ArchiveSafe: Mass-Leakage-Resistant Storage from Proof-of-Work

Moe Sabry[1(✉)], Reza Samavi[2(✉)], and Douglas Stebila[3(✉)]

[1] McMaster University, Hamilton, Canada
alym2@mcmaster.ca
[2] Vector Institute, Ryerson University, Toronto, Canada
samavi@ryerson.ca
[3] University of Waterloo, Waterloo, Canada
dstebila@uwaterloo.ca

Abstract. Data breaches—mass leakage of stored information—are a major security concern. Encryption can provide confidentiality, but encryption depends on a key which, if compromised, allows the attacker to decrypt everything, effectively instantly. Security of encrypted data thus becomes a question of protecting the encryption keys. In this paper, we propose using *keyless encryption* to construct a *mass leakage resistant archiving system*, where decryption of a file is only possible after the requester, whether an authorized user or an adversary, completes a *proof of work* in the form of solving a cryptographic puzzle. This proposal is geared towards protection of infrequently-accessed *archival data*, where any one file may not require too much work to decrypt, decryption of a large number of files—mass leakage—becomes increasingly expensive for an attacker. We present a prototype implementation realized as a user-space file system driver for Linux. We report experimental results of system behaviour under different file sizes and puzzle difficulty levels. Our keyless encryption technique can be added as a layer *on top of* traditional encryption: together they provide strong security against adversaries without the key and resistance against mass decryption by an attacker.

1 Introduction

Attacks on information systems have become increasingly common. Whatever the attack vector, a frequent outcome is a *data breach*, in which a large volume of sensitive information is stolen from the victim organization. Archival data—stored indefinitely but not regularly accessed—has been targeted in many data breaches [4,14,18], leading to loss of privacy, loss of reputation, business setbacks, and costly remediation.

Modern IT security protection techniques focus on *defense-in-depth*, one component of which is encryption of data at rest to support confidentiality. However, encryption, even when implemented using secure, carefully implemented algorithms, is typically all-or-nothing: if the key is secure, the attacker learn virtually

J. Garcia-Alfaro et al. (Eds.): DPM 2020/CBT 2020, LNCS 12484, pp. 89–107, 2020.
https://doi.org/10.1007/978-3-030-66172-4_6

nothing, and the attack cannot succeed, but once the key is compromised, the attacker can decrypt everything, with minimal overhead.

Hardware-assisted cryptography, such as hardware security modules (HSMs), trusted computing, or secure enclaves like Intel SGX[1] or ARM TrustZone[2] may prevent keys from leaking if decryption is only ever done inside a trusted module, but many IT systems remain software-only without use of these technologies.

Scenario and Goals. Against these types of threats, we aim develop a *mass leakage resistant archiving system* with the goal of enhancing defense-in-depth for encryption. We aim to preserve confidentiality even in the presence of an adversary with full access to the system, including ciphertexts and decryption keys. While no system can provide full cryptographic security in the face of such a well-informed adversary, our goal is to increase the economic cost of *mass leakage*, which for our purposes is defined as an adversary obtaining the plaintexts of a large number of files or database records, not just one.

Unlike most applications of cryptography, we do not aim to achieve a difference in work factor between honest parties and adversaries. Rather, we assume that honest parties and adversaries have different *goals*, and we aim to change the economics of data breaches by achieving a difference in the cost of honest parties and adversaries achieving their goals. In our scenario, honest parties need to store a large number of files, but only access a small number of them. Consider for example a tax agency: after processing millions of citizens' tax returns each year, those files must be stored for several years in case an audit or further analysis is required, but only a small fraction of those records will end up actually being pulled for analysis. In contrast, an adversary breaching the tax agency's records may want to read a large number of files to identify good candidates for identity theft or other criminal actions.

1.1 Contributions

We design a system, called *ArchiveSafe*, where access to a resource is only possible after the requester—whether an honest user on adversary—has expended sufficient computational effort, in the form of solving a "moderately hard" proof-of-work or cryptographic puzzle [10]. Since we will not rely on the access control system nor any keys to be uncompromised, the decryption operation itself must be tied to the cryptographic puzzle. In our approach, while a proper cryptographic key is used to encrypt a file, the encryption key is not stored, even for legitimate users. Instead, the key is wrapped in a proof-of-work-based encryption scheme with a desired difficulty level, and all users—adversarial or honest—must perform the proof-of-work to recover the key and then decrypt the file.

Our main technical tool for building of ArchiveSafe is a new cryptographic primitive that we call *difficulty-based keyless encryption* (DBKE), which is an encryption scheme that does not make use of a stored key. We give a generic

[1] https://software.intel.com/en-us/sgx.
[2] https://developer.arm.com/ip-products/security-ip/trustzone.

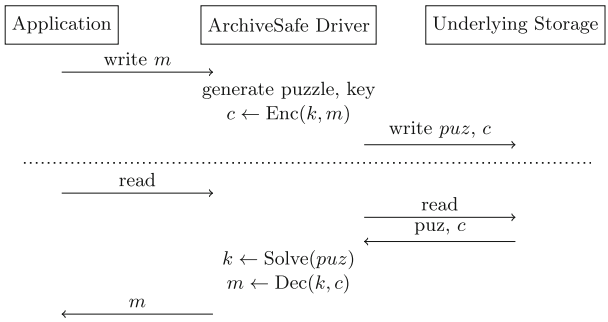

Fig. 1. High-level overview of ArchiveSafe, showing a write followed by a read.

construction for DBKE from a standard symmetric encryption scheme and a new tool called *difficulty-based keyless key wrap*, which wraps the symmetric encryption key in an encapsulation that can only be unwrapped by performing a sufficiently high number of operations, as in a proof-of-work scheme. Difficulty-based keyless key wrap can be achieved from many types of cryptographic puzzles, and we show one example based on hash function partial pre-image finding [11,12]. One interesting feature of using this form of hash-based puzzle, which to our knowledge is a novel observation on hash-based puzzles, is that the puzzle and ciphertext can be *degraded*—i.e., turned into a harder one—essentially for free. We use the reductionist security methodology to formalize the syntax and security properties of difficulty-based keyless encryption and keyless key wrap and show that our hash-based construction achieves these properties.

Figure 1 gives a high-level overview of how an application interacts with the ArchiveSafe system. The two main operations performed by the ArchiveSafe system are (i) creating a puzzle and encrypting during writes, and (ii) solving the puzzle and decrypting during reads. ArchiveSafe could be used in a variety of data storage architectures: on a local computer; on a file server; or in a cloud architecture. In a file server or cloud scenario, an IT system may be set up so the file server enforces that all files are protected by ArchiveSafe during writes by centralizing puzzle creation and encryption, but leaves puzzle solving and decryption to clients. Since puzzle creation and encryption in our system is cheap, this avoids bottlenecks on the file server. Individual client applications occasionally reading a small number of files have to do a moderate, but not prohibitive, amount of work to solve the puzzle to obtain the key to decrypt.

We build a prototype implementation showing the use of ArchiveSafe on a local computer. Our prototype is implemented as a filesystem-in-userspace (FUSE) driver on Linux. A FUSE driver can be used to intercept I/O operations in certain directories (mount points) before reading/writing to disk. This allows us to implement ArchiveSafe in a manner that is transparent to the application, as well as transparent to the underlying storage mechanism, which could be a local disk (with normal disk encryption enabled or not), or a network share mounted locally. We validate the performance of our prototype implementation,

focusing primarily on ensuring that write operations incur minimal overhead. (Since system administrators can set policies with puzzle difficulties requiring seconds or minutes of computational effort to solve, slow read performance is *intended*, and there is little sense in performance measurements on reads, beyond checking that they scale as intended with no unexpected overhead.) We envision that, when used on a local computer, ArchiveSafe would be applied only to a subset of the directories on the computer. One might use ArchiveSafe to protect documents created by the user more than a certain number of days ago, but would not use it on system libraries and executables.

We highlight that ArchiveSafe is meant to add *defense-in-depth* to confidentiality: one would typically not rely on ArchiveSafe alone, but combine it with traditional encrypted file system or database encryption. In this combination, traditional encryption using strong algorithms and keys, provides a high level of security if the keys are not compromised, but we still have the difficulty-based keyless encryption of ArchiveSafe as a bulwark if the keys are compromised. To succeed under this setup, the adversary must compromise the traditional encryption keys in addition to solving a large number of DBKE puzzles corresponding to the files in the archive.

1.2 Related Work

Filesystem Encryption. Blaze [6] introduced the Cryptographic File System (CFS). CFS uses a different key for each directory, and the user is required to enter the key in every session to access the directory and its contents. Subsequent proposals include the Transparent Cryptographic File System (TCFS) [7], Cryptfs [24] and Ncryptfs [23]. In recent years, encrypted filesystems have become widespread, and all major operating systems provide implementations, often enabled by default (FileVault on Apple's macOS[3], BitLocker on Microsoft Windows[4], and a range of options on Linux such as Linux Unified Key Setup (LUKS)[5]). The common practice in these technologies is to use a single master key from which multiple keys are derived per-file, per-directory, or per-sector; the master key is usually stored on the device itself, encrypted under the user's password. Once the user has logged in, the filesystem transparently and automatically decrypts files.

Over the past decade, there has been much research on encrypted databases (e.g., [13,15,16]) that retain some functionality for legitimate users, for example using order-preserving encryption so that sorting a column of ciphertexts yields approximately the same order as if the plaintexts were sorted. This increased functionality comes at the cost of information leakage, and there is an extensive debate in the literature about these techniques.

[3] https://support.apple.com/en-ca/HT204837.
[4] https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview.
[5] https://guardianproject.info/archive/luks/.

Proof-of-Work Systems. Dwork and Naor [10] introduced client puzzles to control junk email: recipients would only accept emails if the sender was able to solve a puzzle. It should be "moderately hard" for the sender to solve the puzzle, but easy for recipient to check whether a solution is valid. This was the first example of a proof-of-work system, which in general grants access to a resource dependent on the requester being able to demonstrate proof that they have performed some work, typically in the form of solving a puzzle. Client puzzles were for many years suggested as a means to prevent denial of service attacks in a range of contexts [2,8,12,17,20,22], but have seen renewed interest as a building block for cryptocurrencies and blockchains. Client puzzles are generally classified either based on their limiting factors in solving the puzzle (CPU-bound versus memory-bound) or based on whether the operations required to solve the puzzle is parallelizable. The simplest CPU-bound puzzles are based on cryptographic hash functions, such as: finding a preimage of a hash given a hint (e.g., a part of the preimage) [11,12]; or finding an input whose hash starts with a certain number of zero bits [3]. Non-parallelizable CPU-bound puzzles often rely on a number of theoretical approaches. For example, [19] uses repeated squaring modulo an RSA modulus. Memory-bound puzzles [1,9] use techniques for which the best known solving algorithm involves a large number of memory accesses; it is argued that memory access time varies less than CPU speed between small and large computing platforms, and that building customized hardware is more expensive for memory-bound puzzles.

Proof-of-Work Systems for Confidentiality. In [19], time-lock encryption was proposed as a way of "sending information into the future", and focused specifically on hiding keys or data in a proof-of-work system that had a predictable wall-clock time for solving, thus focusing on puzzles for which the best known solving algorithm is inherently sequential. Vargas et al. [21] designed a database encryption system called "Dragchute" based on time-lock encryption, aiming to provide both confidentiality and the ability to demonstrate compliance with retention laws. Each ciphertext in this system is accompanied by an authentication tag which contains a non-interactive zero-knowledge proof. Solving the puzzle will yield a valid decryption key for the ciphertext; moreover, the proof can be checked much more efficiently than the full work required to solve and decrypt the ciphertext. A simpler database encryption scheme relying on hash-based client puzzles, without any efficient verification of well-formedness, was proposed by Moghimifar [13].

2 Requirements

In this section, we discuss the functionality and security requirements for a mass leakage resistant archiving system, which informs our construction and evaluation in subsequent sections.

2.1 Design Criteria

Confidentiality in the Face of Compromised Keys. The system should achieve some level of confidentiality even if all stored keys are compromised. This means we assume that an adversary can learn a symmetric key or a private key corresponding to a public key stored for later use in decrypting a ciphertext, even if the key is stored in a separate key management service, trusted computing or secure enclave environment, or separate tamper-resistant device.

Cooperation with Traditional Encryption. It should be possible to use the system in conjunction with the traditional encryption mechanisms applied to storage systems (folder/disk encryption, database encryption, etc.), so that strong confidentiality is achieved if keys are not compromised, but some confidentiality is retained in the face of compromised keys.

Reliance on Industry Standard Cryptographic Algorithms. Deployed IT systems should rely only on well-vetted, standardized cryptographic algorithms. But all such algorithms for achieving confidentiality—public key or symmetric—require a secret key, seemingly conflicting with the first design criteria of confidentiality in the face of compromised keys. Our construction builds a mechanism for confidentiality without keys while still relying on standard cryptographic algorithms like AES for symmetric encryption: while a proper cryptographic key is used to encrypt data, that key is not kept, even by authorized users. Instead, the key is wrapped in a proof-of-work-based encryption scheme with a desired difficulty level, and users must solve the proof-of-work to recover the key and then decrypt the data. We introduce difficulty-based keyless encryption in Sect. 3 which formalizes this idea and generically construct it from standard cryptographic algorithms such as AES and Argon2.

Imposing a Significant Cost to Access a Large Number of Files While Maintaining Acceptable Cost to Access One File. Since we do not have a key that gives honest users an advantage over the adversary, we should look at things from the viewpoint of typical honest behaviour—periodically accessing a small number of files—versus adversary behaviour—accessing a large number of files in a data breach. Proof-of-work and related techniques have long been used to achieve security goals from that viewpoint, whether in password hardening or client puzzles for denial of service resistance.

Customizing File Access Cost. It should be possible for a system administrator or user to control the cost incurred by the adversary or honest user for accessing a file. This may be set as a system-wide policy or a file-by-file basis, depending on the desired access control paradigm. This is achieved in our system by varying the difficulty level of the puzzle wrapping the decryption key.

A related design criteria is the ability to customize file access cost *over time*. Demand for access to records may change over time; for example, records older than 5 years may be accessed much less frequently than more recent records. Our system allows the file access cost to be increased with minimal effort, through a process we call *puzzle degradation*, that could be performed as part of regular

system maintenance. This is a novel feature available from some types of puzzle constructions but not others, and in particular not from the number-theoretic repeated squaring non-parallelizable constructions used in time-lock puzzles [19] and the Dragchute database encryption system [21].

2.2 Choice of Puzzle

One of the major design decisions for our system is which type of puzzles to use: sequential versus parallelizable, and CPU-bound versus memory-bound.

As our design criteria focus on mass leakage adversaries trying to decrypt *many* files, and since we think of cost in a general economic sense, we do not have to restrict to proof-of-work mechanisms that are sequential/non-parallelizable. Concerned with an adversary trying to decrypt many files who has parallel computing resources available to them, it does not matter whether they choose to deploy their parallel resources to sequentially decrypt each file quickly or in parallel decrypt many files more slowly. Overall, they will decrypt the same number of files with the same resources. We also need not worry about the variability of puzzle solving time for individual instances, only the expected puzzle solving time for many instances. These design choices are, for example, significantly different from those of the Dragchute system for database confidentiality and integrity from proof-of-work. Moreover, parallelization permits honest users to reduce the latency in occasional access of files by taking advantage of short, on-demand use of cloud servers (see Table 3).

Whereas sequential versus parallelizable puzzles is a qualitative choice for our scenario, CPU-bound versus memory-bound is a quantitative choice with respect to the economic cost. To achieve a given dollar-cost-for-adversary, it is possible to pick appropriate parameters for both CPU-bound and memory-bound puzzles under appropriate cost and puzzle-solving assumptions. So, a priori, either can be used in our constructions. For our prototype we choose simple hash-based CPU-bound puzzles because puzzle creation is cheaper (thereby achieving extremely low overhead on write operations) and because they allow us to obtain novel useful functionality such as puzzle degradation (Sect. 3.3), but with the hash function being Argon2 which is designed to be resistant to GPU and ASIC optimization. Picking appropriate difficulty levels for puzzles is something an adopter must do as a function of the tolerable cost for honest users to access data, the perceived risk of a data breach, and the anticipated value of the information to an adversary. We do not aim to study such economic calculations exhaustively, but we provide one worked example in Sect. 4.4 and Table 3.

2.3 Threat Model

ArchiveSafe is a software system with one target asset, the data files. The security goal for the target asset is confidentiality. As shown in Fig. 1, information flows from the user application through the ArchiveSafe driver to the underlying storage during writes, and in the reverse direction during reads.

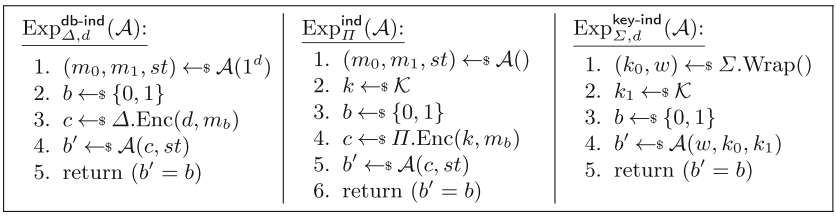

Fig. 2. Security experiments for (*left*) indistinguishability of difficulty-based keyless encryption scheme Δ at difficulty level d; (*centre*) one-time indistinguishability of symmetric encryption scheme Π; and (*right*) indistinguishability of difficulty-based keyless key wrap scheme Σ with keyspace $\mathcal{K}$ and difficulty level d.

An adversary could access the system either via the same mechanism as an honest user application (i.e., mediated by the ArchiveSafe driver), or may have direct access to the underlying storage. We aim to achieve confidentiality against a strong adversary that can bypass the ArchiveSafe driver during read operations (e.g., because they are untrusted server administrators, or because they have compromised the kernel using privilege escalation), or who can directly read from the underlying storage (e.g., an untrusted cloud storage provider, or physical theft of a hard drive). We do not consider in our threat model an adversary who undermines the write operation to intercept data during a write operation or who prevents the ArchiveSafe technique from being applied when saving files. We assume operations by honest parties are performed on a trusted and uncompromised system that faithfully deletes keys from memory once an operation is completed.

3 Difficulty-Based Keyless Encryption

A difficulty-based key encryption scheme is similar to a symmetric encryption scheme, except that no secret key is kept for use between the encryption and decryption algorithm.

Definition 1 (Difficulty-Based Keyless Encryption). *A* difficulty-based keyless encryption (DBKE) scheme Δ *for a message space* $\mathcal{M}$ *with maximum difficulty* $D \in \mathbb{N}$ *consists of two algorithms:*

- $\Delta.\mathrm{Enc}(d, m) \xrightarrow{\$} c$*: A (probabilistic) encryption algorithm that takes as input difficulty level* $d \leq D$ *and message* m *and outputs ciphertext* c*.*
- $\Delta.\mathrm{Dec}(c) \rightarrow m'$*: A deterministic decryption algorithm that takes as input ciphertext* c *and outputs message* m' *or an error* $\perp \notin \mathcal{M}$*.*

A DBKE Δ is *correct* if, for all messages $m \in \mathcal{M}$ and all difficulty levels $d \leq D$, we have that $\Pr[\Delta.\mathrm{Dec}(\Delta.\mathrm{Enc}(d, m)) = m] = 1$, where the probability is taken over the randomness of $\Delta.\mathrm{Enc}$.

The desired security property for a DBKE is semantic security in the form of ciphertext indistinguishability. Since there is no persistent secret key, there is no need to consider security notions incorporating chosen plaintext or chosen ciphertext attacks: each plaintext is protected by independent randomness. The security experiment $\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A})$ for an adversary $\mathcal{A}$ trying to break indistinguishability of DBKE scheme Δ at difficulty level d is shown in Fig. 2. We define the advantage of such an adversary in the security experiment as $\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A}) = \left|2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A}) \Rightarrow \mathtt{true}\right] - 1\right|$. Useful forms of $\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A})$ will relate the amount of work done by the adversary, the difficulty level, and the adversary's success probability.

3.1 Generic Construction of DBKE

Our main construction of DBKE, as shown in Fig. 3, generically combines a traditional symmetric encryption scheme with a "keyless key wrap", which is difficulty-based form of key wrapping: there is no "master key" wrapping the session key, instead the session key is recovered via some difficulty-based operation. In this subsection we present the generic building blocks we use to construct DBKE. In Sect. 3.2 we show how to instantiate the keyless key wrap.

Definition 2 (Symmetric encryption scheme). *A* symmetric encryption scheme Π *with secret key space* $\mathcal{K} = \{0,1\}^\lambda$ *and message space* $\mathcal{M}$ *consists of two algorithms:*

- $\Pi.\mathrm{Enc}(k, m) \xrightarrow{\$} c$*: A (probabilistic) encryption algorithm that takes as input key* $k \in \mathcal{K}$ *and message* $m \in \mathcal{M}$ *and outputs ciphertext* c*.*
- $\Pi.\mathrm{Dec}(k, c) \to m'$*: A deterministic decryption algorithm that takes as input key* $k \in \mathcal{K}$ *and ciphertext* c *and outputs message* $m' \in \mathcal{M}$ *or an error* $\perp \notin \mathcal{M}$*.*

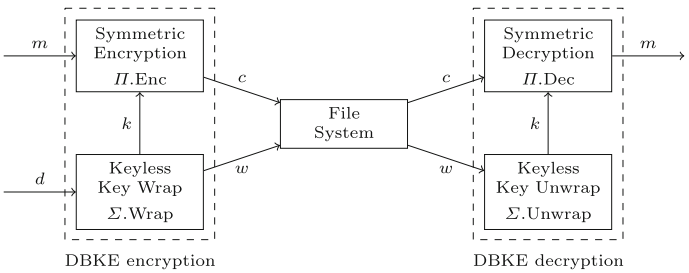

Fig. 3. Architectural diagram for generic construction of a difficulty-based keyless encryption scheme $\Gamma = \Gamma[\Pi, \Sigma]$ from a difficulty-based keyless key wrap scheme Σ and a symmetric encryption scheme Π.

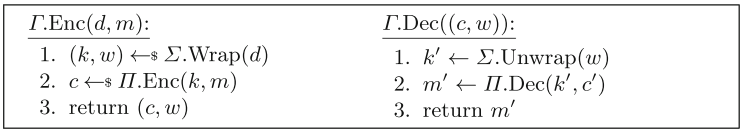

Fig. 4. Generic construction of a difficulty-based keyless encryption scheme $\Gamma = \Gamma[\Pi, \Sigma]$ from a difficulty-based keyless key wrap scheme Σ and a symmetric encryption scheme Π.

Correctness is defined in the obvious way. For our purposes, a sufficient security property will be one-time semantic security, in the form of ciphertext indistinguishability. As above, we will not need to consider security notions incorporating chosen plaintext or chosen ciphertext attacks, since our system will use a key only once. The security experiment $\mathrm{Exp}^{\mathsf{ind}}_{\Pi}(\mathcal{A})$ for an adversary $\mathcal{A}$ trying to break indistinguishability of symmetric encryption scheme Π is shown in Fig. 2. We define the advantage of such an adversary in the security experiment as $\mathrm{Adv}^{\mathsf{ind}}_{\Pi}(\mathcal{A}) = \left|2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{ind}}_{\Pi}(\mathcal{A}) \Rightarrow \mathtt{true}\right] - 1\right|$.

The second building block for our construction is a keyless key wrap scheme.

Definition 3 (Keyless key wrap scheme). *A keyless key wrap scheme Σ for a key space $\mathcal{K} = \{0,1\}^\lambda$ with maximum difficulty level $D \in \mathbb{N}$ consists of two algorithms:*

- $\Sigma.\mathrm{Wrap}(d) \xrightarrow{\$} (k, w)$*: A (probabilistic) key wrapping algorithm that takes as input difficulty level $d \leq D$ and outputs key $k \in \mathcal{K}$ and wrapped key w.*
- $\Sigma.\mathrm{Unwrap}(w) \rightarrow k'$*: A deterministic key unwrapping algorithm that takes as input wrapped key w and outputs key $k \in \mathcal{K}$ or an error $\perp \notin \mathcal{K}$.*

Correctness, again, is defined in the natural way: applying Unwrap to a wrapped key w output by Wrap should yield, with certainty, the same key k as originally output by Wrap.

The desirable security property for a keyless key wrap scheme will be indistinguishability of keys: given the wrapped key, can the adversary learn anything about the key within it? The key indistinguishability security experiment $\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}$ for an adversary $\mathcal{A}$ trying to break key indistinguishability of a keyless key wrap scheme at difficulty level d is shown in Fig. 2. We define the advantage of such an adversary in the security experiment as $\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A}) = \left|2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A}) \Rightarrow \mathtt{true}\right] - 1\right|$. As with DBKE security, useful forms of $\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A})$ will relate the amount of work done by the adversary, the difficulty level, and the adversary's success probability.

As noted above, we generically construct a difficulty-based keyless encryption scheme by combining a traditional symmetric encryption scheme with a keyless key wrap scheme, as outlined in Fig. 3. Let Π be a symmetric encryption scheme with key space $\mathcal{K} = \{0,1\}^\lambda$, and let Σ be a keyless key wrap scheme for key space $\mathcal{K}$ with maximum difficulty level D. Construct the difficulty-based keyless

encryption scheme $\Gamma[\Pi, \Sigma]$ from Π and Σ as outlined in Fig. 3 and specified in Fig. 4.

Our DBKE scheme Γ is secure, in the sense of Fig. 2, under the assumption that the building blocks are secure. The proof follows from a straightforward game-hopping argument; details are omitted due to space constraints and appear in the full version.[6]

Theorem 1. *If Σ is a key-indistinguishable difficulty-based keyless key wrap scheme, and Π is a one-time indistinguishable symmetric encryption scheme, then $\Gamma = \Gamma[\Pi, \Sigma]$ is a secure difficulty-based keyless encryption scheme. More precisely, let $d \leq D$ and let $\mathcal{A}$ be a probabilistic algorithm. Then there exists algorithms $\mathcal{B}_1$ and $\mathcal{B}_2$, such that* $\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Gamma,d}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) + \mathrm{Adv}^{\mathsf{ind}}_{\Pi}(\mathcal{B}_2^{\mathcal{A}})$. *Moreover, $\mathcal{B}_1^{\mathcal{A}}$ and $\mathcal{B}_2^{\mathcal{A}}$ have about the same runtime as $\mathcal{A}$.*

3.2 Hash-Based Construction of Difficulty-Based Keyless Key Wrap

We now show how to construct our difficulty-based keyless key wrap using a hash-based puzzle. The idea is simple: a random seed r is chosen, and the key and a checksum of the seed are derived from the seed using hash functions. The wrapped key consists of the checksum of the seed and the seed with *some of its bits removed*; the number of bits removed corresponds to the difficulty of the puzzle. This is similar to the sub-puzzle construction of Juels and Brainard [12] or partial inversion proof of work by Jakobsson and Juels [11]. Such a puzzle is solved by trying all possibilities for the missing bits, in any order and with or without using parallelization.

In particular, let $\lambda \in \mathbb{N}$, and let $H_1, H_2 : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be independent hash functions. Define keyless key wrap scheme $P = P[H_1, H_2]$ as in Fig. 5 (left). The notation $r[\lambda - d : \lambda]$ on line 2 of P.Wrap denotes taking the substring of r corresponding to indices $\lambda - d$ up to λ, removing the first d bits of r.

P.Wrap(d):
1. $r \leftarrow\!\!\$\ \{0,1\}^\lambda$
2. $\overline{r} \leftarrow r[\lambda - d : \lambda]$
3. $h \leftarrow H_1(r)$
4. $k \leftarrow H_2(r)$
5. $w \leftarrow (h, \overline{r})$
6. return (k, w)

P.Unwrap($w = (h, \overline{r})$):
1. $d \leftarrow \lambda - |\overline{r}|$
2. for $i \in \{0,1\}^d$:
3. $\quad r' \leftarrow i \| \overline{r}$
4. $\quad h' \leftarrow H_1(r')$
5. $\quad$ if $h' = h$:
6. $\quad\quad k \leftarrow H_2(r')$
7. $\quad\quad$ return k
8. return $\perp$

$\Gamma[\Pi, P]$.Degrade($\hat{c}, d'$):
1. parse $\hat{c}$ as $(c, w = (h, \overline{r}))$
2. $d \leftarrow \lambda - |\overline{r}|$
3. abort if $d' < d$
4. $\overline{r}' \leftarrow \overline{r}[d' - d : |\overline{r}|]$
5. $w' \leftarrow (h, \overline{r}')$
6. return (c, w')

Fig. 5. *Left:* Construction of a hash-based keyless key wrap scheme $P = P[H_1, H_2]$ from hash functions H_1, H_2. *Right:* Degradation algorithm for DBKE $\Gamma = \Gamma[\Pi, P]$ constructed using generic construction Γ of Fig. 4 using hash-based keyless key wrap scheme P of left.

[6] https://arxiv.org/abs/2009.00086.

The following theorem shows the key indistinguishability security of our hash-based keyless key wrap scheme P in the random oracle model. The proof consists of a query counting argument in the random oracle model; details are omitted due to space constraints and appear in the full version.

Theorem 2. *Let H_1 and H_2 be random oracles. Let $\lambda \in \mathbb{N}$ and let $d \leq \lambda$. Let $P = P[H_1, H_2]$ be the keyless key wrap scheme from Fig. 5 (left). Let $\mathcal{A}$ be an adversary in key indistinguishability experiment against P which makes q_1 and q_2 distinct queries to its H_1 and H_2 random oracles, respectively. Then* $\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{P,d}(\mathcal{A}) \leq \frac{q_1}{2^{d-1}} + \frac{2}{2^d - q_1}$.

Puzzle Granularity. The partial pre-image puzzle construction used in Fig. 5 does not allow for fine-grained control of difficulty: removing each additional bit increases the expected computational cost by a factor of 2. Higher granularity can be achieved similar to how the puzzle difficulty in Bitcoin is set, by giving a hint that narrows the range of data from 2^d to some smaller subset.

3.3 Puzzle Degradation

We now introduce an additional feature of difficulty-based keyless encryption that emerges naturally from our hash-based keyless key wrap construction: *puzzle degradation*. Abstractly, puzzle degradation is a process that takes a DBKE ciphertext and increases the difficulty of decrypting it, preferably without needing to decrypt and then re-encrypt at a higher difficulty level.

In the context of the ArchiveSafe long-term archiving system, this may be used to gradually increase the difficulty of files that have not been accessed for a certain period of time. For example, a monthly maintenance process could apply degradation to stored files to gradually increase the cost (to both an attacker and an honest party) of accessing increasingly older files.

The DBKE system Δ from Definition 1 is augmented with the algorithm:

- $\Delta.\mathrm{Degrade}(c, d') \xrightarrow{\$} c'$: A (possibly probabilistic) algorithm that takes as input ciphertext c and target difficulty level $d' \leq D$, and outputs updated ciphertext c'.

Correctness is extended to demand that a ciphertext output by $\Delta.\mathrm{Enc}$ then degraded any number of times is still correctly decrypted by $\Delta.\mathrm{Dec}$ (although decryption may take longer).

Security with the degraded algorithm included should mean, intuitively, that a ciphertext degraded any number of times can be decrypted only using the required amount of work at the new difficulty level.

We capture both correctness and security of degradation formally by demanding that, for all $d \leq d' \leq D$ and all $m \in \mathcal{M}$, we have that $\Delta.\mathrm{Enc}(d', m) \equiv \Delta.\mathrm{Degrade}(d', \Delta.\mathrm{Enc}(d, m))$; in other words: the distribution of ciphertexts produced by encrypting at difficulty d' is identical to the distribution of ciphertexts produced by encrypting at difficulty d and then degrading to difficulty d'.

We can achieve degradation in DBKE $\Gamma = \Gamma[\Pi, P]$ constructed from our hash-based keyless key wrap P in a trivial way: by removing $(d' - d)$ more bits from the puzzle hint $\overline{r}$. This clearly requires no decryption and re-encryption, only a constant-time edit to the metadata stored containing the wrapped key. The procedure Γ.Degrade is stated in Fig. 5 (*right*). Degraded ciphertexts are identically distributed to ciphertexts freshly generated at the target difficulty level, as removing additional bits of the partial seed $\overline{r}$ is associative. An adversary who possess a copy of the metadata from an earlier version of the archive prior to degradation can solve puzzles and decrypt at the earlier, non-degraded difficulty level.

3.4 Additional Considerations

Outsourcing Puzzle Solving. The generic DBKE construction Γ of Fig. 4 allows the key unwrapping and ciphertext decryption to be done separately, so the expensive key unwrapping could be outsourced to a cloud server. In the example of the hash-based keyless key wrap scheme P of Fig. 5, the user could give the wrapped key $w = (h, \overline{r})$ to the cloud server who unwraps and returns the key k, which the user then locally uses to decrypt the ciphertext c.

This does mean that the cloud server learns the encryption key k. However, this can be avoided with the following adaption to the construction P of Fig. 5. During wrapping, the algorithm generates an additional *salt* value $s \leftarrow\!\!\$\ \{0,1\}^\lambda$ and computes $k \leftarrow H_2(r \| s)$; s is stored in the wrapped key w. When outsourcing the unwrapping to the cloud server, the user only sends h and $\overline{r}$, but not s. The cloud server is still able to use the checksum h with the partial seed $\overline{r}$ to recover the full seed r, but lacks the salt s and thus the cloud server alone cannot compute the decryption key k. Theorem 2 still applies to this adaptation.

Combining Keyless and Keyed Encryption. As previously mentioned, our keyless encryption approach can (and should) be used in conjunction with traditional keyed encryption mechanisms using a different set of keys. Traditional keyed encryption gives honest parties a (conjecturally exponential) work factor advantage over adversaries if keys remain uncompromised, while keyless encryption slows adversaries if the traditional encryption keys are compromised. The two schemes can be layered in one of two ways: first applying keyless encryption DBKE and encrypting the result using keyed symmetric encryption Sym (i.e., $c \leftarrow \mathrm{Sym.Enc}(k, \mathrm{DBKE.Enc}(d, m))$) or in the order, with keyless encryption on the outer layer (i.e., $c \leftarrow \mathrm{DBKE.Enc}(d, \mathrm{Sym.Enc}(k, m))$). Either approach yields robust confidentiality, but we recommend the latter method as it facilitates the puzzle degradation process described in Sect. 3.3.

4 Evaluation

We evaluate ArchiveSafe by measuring its performance against other systems through real life experiment. The goals of the experiment are to: (1) measure

the overhead ArchiveSafe introduces on adversaries and honest users, and (2) verify that puzzle solving difficulty scale according to the theoretical system design.

4.1 Prototype Implementation

To run the evaluation experiment, we implemented a prototype of ArchiveSafe.[7] In terms of instantiating the difficulty-based keyless encryption using the generic construction from Sect. 3.1, our proof-of-concept uses AES-128 in CBC mode for the symmetric encryption scheme. The hash functions H_1 and H_2 in the hash-based keyless key wrap scheme are both instantiated with Argon2id [5] with a prefix byte acting as a domain separator between H_1 and H_2, with the following parameters: parallelism level: 8; memory: 102,400 KiB; iterations: 2; output length: 128 bits. We did not parallelize puzzle solving in Unwrap to avoid locking other system operations, but it is easily parallelized.

The ArchiveSafe prototype is implemented as a Linux Filesystem in Userspace (FUSE) using a Python toolkit[8] to simplify implementation. Our Python FUSE driver relies on the OpenSSL library for encryption and decryption, and Ubuntu's `argon2` package. In a real deployment in the context of a filesystem, ArchiveSafe would be implemented as a kernel module, likely written in C, for improved performance and reliability.

Our prototype has a tuneable difficulty level, which we label in this section as D1, D2, D3, etc. Difficulty Dx corresponds to hash-based keyless key wrap scheme P of Fig. 5 with difficulty parameter $d = 4x$; in other words, D1 removes 4 bits of the seed, D2 removes 8 bits of the seed, etc. We chose a 4-bit step between difficulty levels to focus on how system behaviour scales across difficulty levels; finer gradations could be chosen by users.

4.2 Experimental Setup

The experiment measures ArchiveSafe's performance at three difficulty levels (D1, D2, D3) compared to an unencrypted file system (denoted UN) and Linux's built-in folder encryption using eCryptfs[9] (denoted FE) and disk encryption (denoted DE) on read and write tasks at different file sizes. When running the ArchiveSafe experiments, the ArchiveSafe FUSE driver was writing its files to an unencrypted file system.

Measurements. For each storage system being evaluated, we measure *read* and *write* times for files of sizes 1 KB, 100 KB, 1 MB, 10 MB, and 100 MB. Performance is measured at the application level, from the time the file is opened until the time the read/write operation is completed. For folder and disk encryption,

[7] Our prototype is available at https://github.com/moesabry/ArchiveSafe.
[8] https://github.com/skorokithakis/python-fuse-sample.
[9] https://www.ecryptfs.org/.

this includes the filesystem's encryption operations. For ArchiveSafe, we instrumented the driver to record the total time as well times for different sub-tasks (encryption, puzzle solving, decryption, file system I/O).

Test Environment. Measurements were performed on a single-user Linux machine with no other processes running. The computer was a MacBook Pro running Ubuntu Linux 18.04 LTS with an 4-core Intel Core i7-4770HQ processor with base frequency 2.2 GHz, bursting to 3.4 GHz. The computer had 16 GiB of RAM. The hard drive was a 256 GiB solid state drive with 512-byte logical sectors and 4096-byte physical sectors. The disk encryption was done using Linux Unified Key Setup system version 2.0, and folder encryption was done using the Enterprise Cryptographic Filesystem (eCryptfs) version 5.3.

Execution. For each storage system and file size, we performed many repetitions of the following tasks. A file was created with randomly generated alphanumeric characters using a non-cryptographic random number generator. Read and write operations were measured as indicated above. For file sizes of 1 KB, 100 KB, 1 MB, and 10 MB, we collected data for 1000 writes and reads; for 100 MB files, we ran 200 writes and reads, due to extensive time of operations at this size.

4.3 Results

Table 1 shows average read and write times for the file systems under consideration at different file sizes. Since read operations in the ArchiveSafe system become increasingly expensive with difficulty, we show in Table 2 the average time of sub-tasks of ArchiveSafe read operations at different file sizes and difficulties: the puzzle solving time (which should scale with puzzle difficulty), the system file read time plus decryption time (which should scale with file size), and the overhead from other file system driver operations (which includes puzzle read and system file open times). As the partial pre-image puzzle used in ArchiveSafe leads to highly variable solving times, Fig. 6 shows the average time and standard deviation for puzzle solving at difficulties D1, D2, and D3.

Table 1. Average read and write times in milliseconds

File system	Read					Write				
	1 KB	100 KB	1 MB	10 MB	100 MB	1 KB	100 KB	1 MB	10 MB	100 MB
Unencrypted (UN)	0.526	0.550	1.70	10.1	110	0.07	0.25	0.85	6.76	97.82
Disk Encryption (DE)	0.737	0.924	3.15	10.5	160	0.08	0.25	0.83	6.63	97.97
Folder Encryption (FE)	0.737	0.961	3.42	10.9	190	0.12	0.50	3.31	29.07	319.88
ArchiveSafe D1	630	630	630	650	860	141.05	141.67	146.09	221.73	848.30
ArchiveSafe D2	7070	7080	7310	7180	7290	141.25	141.43	145.08	223.50	847.02
ArchiveSafe D3	112140	111760	107390	114530	107630	141.01	140.98	145.74	222.40	846.06

Table 2. Read sub-tasks average times in milliseconds

Diff.		1 KB	100 KB	1 MB	10 MB	100 MB
D1	Puzzle Solve	510	510	510	510	500
	Decryption	5.42	5.71	7.25	20	150
	Other	0.387	0.373	0.378	0.384	0.363
D2	Puzzle Solve	6960	6980	7210	7050	6930
	Decryption	5.58	6.12	7.89	20	140
	Other	0.357	0.373	0.376	0.374	0.335
D3	Puzzle Solve	112040	111730	107280	114410	107270
	Decryption	5.56	5.94	7.96	20	140
	Other	1.075	1.216	0.971	1.195	1.045

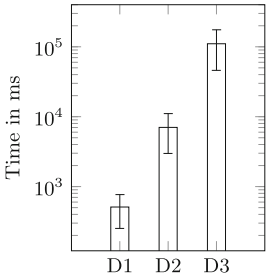

Fig. 6. Puzzle solving time in milliseconds (average, standard deviation)

4.4 Discussion

The results show consistent behaviour across different file sizes. The larger files consumed more time in decrypting and reading. We also observed that the time consumed is roughly the same for smaller file sizes (1 KB and 100 KB) where operation cost is dominated by overhead.

As expected, the read speeds decrease with the difficulty level because the system must solve the puzzle before reading the file and the puzzle solving effort scales with the difficulty level. As per Table 2, puzzle solve times on average scale by a factor of 13.6–14.1× between D1 and D2 and a factor of 14.9–16.2× between D2 and D3, roughly in line with the theoretical scaling factor of 16×.

Evaluating the overhead added by ArchiveSafe for write operations, we see in Table 1 that ArchiveSafe incurs a baseline overhead related to setting up the puzzle (which involves 2 Argon2 calls), then scales with the file size due to the cost of AES encryption and writing. Note that ArchiveSafe uses a different encryption library (user-space calls to OpenSSL) compared with disk and file encryption (kernel encryption via dm-crypt), so symmetric encryption/decryption performance is not directly comparable, but we see similar scaling.

The short summary of performance is that ArchiveSafe adds a 140–520 ms overhead when writing a file, and a customizable overhead when reading a file, ranging from 510 ms at difficulty D1, 7 s at D2, or 110 s at D3. But recall that adding computational overhead at read time is exactly the purpose of ArchiveSafe! What an acceptable difficulty level—and hence acceptable computational overhead at read time for honest users—is a policy choice by the system administrator. As noted earlier, choosing the difficulty level depends on the tolerable cost for honest users to access data, the perceived risk of a data breach, and the anticipated value of the information to an adversary, and is a calculation that must be left to the adopter. Note that honest users need not solely rely on sequential operations on their own computer: as described in Sect. 3.4 an ArchiveSafe installation could be configured so that honest users offload their puzzle solving tasks to private or commercial clouds which are spun

Table 3. Dollar cost and computation time required to unlock ArchiveSafe files

	D3	D4	D5	D6
Honest user decrypting 1 file				
Local machine, threaded 4 cores, 2.2 GHz	0.5 min	7.3 min	2 h	31 h
Cloud server `c5.metal`, spot pricing	$\ll$\$0.01	$<$\$0.01	\$0.05	\$0.73
Adversary decrypting 1 million files				
Cloud server `c5.metal`	8 days	130 days	5.7 yrs	91.4 yrs
Cloud server `c5.metal`, spot pricing	\$178	\$2,852	\$45,648	\$730,364

up on demand with large amounts of parallelization to reduce the wall clock time before they can access a file.

Table 3 shows examples of costs at higher difficulty levels. To provide further interpretation to these costs, we look not only at the computation time required for an honest user on our test platform to decrypt a file, but also at the real-world cost for an adversary, based on the cost of renting computation time on Amazon Web Services (AWS) Elastic Cloud Compute (EC2) platform. EC2 has many machine types available; Argon2 is designed to not be substantially accelerated by more sophisticated architectures, GPUs, or ASICs. As such we choose for our pricing example an EC2 instance that minimizes cost per core-GHz-hour; the `c5.metal` EC2 instance type has 96 Intel Xeon cores running at 3.6 GHz at a cost of USD\$0.9122 per hour using Amazon's cheapest spot pricing model.[10]

We can see, for example, that at difficulty D5, an honest user can unlock an archived file with about 2 h of work on a local machine, or about 3 min of `c5.metal` rental costing 4.5 cents at spot pricing (20 cents on-demand pricing). However, an adversary trying to decrypt 1 million such files from a data breach would need 5.7 years of `c5.metal` rental at a spot pricing cost of USD\$45,648.

5 Conclusion

ArchiveSafe, using difficulty-based keyless encryption, can add defense-in-depth to confidentiality of archived data and change the economics of mass leakage attacks via data breaches. We expect that most uses of ArchiveSafe would be in addition to, not as a replacement for, traditional keyed encryption; full cryptographic security would be achieved if encryption keys are properly managed and kept safe, but ArchiveSafe provides a residual level of protection if traditional encryption keys are also breached. This means the key management service is no longer a single point of failure.

One target application is IT systems which retain large amounts of archival data, most of which will be rarely or perhaps never again accessed by legitimate users. Although honest users have no advantage in difficulty-based decryp-

[10] https://aws.amazon.com/ec2/instance-types/, https://aws.amazon.com/ec2/spot/pricing/; prices as of April 23, 2020.

tion compared to an adversary on a file-by-file basis, if their operational goals are different—an honest user decrypting 1 file occasionally, versus an adversary decrypting thousands or millions of files quickly—their costs are different.

Our approach can be applied in a variety of system architectures: local storage and execution (as demonstrated by our prototype), local storage with private or public cloud assistance for puzzle solving, or remote (file server/cloud) storage with local or assisted puzzle solving. Our approach can also apply to different storage paradigms, including file systems, cloud "blob" storage, and databases.

Puzzle difficulty can be set as a system-wide or with higher granularity based individual records' sensitivity. A novel features of our construction is the ability to degrade puzzle difficulty effectively for free, which could be built into periodic maintenance or through a heuristic system based on suspicious activity.

Acknowledgements. This work grew out of earlier discussions on use of puzzles for database encryption with Farhad Moghimifar, Suriadi Suriadi, and Ernest Foo at the Queensland University of Technology. R.S. is supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2016-06062. D.S. is supported by NSERC Discovery grant RGPIN-2016-05146 and NSERC Discovery Accelerator Supplement grant RGPIN-2016-05146.

References

1. Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. ACM Trans. Internet Technol. **5**(2), 299–327 (2005)
2. Aura, T., Nikander, P., Leiwo, J.: DOS-resistant authentication with client puzzles. In: Christianson, B., Malcolm, J.A., Crispo, B., Roe, M. (eds.) Security Protocols 2000. LNCS, vol. 2133, pp. 170–177. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44810-1_22
3. Back, A.: Hashcash: a denial of service counter-measure (2004). http://www.hashcash.org/docs/hashcash.html
4. BBC: Adobe hack: at least 38 million accounts breached (2013). https://www.bbc.com/news/technology-24740873
5. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: new generation of memory-hard functions for password hashing and other applications. In: IEEE EuroS&P (2016)
6. Blaze, M.: A cryptographic file system for UNIX. In: ACM CCS (1993)
7. Cattaneo, G., Catuogno, L., Del Sorbo, A., Persiano, P.: The design and implementation of a transparent cryptographic file system for UNIX. In: USENIX Technical Conference, FREENIX Track (2001)
8. Dean, D., Stubblefield, A.: Using client puzzles to protect TLS. In: USENIX Security Symposium, vol. 42 (2001)
9. Dwork, C., Goldberg, A., Naor, M.: On Memory-bound functions for fighting spam. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 426–444. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_25
10. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-48071-4_10
11. Jakobsson, M., Juels, A.: Proofs of work and bread pudding protocols (extended abstract). In: IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security, pp. 258–272 (1999)

12. Juels, A., Brainard, J.: Client puzzles: a cryptographic countermeasure against connection depletion attacks. In: NDSS (1999)
13. Moghimifar, F.: Securing database using client puzzles. Master's report, Queensland University of Technology, November 2015
14. NBC News: Ameritrade warns 200,000 clients of lost data (2005). http://www.nbcnews.com/id/7561268.XeryPuhKiMo
15. Poddar, R., Boelter, T., Popa, R.A.: Arx: a strongly encrypted database system. IACR Cryptology ePrint Archive (2016)
16. Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 85–100. ACM (2011)
17. Rangasamy, J., Stebila, D., Boyd, C., González-Nieto, J.M., Kuppusamy, L.: Effort-release public-key encryption from cryptographic puzzles. In: ACISP (2012)
18. Reuters: Bank of NY... (2008). https://www.reuters.com/article/bankofnymellon-breach-idUSWNAB863220080828
19. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Technical report, MIT, March 1996. https://people.csail.mit.edu/rivest/pubs/RSW96.pdf
20. Suriadi, S., Stebila, D., Clark, A., Liu, H.: Defending web services against denial of service attacks using client puzzles. In: ICWS (2011)
21. Vargas, L., et al.: Mitigating risk while complying with data retention laws. In: ACM CCS (2018)
22. Waters, B., Juels, A., Halderman, J.A., Felten, E.W.: New client puzzle outsourcing techniques for DoS resistance. In: ACM CCS (2004)
23. Wright, C., Martino, M., Zadok, E.: NCryptfs: a secure and convenient cryptographic file system. In: USENIX Technical Conference, General Track, pp. 197–210 (2003)
24. Zadok, E., Badulescu, I., Shender, A.: Cryptfs: a stackable vnode level encryption file system. Technical report CUCS-021-98, CS Department, Columbia University (1998)