interaction from a ZKP, and point the reader to that work for proof that the resulting NIZKP retains the zero-knowledge properties of the original. The remainder of this section describes the resulting NIZKP.

Zero-knowledge proofs involve a *prover* and a *verifier*, executing algorithms Prv and Vfy respectively. The prover in the present context is the party that is encrypting and committing data via the method described above (that is, has computed $u = g^{2^{2^t}}$ quickly using their knowledge of p and q), while the verifier is the party that wishes to access the committed data. The purpose of this proof is to assure the verifier that the value of u has been constructed properly (as that uniquely determines the BBS blinding sequence). Our NIZKP is produced during the Dragchute Lock operation, so algorithms Prv and Vfy potentially have access to cryptographically sensitive information (container *sp*) that is later discarded. The proof, however, is simulating the associated interactive proof, in which there is an asymmetry between the prover and the verifier. We thus require that algorithm Vfy not have access to any private cryptographic information that is to be subsequently discarded. In particular, the inputs to Prv are γ, δ (i.e., p and q), a (public) security parameter R, and ord(g). Inputs to Vfy are γ and R. In addition, both algorithms have access to a random oracle σ, which for purposes of simplicity of exposition, accepts binary strings as inputs, and always outputs values of the appropriate size or range for their stated purpose.

As in the previous sections, all arithmetic in the following is understood to be done modulo N, unless specified. The proof constructed by the prover begins as an empty string τ. Construction of the proof begins with algorithm Prv computing the vector

$$W = \left\langle g^2, g^4, g^{16}, g^{256}, \ldots, g^{2^{2^i}}, \ldots, g^{2^{2^{(t-1)}}}, g^{2^{2^t}} \right\rangle,$$

which is then appended to τ. Since Prv knows p and q, the $t + 1$ component values of W can be computed quickly. For brevity, let us

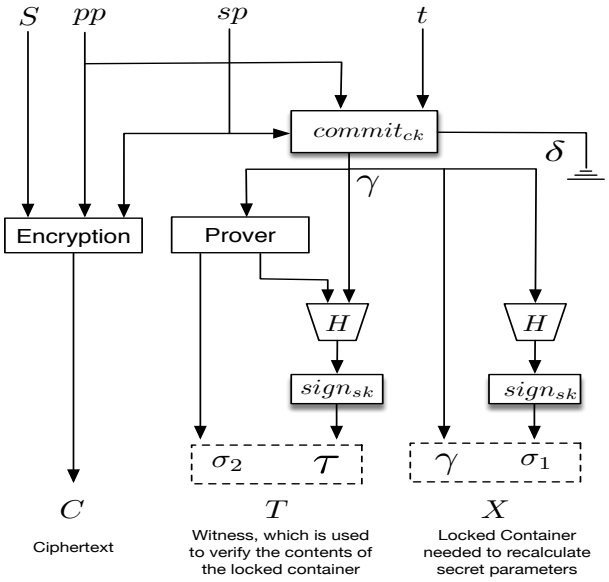

Figure 3: In addition to the encryption of a string, the Lock **function also provides a time locked container, which requires sequential computational work in order to extract the secret parameters, and a witness that verifies the contents of the locked container.**

write $W = \langle b_0, b_1, \ldots, b_t \rangle$, where each $b_i = g^{2^{2^i}}$. For $i = 1, 2, \ldots, t$, the prover intends to prove to the verifier that each triple[4] of the form (g, b_{i-1}, b_i) has the form (g, g^x, g^{x^2}) for some x.

The proof,[5] which simultaneously proves the relation for all t of the (g, b_{i-1}, b_i) tuples requires multiple rounds each consisting of the following four steps.

Step 1: Prv queries σ in order to generate t values, $\alpha_1, \alpha_2, \ldots, \alpha_t \in \mathbb{Z}_{\text{ord}(g)}$. Prv then computes $z_i = g^{\alpha_i}$ and $w_i = b_{i-1}^{\alpha_i}$ for $i = 1, 2, \ldots t$, and appends all pairs $\langle z_i, w_i \rangle_{i=1}^{k}$, in order, to τ.

Step 2: Prv uses σ to generate t integer values $c_1, c_2, \ldots, c_t$, each in $[0, R]$, as follows. First, vector W, along with all of the z_i and w_i is input to σ. Thefirst $t \log(R)$ bits of $\sigma(W, z_1, w_1, z_2, w_2, \ldots, z_t, w_t)$, broken into t-bit chunks, then specify, in order, the c_i. Prv appends the c_i values to the proof τ. Prv then computes $y_i = c_i \cdot 2^{2^{i-1}} + \alpha_i \pmod{\text{ord}(g)}$, for all $i = 1, \ldots, t$, and appends these as well to τ.

Step 3: Prv stores τ, for later transmission to Vfy, if necessary.

Step 4: Vfy, upon receipt of τ, uses W, the z_i, and the w_i to generate (and verify the values of) c_i and check that for all $i = 1, \ldots t$,

$$g^{y_i} b_{i-1}^{-c_i} = z_i \qquad \text{and} \qquad b_{i-1}^{y_i} b_i^{-c_i} = w_i \tag{1}$$

Vfy returns 1 (accept) if each of these relations holds, and 0 (reject) if any of the relations fails.

This process is repeated for a number of iterations dependent on the desired level of security. Boneh and Naor prove that their zero-knowledge protocol is such that in a single round, Prv can fool Vfy into accepting an incorrectly formed W with probability at most

$$t \cdot \left[\frac{1}{\min(\hat{p}, \hat{q}, R)} + o\left(\frac{1}{R}\right) \right].$$

Thus larger R values result in greater assurance per round, at the cost of potentially higher round computation costs due to larger c_i values. We also note that security parameter R strictly deals with the *probability* of accepting a fraudulent proof and not the privacy of the proof itself.

6.3 Construction of Our Base Dragchute Escrow Scheme

Given the core timed-commitment primitive and NIZKP described above, we now provide a description of our base Dragchute scheme. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a traditional authenticated encryption scheme with IV-space $\mathcal{V}$, and let DS = (Dsgen, Sign, Versig) be a digital signature scheme. Let CS = (Csgen, Commit, Open, Fopen) be our timed-commitment scheme. Let $H\colon \{0,1\}^* \to \{0,1\}^\ell$ be a hash function for somefi xed $\ell > 0$ and $R > 0$ be a security parameter. Finally, let $\mathcal{K}'$ be an auxiliary (salt) key space consisting of keys of length ℓ'.

Given these cryptographic primitives we build the operations Generate, Lock, Unlock, and Attest as follows.

Generate. We define Generate(1^k) to run $(pk, sk) \leftarrow\!\!\$$ Dsgen(1^k)[6], $ck \leftarrow\!\!\$$ Csgen(1^k), and $K \leftarrow\!\!\$ \mathcal{K}$; sample an IV $V \leftarrow\!\!\$ \mathcal{V}$, run $K' \leftarrow\!\!\$ \mathcal{K}'$, and return $pp \leftarrow (pk, ck, V)$, $sp \leftarrow (K, K', sk)$.

Lock. The locking algorithm Lock(pp, sp, S, t) first computes $C \leftarrow \mathcal{E}_K^V(S)$, the encryption of S under key K and initialization vector V. Next, it runs $(\gamma, \delta) \leftarrow\!\!\$$ Commit$_{ck}((K, K'||H(K'||C))$, $(t, \lceil \log(\ell' + \ell) \rceil))$ (here $\mathcal{S} = (K, K'||H(K'||C))$ and $\mathcal{T} = (t, \lceil \log(\ell' + \ell) \rceil = \lceil \log(|K'||H(K'||C)|) \rceil$[7]). It then computes signature $\sigma_1 \leftarrow \text{Sign}_{sk}(H(\gamma))$ and sets $X \leftarrow (\gamma, \sigma_1)$. Finally, it builds the NIZKP τ for the commitment γ, and generates a signature $\sigma_2 \leftarrow \text{Sign}_{sk}(H(\langle \tau, \gamma \rangle))$, where $\langle \cdot \rangle$ is some unambiguous encoding of its argument as a bitstring. The algorithm assigns $T \leftarrow (\tau, \sigma_2)$, and returns C, X, T. We show a diagram of this algorithm in Figure 3.

Unlock. Unlock(pp, t, C, X) first parses X into γ and σ_1, and runs the signature verification algorithm Versig$_{pk}(H(\gamma), \sigma_1)$; if this fails then Unlock returns $\perp$. Otherwise, it runs $sp \leftarrow$ Fopen$(\mathcal{T}, 1, \gamma)$, recovering K. It then decrypts $S \leftarrow \mathcal{D}_K^V(C)$. If decryption fails, Unlock returns $\perp$; else it returns S. We show a diagram of this algorithm in Figure 4.

[4] In the typical language used in much of the zero-knowledge proof literature, one is proving that (g, g^x, g^x, g^{x^2}) is a discrete Diffie-Hellman tuple (DDH-tuple) - by definition a tuple of the form (g, h, A, B), where $A = g^x$ and $B = h^x$ for some unknown exponent x.

[5] We note that publicizing W has a minor effect on choosing the t value for a desired lock time. We will explain such effect in Section 7.2.

[6] We note that the public-private key pair generated here must be a session key pair, rather than a long term key pair, as the private key sk is discarded during the Dragchute process.

[7] Recall that time parameter t corresponds to 2^t sequential squarings. Thus the value of t required to generate at least $\ell' + \ell$ blinding bits is the least t such that $2^t \geq \ell' + \ell$, from which the log expression follows.

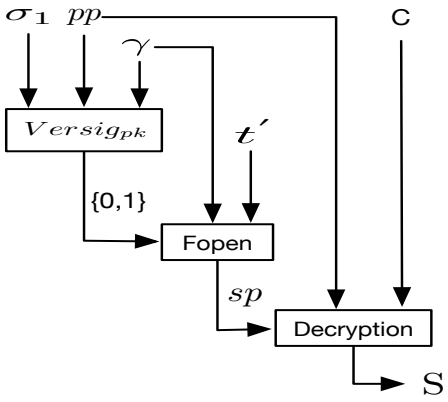

Figure 4: The Unlock **scheme is given a locked container, ciphertext, public parameters, and a time value. After verifying the signature and spending and extensive computation time,** Unlock **return the decrypted string.**

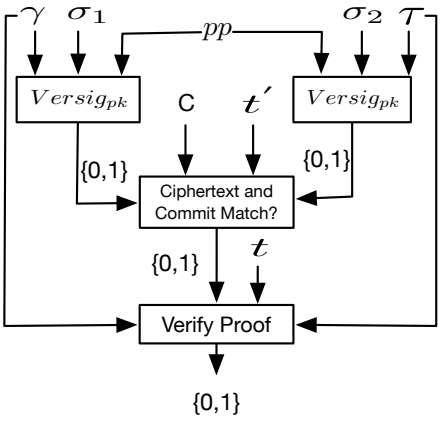

Figure 5: Attest **is able to verify that NIZKP is well formed, and that the commitment γ matches the ciphertext. This information ensures the verifier that if they complete the required computation, they will be able to retrieve the locked secret parameters. In addition, the verified binding with the ciphertext allows γ to serve as evidence of fraud should the commit fail to contain the key(s) that decrypt ciphertext C.**

Attest. In our base scheme, Attest(pp, t, X, T, C) first parses X into γ and σ_1, and T into τ and σ_2. If either verification Versig$_{pk}(H(\gamma), \sigma_1)$ or Versig$_{pk}(H(\tau), \sigma_2)$ fails, then Attest returns 0. Next, Attest runs $U||V \leftarrow$ Fopen$(\mathcal{T}, 2, \gamma)$, where $|U| = \ell'$ and $|V| = \ell$, and tests whether $V = H(U||C)$. If the test fails, then Attest returns 0. Otherwise, it runs the NIZKP verifier Vfy(τ, γ). If this fails then Attest returns 0; otherwise it returns 1. We show a diagram of this algorithm in Figure 5.

We note that, although not explicitly stated as part of the construction, it is intended that the secret parameters sp and the decommitment δ are securely destroyed after the locking algorithm is run. This is implicit in the fact that neither Unlock nor Attest use these quantities.

6.4 Extensions to the Base Scheme

Up to this point, we have presented the base Dragchute system. While it provides sufficient protection in some contexts, the base scheme does not protect against an *active* adversary who actively replaces data and then protects that data using Dragchute (e.g., replacing tax returns with fraudulent ones). Nor does it provide any guarantee that the escrowed material (locked container) contains the key that decrypts the corresponding ciphertext. We discuss here extensions that provide provenance verification in thefirst scenario and succinct non-interactive verification in the second. The extensions presented here are orthogonal in that they can be used either by themselves or in tandem or as context may require.

Provenance Verification. If an adversary is able to create fraudulent data and protect it via Dragchute, the base version of Attest will show that the (fraudulent) container has been correctly constructed and that it corresponds to the (fraudulent) ciphertext. It will thus prove to a third party that completing the required computational puzzle will ultimately unlock the desired data. The honest committer, however, may not be aware of such replacement until after the verifier has spent the time to perform Unlock. If at this point the verifier concludes that the unlocked data is fraudulent, the success of Attest points to obstruction by the unaware honest committer. Potentially worse is the possibility that the fraud is *not* detected, in which case the authority unknowingly acquires fraudulent data. The underlying issue is the result of not being able to verify the identity of *who* encrypted the data.

In order to prevent this, Dragchute can be extended to include provenance verification by making use of a trusted certificate authority (CA). Since the base Dragchute system already uses a public key pair pk, sk, a CA can simply sign a certificate $cert_{de}$ for those values. The honest committer can then tie his/her identity to any Dragchute instantiation by creating a separate certificate in each encryption. More concretely, for every Dragchute instantiation, Generate will now compute the certificate $cert_{de}$ on the public key pair generated. Lock, as before, creates ciphertext C, container X and witness T, but now additionally includes the signature $\sigma = sign_{sk}(H(X||C||T||cert_{de}))$. Output of Lock, when modified by this extension, becomes X, C, T, $cert_{de}$, σ. As with the base scheme, all secret values are discarded. Computing the signature with $cert_{de}$ as a parameter allows us to bind the certificate (and therefore the certificate chain) to all other outputs from the Lock function. As $cert_{de}$ can only be created with a signature from the CA and the CA checks the identity of the honest committer, verifying the signature ties the Dragchute instance to the honest committer.

To match the changes made to Lock, Attest is extended to receive two new inputs: $cert_{de}$ and σ. In addition to the checks required in the base scheme, the verifier now also attests to the identity the creator of the Dragchute instantiation. To do so, she mustfi rst validate the signature σ by recomputing the hash $H(X||C||T||cert_{de})$ and using the public key pk found in $cert_{de}$. If validation succeeds, she then validates the certificate chain of $cert_{de}$. If the certificate chain validates successfully, the verifier should be convinced of the identity of the honest committer. As before, if any step in the attestation process fails, Attest returns $\perp$.

Succinct Non-interactive Verification. While our base system is able to link the ciphertext C, escrow container X, and witness T, it does not prove to the verifier that X hides the key that will decrypt C. Having such proof would immediately lock an honest committer to the output of Unlock, essentially preventing the committer from providing false information to the verifier, whether accidental or intentional. Unfortunately, in the absence of any trap-door information (i.e., the factors p and q of N), the decision problem "does the locked container contain the correct decryption key?" is not likely NP, since Unlock requires exponential time (under the assumption that the quickest path to open the container is by completing the sequential puzzle). The decision problem "Is the preimage of this hash the key that decrypts this ciphertext?", however, is NP, and moreover, zk-SNARKs have been implemented for proving just this statement (see, e.g., [15, 16]). Campanelli et al., [16] specifically shows that trustless (i.e., no trust needed during setup) zero-knowledge proof is possible that can verify that a published value Y and ciphertext C are such that the preimage of Y is the key K (i.e., $SHA256(K) = Y$) that decrypts C. However, their constraints are not the same as ours. Specifically, since the honest committer will not know who the verifier is ahead of time, and the verifier does not necessarily trust the committer, the required proof in the current context must be both trustless and non-interactive. To that end, recent work by Ames et. al. [6] shows how zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK) that are both trustless and non-interactive can be created. We point the reader to their paper for the specifics on the construction of such proofs.

To extend the Dragchute system for this type of verification, Lock needs to compute a witness T_{pre} on the public value Y using the Ames construction. The hash Y is then appended to the string vector $\mathcal{S}$ (defined in Section 6.1) along with a small time value t (appended to $\mathcal{T}$) that determines where in the BBS sequence Y is encoded. Doing so effectively commits the value Y, but does so with a quick unlock time. Lock then outputs T_{pre} in addition to all previous variables. Extending Attest to verify the zk-SNARK requires only the additional input of the new witness T_{pre}. When run, Attest can then quickly verify the zk-SNARK by extracting Y from the BBS string (using t) and then validating that T_{pre} is a witness to the decode valued.

7 PROTOTYPE IMPLEMENTATION AND PERFORMANCE

We now provide implementation details of the base Dragchute system, and evaluate its performance. We focus exclusively on the base scheme rather than providingfi gures for four different configurations (base alone, base with provenance extension, etc) because the costs of the extensions are negligible compared to the costs of the base Dragchute operations.

7.1 System Implementation

We now explain how our prototype implements the Generate, Lock, Unlock, and Attest functions.

Generate**:** This primitive creates the public and private parameters, *pp* and *sp*, used by the other primitives. Generate first generates 2048 bit RSA modulus $N = pq$, the product of strong primes p and q, the primes generated[8] using a technique of Williams and Schmid [76]. Element $g \in \mathbb{Z}_N^* \setminus \{1\}$ is then generated following the method described in Section 6.1. Together, these form commit key $ck = (N, g)$. Next, one 128-bit symmetric key (K_1 – used for encryption), two 256-bit symmetric keys (K_2, K_3 – used for HMAC), a 1024-bit salt (K'), and an RSA 2048 public key pair (K_{pu}, K_{pr} – used for a digital signature) are generated, and all keys but K_{pu} are placed inside container *sp*. Finally, the algorithm generates initialization vector V, and places V, ck, and K_{pu} in container *pp*, and outputs *pp* and *sp*.

Lock**:** Lock takes as input *pp* and *sp*,file S (to be encrypted), and time parameter t, and outputs ciphertext C, a commitment X, and a witness T (containing the NIZKP). To obtain C,first S is encrypted using AES-CTR-128 with key K_1 and initialization vector V to obtain intermediate ciphertext $\tilde{C}$. Lock then computes tag $\tilde{T}$ = HMAC-SHA-256($K_2, \langle$"TAG"$\rangle||\tilde{C}$), where $\langle\omega\rangle$ means an unambiguous binary representation of the enclosed object ω, and truncates the result to 128 bits. The algorithm then creates ciphertext $C = \tilde{C} \,||\, \tilde{T}$. Thus, we instantiate an "encrypt-then-MAC" style authenticated encryption scheme [10, 52].

In order to generate X, containers γ and δ must be computed. Toward this end, Lock uses the values N, p, q, g and t to calculate ord(g) and to quickly calculate u, following the method discussed in Section 6.1. Then ord(g) is used to quickly compute the BBS commit sequence B. Lock then hashes $K'||C$ via SHA-256, and sets M' to $K_3||K'||$SHA-256($K'||C$). The string M is then formed by XORing $K = K_1||K_2$ with the tail end of B, and M' with the front end of B (in the language of Section 6.1, $S_1 = K$, $S_2 = M'$, $t_1 = t$, and $t_2 = 11$, since we require 2^{11} bits to represent string M', which is $256 + 1024 + 256 = 1536$ bits long). Containers γ and δ are then set to (N, g, u, M) and (p, q) respectively. Next, digital signature σ_1 is generated byfi rst hashing γ via SHA-256, and then signing the hash using K_{pr}. Finally, X is set to $X = \gamma \,||\, \sigma_1$.

Witness T is derived as follows. First, security parameter R is set to 2^{64}. Then Lock generates the vector W (as described in Section 6.2) and appends this to proof string τ.

Next, for $i = 1, \ldots, t$, α_i is set to

$$\alpha_i = \text{HMAC-SHA-256}(K_3, \langle\text{ALPHA}\rangle||\langle i\rangle) \bmod (\text{ord}(g)).$$

Lock then computes the z_i and w_i values as discussed in Section 6.2. The (z_i, w_i) pairs are then appended to τ.

The c_i are then generated as follows. First, Lock computes the string $\Pi = \langle W\rangle||\langle z_1\rangle||\langle w_1\rangle||\langle z_2\rangle||\langle w_2\rangle||\ldots||\langle z_t\rangle||\langle w_t\rangle$. Then Lock generates the $(256\lceil t/4\rceil)$-bit string resulting from the concatenation of HMAC-SHA-256($K_3, \langle 1\rangle||\Pi$) through HMAC-SHA-256($K_3$, $\langle\lceil t/4\rceil\rangle||\Pi$), from which the c_i are created (in order) by selecting consecutive 64 bit chunks. Lock then uses the c_i to compute the y_i

[8] Generate performance can be improved by precomputing primes. This is reasonable, so long as any prime used in the Lock operation is immediately removed from the collection and securely deleted. Though keeping such primes might seem to violate the requirement that no cryptographically sensitive information be retained, such a list provides no advantage to attackers attempting to obtain currently existingfiles that might subsequently be protected by Dragchute, ostensibly because they already have access to thesefi les. We note, however, that should a breach be detected, all currently precomputed primes must be discarded, in order to prevent the adversary from potentially gaining access tofi les created and protected by the system in the future.

values. Both c_i and y_i are appended to τ. Finally, digital signature σ_2 is generated by signing the SHA-256 hash of τ under the key K_{pr}. Thefi nal witness T is set to $T = \tau \,||\, \sigma_2$.

At this point the secret keys K_1, K_2, and K_{pr}, along with δ are securely deleted, and values C, X, and T are output.

Earlier, in Figure 3, we showed that sp and pp are used independently to create the ciphertext C and both the locked container X and witness T. Since thefi le encryption and the creation of the locked container/witness are independent processes, they can be executed in parallel. Thus Lock can perform the calculations needed for key escrow at the same time as the encryption of thefi le. As a result, asfi le sizes grow, the time required to apply Lock to afi le is dominated by the time for (traditional) encryption, instead of the time required to apply Dragchute-specific primitives. In addition to this internal parallelization, multiple instantiations of Lock can run simultaneously in order to encryptfi les in parallel [9]. We further discuss performance as a function offi le size in Section 7.2.

Unlock**:** Unlock accepts pp, t, C, and X as inputs. Next pp is unpacked, revealing K_{pu} and V, and X is parsed to obtain γ and σ_1. Unlock then checks signature σ_1 using K_{pu}. If the signature passes, γ is parsed to obtain secret M, along with all parameters required to calculate the BBS sequence B. As $ord(g)$ is no longer available, Unlock *must* calculate sequential squares to learn the blinding sequence B and decode the secret K; a time consuming process. Once completed, the decoding process returns K_1, and K_2. K_2 is then used to check that HMAC-SHA-256$(K_2, \langle \text{TAG} \rangle || \tilde{C})$, truncated to 128 bits, matches $\tilde{T}$. If it succeeds, the algorithm decrypts $\tilde{C}$ using K_1 and V, and outputs plaintextfile S. If the check fails, Unlock returns $\perp$.

Attest**:** Attest accepts inputs pp, t, X, T, and C. The algorithmfirst unpacks K_{pu} from pp, then parses X and T to obtain γ and τ, along with signatures σ_1 and σ_2. K_{pu} is then used to check whether the signed hashes of γ and τ match the corresponding signatures. If both checks pass, Attest unpacks γ and quickly unlocks K_3, K', and SHA-256$(K'||C)$ from the string M contained in γ. Attest then uses K' and C to compute SHA-256$(K'||C)$ and verify that it matches the value that was unlocked from M. If the test passes, Attest unpacks the values contained in τ, and uses W, z_i, w_i, and K_3 to recompute c_i, and checks that these values match the c_i values contained in τ. Finally, Attest checks that the b_i (components of W), c_i, z_i, w_i, y_i, and g, satisfy equation (1). If this check passes, Attest returns 1. Otherwise Attest returns 0.

7.2 Performance Evaluation

We tested our prototype on a 40 core Intel(R) Xeon(R) CPU E5-2630 v4 2.20GHz server with 128 GB of memory. Implementation code was built using Java openjdk 1.8, which provides JIT compilation to all programs by default. We used Bouncy Castle as our standard cryptographic library.

All required strong prime numbers were precomputed prior to runtime using Java's Secure Random generator. Though there is considerablefl exibility when choosing the size of the primes, we chose 1024-bit primes as these provide the best performance while still maintaining a sufficient level of security. Further discussion of our choice of prime size is contained in Appendix A.

[9]Each instantiation would need its own public and secret parameters.

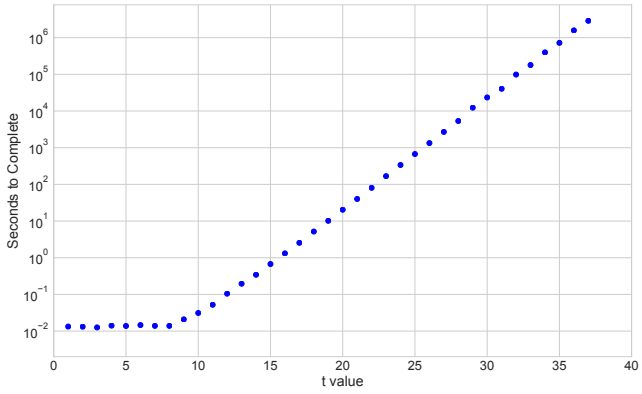

Figure 6: Files locked with Dragchute require exponential time to unlock the content.

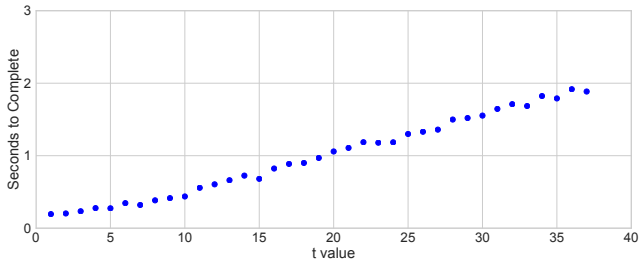

Figure 7: The Lock **function is able to encrypt afi le and assure confidentiality for a user defined time by providing a locked container. It also provides a NIZKP witness that can be used to verify if the container is formed correctly.**

We measured 10 runs per tested t value, averaging the outputs corresponding to each. Because there is no shortcut for computing Unlock, the tests required significant computing resources and time. Specifically, in order to ensure that results were representative, *a total of 686 days of computational time was required for the analysis.*

We now consider the performance of each of the four Dragchute operations. The graphs for the three primitives, Lock, Attest, and Unlock that depend on parameter t do not display error bars because the standard deviations of all measured quantities were between one and two magnitudes smaller than the mean.

Generate. This function is responsible for the generation of strong primes, symmetric keys and public parameters. The average total time for this set of operations was 35.95 seconds, with standard deviation $\sigma = 1$ sec. On average, 97.2% of this time was spent finding strong primes. Fortunately, the search for strong primes is parallelizable and readily pre-computable, so that an administrator can create many such values prior to use. Since this function does not depend on t, its performance is relatively constant.

We note that traditionally, cryptosystems run the Generate function once, generating a set of "master keys". With our system, however, Generate is run on a per-use basis. There are two reasons for this. First, as implemented, running Generate only once can result in unsafe initialization vector reuse. Worse, using the same g

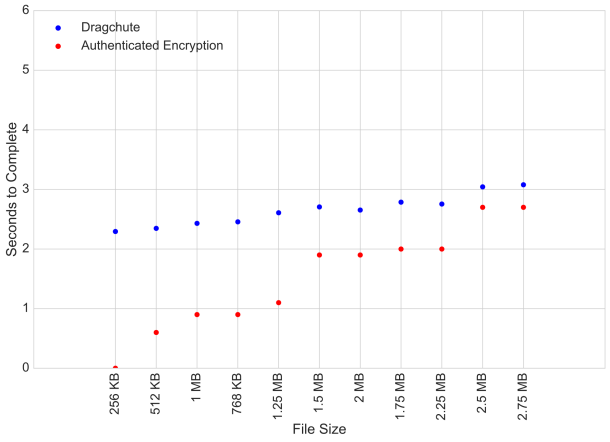

Figure 8: The overhead percentage increase of Dragchute vs authenticated encryption decreases asfi le size increases, becoming negligible (less than 6%) forfi les larger than 2.5 MB.

multiple times can result in intersection attacks, as the same pseudorandom BBS sequence could be generated for multiple commits.

Unlock. For this and the remaining functions, we characterize performance via regression analysis and present our results in an appropriate regression model (i.e., $y = mx + b$ or $y = AB^x$, where $x = t$) with corresponding coefficient of determination (r^2) values.

As shown in Figure 6, for practical t values, Unlock is exponential in nature, so we use an AB-exponential (i.e., $y = AB^x$) regression test to model its performance over this practical range. Specifically, for very small values of t, the costs of running the underlying algorithms (e.g., memory allocation operations) dominate, making these small values poor indicators of what can be expected in a real deployment. For values of t greater than 10 (t values less than 10 require under a second to unlock), the performance of Unlock is well modeled by the curve $y = 2.316 \cdot 10^{-5} \cdot 1.992^x$, with $r^2 = 0.9998$. A practical result from our experiments shows that if a Dragchute system requires one month of Unlock time, the corresponding t value to be used would be 37. Additionally, given the extremely goodfi t from our model, we can accurately and confidently predict performance for larger t values, without the necessity of performing additional experiments that would require months.

Lock. Figure 7 shows the performance of Lock as a function of t. The primitive requires between 0.19 ($\sigma = 0.03$) and 1.88 ($\sigma = 0.14$) seconds for values of t from 1 to 37, and is best modeled by the curve $y = 0.051x + 0.018$, with $r^2 = 0.996$. Accordingly, we can calculate that the costs for even larger values of t will be relatively low. We note that the time required for Lock to complete is dominated by the time needed to calculate the NIZKP. Like the Unlock function, values less than $t = 10$ are poor indicators of performance. As such, generating the NIZKP takes 74%, at $t = 10$, to 92%, at $t = 37$, of the total Lock time. This is expected as each increment of t adds a new round of calculations needed for the NIZKP.

As mentioned in Section 7.1,fi le encryption and the generation of a locked container/witness can be executed simultaneously. In Figure 8, we measure the performance of Lock with t fixed at $t = 37$

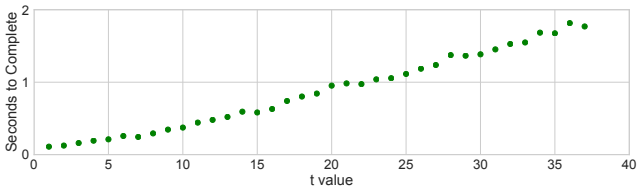

Figure 9: Verifying the NIZKP provided by Dragchute can be done quickly. This verification will prove that the ciphertext matches with the commitment container and that if someone is willing to put the computational time, they will reach the secret parameters at the end.

(approximately one month of Unlock time) and varyingfi le sizes. The graph shows that Lock overhead is negligible (less than a 6% increase) over regular AEAD. While the overhead for smallerfiles is relatively large,fi le sizes greater than 2.5 MB are only effected by the extra hashes on the ciphertext needed for attesting purposes. It is because of these hashes that Dragchute requires slightly more time than authenticated encryption.

Given the experimental results for both the Lock and Unlock functions, we can see that for one month of unlocking time, locking afi le would take approximately 2 seconds. Additionally, since both models have a high coefficient of determination, larger values for locking and unlocking may be extrapolated using these models. Because the data being protected by Dragchute is very unlikely to be required during the period during which it is locked, Lock cost represents a reasonable overhead, especially for a process that might be running in the background.

Attest. Figure 9 shows the performance of the Attest function, which is modeled well by the curve $y = 0.049x - 0.06$, with $r^2 = 0.995$ for t values between 1 and 37. As with Lock, Attest grows slowly with increasing t. This is to be expected, as incrementing t has the effect of requiring just a handful of additional calculations to the primitive.

Implementation Consideration. As mentioned in Section 6.2, publicizing vector W has an effect on the choice of t value appropriate for a desired unlock time. Any party running Unlock will be given the exact half way point of computation required by the BBS sequence by looking at the next to last component of vector W. Because Unlock time doubles for every increment of t, an implementation of a Dragchute system needs to use $t + 1$ to reach the desired unlock time.
We discuss other application specific considerations in Appendix B.

8 DATA RETENTION LAWS

In the United States, data retention laws affect data collected in the processing of corporate audits, telecommunications, credit and mortgage applications, credit records, employment records and applications, and banking, among others. Perhaps the most influential of these laws is the Sarbanes-Oxley act [2]. Drafted in the wake of the Enron and Worldcom scandals, Sarbanes-Oxley Section 802 requires that audit data pertaining to publicly traded companies must be retained for a period offi ve years. The act specifically charged

the Securities and Exchange Commission (SEC) with promulgating the details as to what specific data must be retained, and for how long. The SEC met this charge by adopting Rule 2-06 of Regulation S-X [65]. Rule 2-06 mandates that among those records that must be retained are "...records relevant to the audits or reviews of...fi nancial statements, including workpapers and other documents that form the basis of the audit or review, and memoranda, correspondence, communications, other documents, and records (including electronic records), which are created, sent or received in connection with the audit or review, and contain conclusions, opinions, analyses, orfi nancial data related to the audit or review." The SEC extended the required retention time for this data to seven years (two extra years) after the auditor concludes the audit.

Despite the seven year rule stated in the statute, several sources suggest that data should be stored much longer. In particular, The Federal Taxation Committee of the Massachusetts Society of Certified Public Accountants [74] has published recommended guidelines for retention of accounting records corresponding to several classes of data, including records from audits of corporate entities, legal proceedings related records, human resource and payroll related records, and records associated with individuals. Under these guidelines, several categories of records should be retained for 10 years, and more than 80 classes of records are recommended to be retained in perpetuity. The latter include, among records pertaining to individuals, tax returns, medical records, W2 forms, canceled checks for purchase of major improvements and maintenance of houses, wills, trust agreements, detailed lists offi nancial assets, alimony, custody or prenuptial agreements, military papers, and photos or videotapes of valuables. Among records pertaining to corporations, classes recommended to be retained permanently include all general correspondence, canceled dividend checks, cash disbursement and receipt records.

In addition to data retention required by Sarbanes-Oxley, laws such as the Home Mortgage Disclosure Act (HMDA) [4], the Equal Credit Opportunity Act (ECOA) [3], HIPAA [1], and various individual state laws (e.g., California's Peace Officer Selection Requirement Regulations [37]) also order the mandatory retention of data. The HMDA requires that data records for home purchases and improvements be retained for at least three years, some data for at leastfi ve years. The ECOA requires that all written or recorded information corresponding to a credit application be retained for 25 months after the data at which the applicant is informed of the action taken on the application. Furthermore, HIPAA requires the retention of policies and procedure documentation for six years after creation or their last effective date. Topping all of these, regarding length of retention, the California Peace Officer Selection Requirement Regulations require that all information collected during the application of background checks (which includes credit, employment, criminal, and educational records, among others) be retained for a minimum of two years for every applicant, and for as long as the applicant remains employed if hired by the department.

8.1 Legal Aspects of Data Access Delay

That Dragchute protected data is temporarily inaccessible, even for the parties that own it, is not legally problematic. In the United States, procedures for producing (i.e., "turning over") data in response to a subpoena are specified by Rule 34 of the Federal Rules of Civil Procedure [31]. Rule 34 allows a period of 30 days to *respond* to a request for data. The response typically initiates a period of negotiation to determine when and in what form the data will be produced, objections, etc. Production of data often occurs several months after the initial request. Moreover, discovery rules do not require companies to keep their data in any particular form, and explicitly allow the production of data in the format in which it is normally stored by the owning organization. An organization cannot make their data more inaccessible *once litigation is likely*, but an organization that uses Dragchute in the normal course of business is within their right to produce data in that form [30].

Similarly, delayed access poses no difficulties with regard to criminal procedures for the type of data under consideration here. Certainly immediate data access is sometimes necessary for law enforcement purposes, such as accessing current phone records while tracking a fugitive. But protecting such data is not the question we address in this paper. Rather, our method is intended to protect historical information required to be stored for regulatory purposes (e.g., receipts containing account information). One can never be certain what data law enforcement will need or why they will need it, and they can and do ask for such historical data, but there is no reason to believe that they would ever need such data "immediately" [19, 27].

Regarding the time duration during which data is inaccessible, we envision forced-retention data being locked for at most one or two months.[10] Should our method be implemented in real-world systems, it is likely that the legal system in relevant jurisdictions will decide both what classes of data can be protected via the method, as well as the allowable time periods for which data can be locked.

9 CONCLUSION

Data breaches will continue to pose a serious threat to organizations large and small. While many techniques already exist to attempt to limit the damage caused by such events, these methods often fail in practice because the secrets (e.g., cryptographic keys, data shares, etc.) necessary to protectfi les often end up themselves being compromised as well. This situation is only made worse by the fact that manyfi les simply can not be deleted due to regulation, potentially further increasing a data owner's liability. This paper uses Dragchute systems to mitigate damage caused by data breaches. We provide a construction that protects data for a predetermined window of time even when all of its stored values become compromised. Even still, we show the ability to verify that such protection is in place, should the data be legally required, and demonstrated that such a solution can be efficiently deployed, by adding negligible overhead to traditional encryption methods. In so doing, we argue that Dragchute represents an important means of mitigating the risks associated with the long-term storage of forced-retention data.

[10] We note, however, that our method can be used to lock data for years. Moroever, we can envision scenarios unrelated to forced-retention data (e.g., an author who wishes to lock unpublished manuscripts until several years after her death) in which a party might wish to lock data for such long periods.

ACKNOWLEDGEMENTS

The authors would like to thank our anonymous reviewers for their helpful comments. We would also like to thank Jessica Erickson, Hank Chambers, and John Douglas for their guidance in the legal aspects of this work. The icons found in Figure 2 were created by Kidiladon, AliWijaya, B Farias, Guru, and Alena Artemova from the Noun Project under the Creative Commons 3.0 License. Finally, this work was supported by the National Science Foundation grant numbers CNS-1562485 and CNS-1540217. Anyfi ndings, comments, conclusion found in this work are from the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 104th Congress of the United States of America. 1996. PUBLIC LAW 104-191 - Health Insurance Portability and Accountability Act of 1996. https://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm. (1996).

[2] 107th Congress of the United States of America. 2002. PUBLIC LAW 107-204 - Sarbanes Oxley Act of 2002. http://www.sec.gov/about/laws/soa2002.pdf. (2002).

[3] 93rd Congress of the United States of America. 1974. PUBLIC LAW 93-495 - An Act To Increase Deposit Insurance From $20,000 To $40,000, To Provide Full Insurance For Public Unit Deposits Of $100,000 Per Account, To Establish A National Commission On Electronic Fund Transfers, And For Other Purposes; Equal Credit Opportunity Act; Fair Credit Billing Act. http://purl.fdlp.gov/GPO/gpo51530. (1974).

[4] 94th Congress of the United States of America. 2011. PUBLIC LAW 94-200 - Home Mortgage Disclosure Act of 1975. https://www.gpo.gov/fdsys/pkg/USCODE-2011-title12/pdf/USCODE-2011-title12-chap29.pdf. (2011).

[5] Hal Abelson, Ross Anderson, Steven M. Bellovin, Josh Benalob, Matt Blaze, Whitfield Diffie, John Gilmore, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, and Bruce Schneier. 1997. The Risks of Key Recovery, Key Escrow, and Trusted Third-party Encryption. *World Wide Web J.* 2, 3 (June 1997), 241–257.

[6] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2087–2104. https://doi.org/10.1145/3133956.3134104

[7] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. *Deterministic and Efficiently Searchable Encryption.* Springer Berlin Heidelberg, Berlin, Heidelberg, 535–552. https://doi.org/10.1007/978-3-540-74143-5$_3$0

[8] Mihir Bellare and Shafi Goldwasser. 1996. *Encapsulated Key Escrow.* Technical Report.

[9] Mihir Bellare and Shafi Goldwasser. 1997. Verifiable Partial Key Escrow. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS '97)*. ACM, New York, NY, USA, 78–91. https://doi.org/10.1145/266420.266439

[10] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Advances in Cryptology?ASIACRYPT 2000* (2000), 531–545.

[11] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. 1998. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Trans. Database Syst.* 23, 3 (sep 1998), 231–285.

[12] Alex Biryukov and Dmitry Khovratovich. 2016. Egalitarian Computing. In *Proceedings of the USENIX Security Symposium (SECURITY)*.

[13] L Blum, M Blum, and M Shub. 1986. A Simple Unpredictable Pseudo Random Number Generator. *SIAM J. Comput.* 15, 2 (may 1986), 364–383.

[14] Dan Boneh and Moni Naor. 2000. Timed Commitments. In *Advances in Cryptology — CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, August 20–24, 2000 Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 236–254. https://doi.org/10.1007/3-540-44598-6$_1$5

[15] S. Bowe. 2016. pay-to-sudoku. https://github.com/zcash/pay-to-sudoku. (2016).

[16] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. 2017. Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 229–243. https://doi.org/10.1145/3133956.3134060

[17] Julien Cathalo, Benoît Libert, and Jean-Jacques Quisquater. 2005. Efficient and Non-interactive Timed-Release Encryption. In *Information and Communications Security*, Sihan Qing, Wenbo Mao, Javier López, and Guilin Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–303.

[18] Konstantinos Chalkias and George Stephanides. 2006. Timed Release Cryptography from Bilinear Pairings Using Hash Chains. In *Communications and Multimedia Security*, Herbert Leitold and Evangelos P. Markatos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 130–140.

[19] Hank Chambers. 2018. personal communication. (July 2018). Professor of Law, University of Richmond.

[20] Wu chang Feng, Ed Kaiser, Wu chi Feng, and Antoine Luu. 2005. Design and Implementation of Network Puzzles. (2005).

[21] David Chaum and Torben P. Pedersen. 1993. Wallet Databases with Observers. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '92)*. Springer-Verlag, London, UK, UK, 89–105.

[22] Jung Hee Cheon, Nicholas Hopper, Yongdae Kim, and Ivan Osipkov. 2006. Timed-release and key-insulated public key encryption. In *International Conference on Financial Cryptography and Data Security*. Springer, 191–205.

[23] Jung Hee Cheon, Nicholas Hopper, Yongdae Kim, and Ivan Osipkov. 2008. Provably Secure Timed-Release Public Key Encryption. *ACM Trans. Inf. Syst. Secur.* 11, 2, Article 4 (may 2008), 44 pages. https://doi.org/10.1145/1330332.1330336

[24] R. Chirgwin. 2018. Equifax reveals full horror of that monstrous cyber-heist of its servers. https://www.theregister.co.uk/2018/05/08/equifax$_b$reach$_m$ay$_2$018/. (2018).

[25] Drew Dean and Adam Stubblefield. 2001. Using Client Puzzles to Protect TLS. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, 1–1.

[26] Giovanni Di Crescenzo, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. 1999. Conditional Oblivious Transfer and Timed-Release Encryption. In *Advances in Cryptology — EUROCRYPT '99*, Jacques Stern (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 74–89.

[27] John G. Douglas. 2018. personal communication. (July 2018). Professor of Law (and former Dean), University of Richmond, former Chief of the Criminal Division of the United States Attorney's Office in Richmond, Virginia.

[28] C Dwork and M Noar. 1992. Pricing Via Processing or Combating Junk Mail. In *Advances in Cryptology (CRYPTO)*.

[29] Yuval Elovici, Ronen Waisenberg, Erez Shmueli, and Ehud Gudes. 2004. *A Structure Preserving Database Encryption Scheme.* Springer Berlin Heidelberg, Berlin, Heidelberg, 28–40. https://doi.org/10.1007/978-3-540-30073-1$_3$

[30] Jessica Erickson. 2017. personal communication. (Dec 2017). Professor of Law, University of Richmond.

[31] Federal Court Rules Committee. 1938. Federal Rules of Civil Procedure Rule 34: Producing Documents, Electronically Stored Information, and Tangible Things, or Entering onto Land, for Inspection and Other Purposes (most recently ammended April, 2015). (1938).

[32] Amos Fiat and Adi Shamir. 1987. *How To Prove Yourself: Practical Solutions to Identification and Signature Problems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 186–194.

[33] Juan A. Garay and Markus Jakobsson. 2003. *Timed Release of Standard Digital Signatures.* Springer Berlin Heidelberg, Berlin, Heidelberg, 168–182.

[34] Roxana Geambasu, Tadayoshi Kohno, Arvind Krishnamurthy, Amit Levy, Henry Levy, Paul Gardner, and Vinnie Moscaritolo. 2010. New directions for self-destructing data systems. *University of Washington, Tech. Rep* (2010).

[35] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing Data Privacy with Self-Destructing Data.. In *USENIX Security Symposium*. 299–316.

[36] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 626–645.

[37] Government of the State of California. 1980. California Consitution; Government Code, Government of the State of California, Executive Department, Department of Fair Employment and Housing, Chapter 6, Section 12946. http://leginfo.legislature.ca.gov/faces/codes$_d$isplaySection.xhtml?lawCode=GOV&sectionNum=12946. (1980).

[38] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/1180405.1180418

[39] William Greenwood. 2016. Verizon claims $1 billion discount from Yahoo for data breach. http://thetechnews.com/2016/10/07/verizon-claims-1-billion-discount-yahoo-data-breach/. (2016).

[40] Roger A. Grimes. 2015. Your 'Offline' Storage May be Putting You at Risk. Infoworld. (June 16, 2015). http://www.infoworld.com/article/2935613/data-security/your-offline-storage-may-be-putting-you-at-risk.html

[41] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium (SECURITY)*.

[42] Elizabeth A. Harris. 2013. For Target's Shoppers, a New Holiday To-Do List. http://www.nytimes.com/2013/12/20/business/for-targets-shoppers-a-new-holiday-to-do-list.html. (December 2013).

[43] D. Hristu-varsakelis, K. Chalkias, and G. Stephanides. 2008. A versatile secure protocol for anonymous timed-release encryption. In *In Journal of Information Assurance and Security 2 (2008.* Dynamic Publishers, Inc, 80–88.

[44] Ari Juels and John Brainard. 1999. Client Puzzle: A Cryptographic Defense Against Connection Depletion Attacks. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*.

[45] J. Katz and Y. Lindell. 2014. *Introduction to Modern Cryptography (2nd Edition)*. Chapman & Hall/CRC Press.

[46] Jeremy Krik. 2015. How Encryption Keys Could be Stolen By Your Lunch. Computerworld. (June 22, 2015). http://www.computerworld.com/article/2938753/security/how-encryption-keys-could-be-stolen-by-your-lunch.html

[47] Ben Laurie and Richard Clayton. 2004. Proof of Work Proves Not to Work. In *Proceedings of the Workshop on the Economics of Information Security*.

[48] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. 2011. Big data: The next frontier for innovation, competition, and productivity. http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/big-data-the-next-frontier-for-innovation. (June 2011).

[49] Wenbo Mao. 2001. Verifiable Partial Escrow of Integer Factors. *Designs, Codes and Cryptography* 24, 3 (01 Dec 2001), 327–342. https://doi.org/10.1023/A:1011235607071

[50] Srijith K Nair, Mohammad T Dashti, Bruno Crispo, and Andrew S Tanenbaum. 2007. A hybrid PKI-IBC based ephemerizer system. In *IFIP International Information Security Conference*. Springer, 241–252.

[51] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. http://www.bitcoin.org/bitcoin.pdf. (2008).

[52] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. 2014. Reconsidering generic composition. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 257–274.

[53] Ivan Osipkov, Yongdae Kim, and Jung Hee Cheon. 2004. New Approaches to Timed-Release Cryptography. (10 2004).

[54] Damien Paletta. 2015. Wall Street Journal - OPM Breach Was Enormous, FBI Director Says. http://www.wsj.com/articles/breach-was-enormous-fbi-director-says-1436395157. (2015).

[55] Radia Perlman. 2005. The ephemerizer: Making data disappear. (2005).

[56] Huseyin Cavusoglu Ph.D., Birendra Mishra Ph.D., and Srinivasan Raghunathan Ph.D. 2004. The Effect of Internet Security Breach Announcements on Market Value: Capital Market Reactions for Breached Firms and Internet Security Developers. *International Journal of Electronic Commerce* 9, 1 (2004), 70–104.

[57] Niels Provos and David Mazieres. 1999. A Future-Adaptable Password Scheme. In *Proceedings of the USENIX Security Symposium (USENIX)*.

[58] M. O. Rabin. 1979. *Digitalized Signatures and Public-key Functions as Intractable as Factorization.* Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA.

[59] Jothi Rangasamy, Douglas Stebila, Colin Boyd, Juan Manuel González-Nieto, and Lakshmi Kuppusamy. 2012. Effort-Release Public-Key Encryption from Cryptographic Puzzles. In *Information Security and Privacy*, Willy Susilo, Yi Mu, and Jennifer Seberry (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[60] Fahmida Rashid. 2015. Got 'Em! Researchers Steal Keys from Amazon Cloud. Infoworld. (September 30, 2015). http://www.infoworld.com/article/2987109/cloud-security/researchers-steal-crypto-keys-from-amazon-cloud.html

[61] R. L. Rivest, A. Shamir, and D. A. Wagner. 1996. *Time-lock Puzzles and Timed-release Crypto.* Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA.

[62] Jordan Robertson. 2014. Heartbleed Hackers Steal Encryption Keys in Threat Test. Bloomberg Technology. (April 15, 2014). https://techcrunch.com/2015/02/19/the-nsa-reportedly-stole-millions-of-sim-encryption-keys-to-gather-private-data/

[63] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *Computer* 29, 2 (Feb. 1996), 38–47. https://doi.org/10.1109/2.485845

[64] Jeremy Scahill and Josh Begley. 2015. The Great SIM Heist: How Spies Stole the Keys to the Encryption Castle. The Intercept. (February 19, 2015). https://theintercept.com/2015/02/19/great-sim-heist/

[65] Securities and Exchange Commission. 2003. Final Rule: Retention of Records Relevant to Audits and Reviews. 17 CFR Part 210, RIN 3235-AI74. (January 2003). https://www.sec.gov/rules/final/33-8180.htm

[66] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613.

[67] Erez Shmueli, Ronen Vaisenberg, Yuval Elovici, and Chanan Glezer. 2010. Database Encryption: An Overview of Contemporary Challenges and Design Considerations. *SIGMOD Rec.* 38, 3 (Dec. 2010), 29–34.

[68] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00)*. IEEE Computer Society, Washington, DC, USA.

[69] Gina Stevens. 2014. The Federal Trade Commission's Regulation of Data Security Under Its Unfair or Deceptive Acts or Practices (UDAP) Authority. *Congressional Research Service* 11 (2014).

[70] Mark W Storer, Kevin M Greenan, Ethan L Miller, and Kaladhar Voruganti. 2008. POTSHARDS: secure long-term storage without encryption. In *2007 USENIX Annual Technical Conference*. USENIX Association.

[71] Arun Subbiah and Douglas M Blough. 2005. An approach for fault tolerant and secure data storage in collaborative work environments. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*. ACM, 84–93.

[72] Paul F. Syverson. 1998. Weakly Secret Bit Commitment: Applications to Lotteries and Fair Exchange. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop, Rockport, Massachusetts, USA, June 9-11, 1998*. 2–13.

[73] Rahul Telang. 2015. Policy Framework for Data Breaches. *IEEE Security & Privacy* 13, 1 (2015), 77–79.

[74] The Massachusetts Society of Certified Public Accountants: Federal Taxation Committee. 2004. The Record Retention Guide. https://www.cpa.net/resources/retengde.pdf. (2004).

[75] XiaoFeng Wang and Mike Reiter. 2003. Defending Against Denial-of-Service Attacks with Puzzle Auctions. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[76] H. C. Williams and B. Schmid. 1979. Some remarks concerning the M.I.T. public-key cryptosystem. *BIT Numerical Mathematics* 19, 4 (1979), 525–538.

[77] Scott Wolchok, Owen S. Hofmann, Nadia Heninger, Edward W. Felten, J. Alex Halderman, Christopher J. Rossbach, Brent Waters, and Emmett Witchel. 2010. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs. In *Proceedings of the Network and Distributed System Security Symposium*.

[78] Jay J Wylie, Michael W Bigrigg, John D Strunk, Gregory R Ganger, Han Kiliccote, and Pradeep K Khosla. 2000. Survivable information storage systems. *Computer* 33, 8 (2000), 61–68.

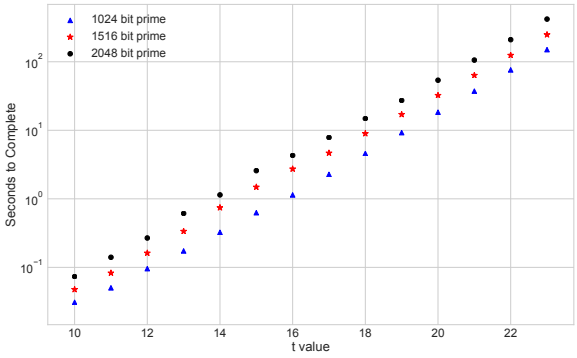

Figure 11: Strong primes with longer lengths have slower time to complete for the Lock **function.**

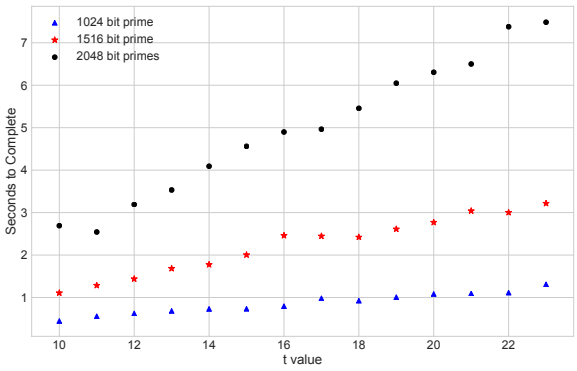

Figure 10: All prime lengths can achieve the overall goal of time needed to unlock by varying the value of t.

A CHOOSING THE SIZE OF THE STRONG PRIMES

In this appendix, we show the performance of the Dragchute Lock operation based on the size of the prime numbers used. We used strong primes of length 1024, 1516, and 2048 bits to measure the time required to complete Lock and Unlock.

Figure 10 shows times for Unlock for varying prime sizes. As expected, Unlock grows exponentially for all tested prime sizes, with the three curves almost identically shaped, but at slightly differing magnitudes. Since a primary goal of Dragchute is to encrypt afile such that the unlock time is predictable, any choice of prime size tested suffices. Moreover, the same overall results can be achieved with any reasonably sized prime, by varying time parameter t. For example, locking afi le for about 2 minutes can be realized using 1024, 1516, or 2048 bit strong primes with t values of 23, 22, and 21 respectively.

Similarly, Figure 11 shows Lock times as a function of the size of the underlying strong primes. Assuming afi le that needs to be locked for at least two minutes, locking requires approximately 1, 2, and 6 seconds for primes of length 1024, 1516, and 2048 bits respectively. We use 1024 bit strong primes in our prototype implementation, and in our analysis throughout the paper, since this offers the best performance while still providing a sufficient level of security.

B APPLICATION CONSIDERATIONS

Extending Data-Access Latency. The value of data is likely to decrease as it ages. For instance, the relevance of user's tax return from last year is higher than those from three years ago. Accordingly, the need to access such data quickly may further degrade with time.

This changing relevance of data can easily be accommodated by a Dragchute system. Naïvely, the administrator could simply unlock the protected data and then perform the lock function on it again with the newly desired value of t. This approach unnecessarily forces the administrator to spend the needless effort to unlock (and potentially expose) the data. Instead, we propose that the administrator simply lock the current ciphertext C with new parameters such that the summation of unlock times for the two layers is equal to the newly desired access delay period.

The above approach is sufficient as long as it is not deemed necessary that a party performing Attest needs to be able to reason about the well-formedness of a multiply-locked ciphertext. Should, on the other hand, a party require attestation for a commit that covers the full duration during which the data is unavailable (rather than being satisfied with two commits, the "innermost" of which can only be verified after the "outermost" commit has been opened), the only possible solution, given that the private unlocking parameters have been discarded, is one in which commits for several durations are computed during the initial commit, when private parameters remain available. Note that this does not pose a security threat – simultaneously storing Dragchute primitives (e.g., locked container X and witness T) corresponding to access times of, say, one month, two months, four months, etc., does not provide access quicker than one month. When the time comes for access latency to be increased to two months, the one month Dragchute primitives are securely discarded.

Selecting Values of t. While we have discussed the performance overhead for our various functions, there may be additional considerations for selecting the amount of time necessary to pick a certain value of t. Using the example of previous years' taxes, an administrator may wonder which value of t is appropriate for their application. There are two critical points to consider. First, the administrator should seek to balance the likelihood and frequency of the data will be needed against their ability to detect a compromise. For instance, a company using an expensive intrusion detection system (IDS) built to watch a singular data type may be able to use relatively small values of t, whereas a less sophisticated operation may need to err on the side of caution and give themselves additional time. In either case, a Dragchute instantiation (along with desired t value) should be used to complement the IDS already in place.

Secondly, an administrator may eventually need to consider legal mandates. While no such mandates exist now because the construction is new, it is not unforeseeable that courts may prescribe rules (i.e., maximum unlock times) to specific laws.

Accordingly, like all other pieces of security infrastructure, these considerations and the experience of the administrator will need to be balanced to select the proper parameters.

Granularity of Dragchute Commit Durations. The Dragchute scheme as described here presents limits on the duration of commitments. To see why this might be the case, consider that if time parameter t provides commit duration Δ, then time parameter $t+1$ will provide commit duration of 2Δ. A problem arises if one wishes, for example, to generate a commit with time duration $\frac{3}{2}\Delta$. This arises, in part, due to the way in which we have defined the time parameter t. Specifically, because we define u to be $g^{2^{2^t}}$, choosing time parameter value t means requiring 2^t sequential squares to unlock the commit. Had we instead defined u as g^{2^t}, then a time parameter value of t would require exactly t squarings, allowing a much wider range of possible commit times.

Defining t as we have, however, has a significant benefit with regards to the non-interactive zero-knowledge proof. In particular, in order to show that the value u has been correctly constructed, we use the vector

$$W = \left\langle g^2, g^4, g^{16}, g^{256}, \ldots, g^{2^{2^i}}, \ldots, g^{2^{2^{(t-1)}}}, g^{2^{2^t}} \right\rangle,$$

which we denoted by $W = \langle b_0, b_1, \ldots, b_t \rangle$, and show via our NIZKP that each triple of the form (g, b_{i-1}, b_i) is a Diffie-Hellman tuple.

It turns out that this same technique works, regardless of how t is defined, with some minor modifications to W. As an example, suppose one wishes to require $\frac{3}{2}2^t$ sequential squarings (halfway between 2^t and 2^{t+1} squarings). Then W can be constructed as follows:

$$W = \left\langle g^8, g^{64}, \ldots, g^{2^{\frac{3}{2}2^i}}, \ldots, g^{2^{\frac{3}{2}2^{(t-1)}}}, g^{2^{\frac{3}{2}2^t}} \right\rangle,$$

(here i starts at 1). This modified W works just as well as the previous definition, since each triple of the form (g, b_{i-1}, b_i) is a Diffie-Hellman tuple, and can thus be verified using the NIZKP presented earlier.

Data with Time-Limited Value. One of the advantages of delaying access to data is that the delay itself may allow a defender to mitigate the loss of that data. For instance, if the data itself has time-limited value, the compromised party can simply render such data irrelevant immediately.

Consider credit card information as an example. Many companies store such information related to their customers for many years, with no immediate, but some potential, future benefit in mind (e.g., a company may hope to predict which kind of sales bring customers back to their stores). Accordingly, credit card data may be stored long term without any short-term use. Such data would also be of interest to an adversary, and therefore a candidate for protection via Dragchute .

The current standard response to detecting a breach of such data is tofi rst cancel such credit card numbers and then to offer credit protection to anyone potentially impacted by the breach. The latter step is an expensive proposition, costing potentially millions of dollars depending on the size of the breach. If the compromised vendor were instead to be able to proactively have the cards replaced by their banks (i.e., using a database where customer name and bank are in plaintext, but the credit card number is protected using a Dragchute scheme), the credit card numbers could be rendered worthless prior to an adversary's ability to unlock them.

We believe that many types of data could potentially benefit from such protection, especially those used for account access (e.g., bank account numbers, passwords, etc).

Data with No Retirement Date. Not all data loses its usefulness with time. Social security numbers, communications covered by Sarbanes-Oxley, and even corporate planning documents may serve as valuable information to an attacker regardless of when they were originally stored. Accordingly, such data can not simply be proactively expired as was discussed previously.

Regardless, having lead time in addressing a breach remains valuable. One of the major challenges of breaches is in rapidly responding to the event. With additional time between data breach and data compromise, victims can better adjust their public response (e.g., by proactively offering social security number monitoring services instead of doing so reactively) and legal strategies accordingly.

Deployment Scenarios. To this point we have discussed Dragchute in a context independent manner, focusing implicitly on a simple scenario: a single machine with a single copy of sensitive data. While a valuable context in which to reason about our primitives, our methods are potentially applicable to a variety of deployment scenarios at a multitude of system levels, each of which presents unique advantages and concerns. This raises numerous questions, as for example about the granularity at which our primitives should be applied (e.g., directory level or tablespace level), and of configuration appropriate for various threat models. Data distribution models also influence deployment configuration – deployment in a modern networked system raises issues very different from those encountered in a disk level solution.

Moreover, the majority of modern enterprises have extensive storage and backup systems. These systems often include the creation of multiple copies of all data, many of which are stored off-site to ensure that even physical destruction of a facility will not result in the loss of criticalfi les. The creation of such copies, each of which may be stored on a machine with potentially different administrators and/or policies (e.g., a cloud system may not provide the same kinds of policy enforcement mechanisms, and may itself be subject to a variety of different laws than a domestically located company) complicates the problem of implementation and deployment of Dragchute (as it would for any solution).

A naive solution to the backup issue might be to simply use offline storage. While true offline storage is in regular use in some industries (e.g., traditional banking), most of what is currently termed offline storage is not actually offline [40], but instead hosted remotely on a virtual host that is "shut down" (and that can be turned on remotely by connecting through the VM management or a "lights-out" systems).

We consider the exploration of such complexities to be an important component of our future Dragchute-related work.