SafetyPin: Encrypted Backups with Human-Memorable Secrets

Emma Dauterman
UC Berkeley

Henry Corrigan-Gibbs
EPFL and MIT CSAIL

David Mazières
Stanford

Abstract. We present the design and implementation of SafetyPin, a system for encrypted mobile-device backups. Like existing cloud-based mobile-backup systems, including those of Apple and Google, SafetyPin requires users to remember only a short PIN and defends against brute-force PIN-guessing attacks using hardware security protections. Unlike today's systems, SafetyPin splits trust over a cluster of hardware security modules (HSMs) in order to provide security guarantees that scale with the number of HSMs. In this way, SafetyPin protects backed-up user data even against an attacker that can adaptively compromise many of the system's constituent HSMs. SafetyPin provides this protection without sacrificing scalability or fault tolerance. Decentralizing trust while respecting the resource limits of today's HSMs requires a synthesis of systems-design principles and cryptographic tools. We evaluate SafetyPin on a cluster of 100 low-cost HSMs and show that a SafetyPin-protected recovery takes 1.01 seconds. To process 1B recoveries a year, we estimate that a SafetyPin deployment would need 3,100 low-cost HSMs.

1 Introduction

Modern mobile phones and tablets back up sensitive data to the cloud. To protect users' privacy, this data must be encrypted under keys that are not available to the cloud provider. Unfortunately, with 3.8 billion smartphone users, it is impractical to expect them all to store, say, a 128-bit AES backup key. Not everyone has a computer, or trustworthy friends who can keep shares of a backup key, or even a safe place to store a backup key on paper. As a result, mobile OSes have fallen back to protecting backups with the least common denominator: device screen-lock PINs. Using PINs is good for security because a user's screen-lock PIN never leaves her device (so the cloud provider never learns it). Using PINs is good for usability because users generally remember them.

Unfortunately, PINs have such low entropy (e.g, six decimal digits) that no feasible amount of key stretching can protect against brute-force PIN-guessing attacks. Instead, modern backup systems—such as those from Apple [47], Google [82], and Signal [55]—rely on hardware-security modules (HSMs) in their data centers to thwart brute-force attacks. Specifically, devices encrypt their backup keys under the public keys of HSMs, but each device includes a hash of its screen-lock PIN as part of the plaintext. HSMs return decrypted plaintext only to clients that can supply this PIN hash. Furthermore, HSMs limit the number of decryption attempts for any given user account. For fault tolerance, a device typically encrypts its backup key to the public keys of five HSMs, allowing any one of the five to recover the backup key.

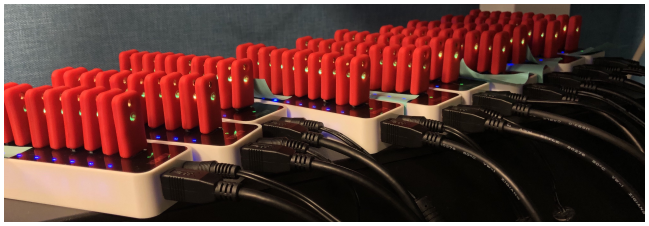


Figure 1: Our cluster of 100 low-cost hardware security modules (SoloKeys [72]) on which we evaluate SafetyPin.

This status quo still falls short of acceptable privacy for two reasons. First, HSMs are not perfect, yet each HSM in these systems is a single point of security failure for millions of users' backup keys. Second, these systems make it difficult for clients to detect security breaches. For instance, if a malicious insider working in a data center physically steals an HSM, then to anyone outside the company it looks like an unremarkable single hardware failure. Alternatively, if an insider successfully guesses someone's PIN, the victim may have no idea her backup was ever compromised.

This paper presents SafetyPin, a PIN-based encrypted-backup system with stronger security properties. The key idea behind SafetyPin is that recovering any user's backed-up data either requires (a) guessing the user's PIN or (b) compromising a very large number of HSMs—e.g., 6% of all HSMs operated by a provider. (The 6% figure here is a tunable system parameter.) Such large-scale attacks would typically need to span multiple data centers, be harder for insiders to pull off undetected against physical devices, cost more, and also likely cause service disruptions visible to end users.

One way to achieve SafetyPin's security goal would be to threshold-encrypt the client's hashed PIN and backup key in such a way that decrypting the client's backup key would require the participation of 6% of all HSMs in the system. Unfortunately, this approach lacks scalability. If each client recovering a backup must interact with 6% of the system's HSMs, adding more HSMs improves security without improving throughput. As the number of HSMs in the system increases, we would like the system's overall throughput to increase in tandem with its security (i.e., the attacker's cost).

To achieve scalability, SafetyPin takes a different approach:

devices threshold-encrypt their backup keys to a small cluster of n HSMs such that decryption requires the participation of most HSMs in the cluster. The cluster size n is independent of the total number of HSMs in the system, and depends on both the fraction of compromised HSMs the system can tolerate and the fraction of HSMs that can fail-stop. (For example, to tolerate the compromise of 6% of HSMs where half of a cluster is allowed to fail-stop, we can set the cluster size $n = 40$.) This design achieves our scalability goal, since each device need only communicate with a small fixed number of HSMs during recovery. This design also achieves our security goal because the cluster of n HSMs that can decrypt a client's backup depends on the client's secret PIN, via a primitive we introduce called *location-hiding encryption*. Hence, even if an attacker compromises 6% of the HSMs in the system as a whole, the chances that the attacker compromises a "useful" set of HSMs—i.e., at least half of the HSMs in the device's chosen cluster—is very small. More precisely, we show that if the total number of HSMs in the system is large enough (a few hundred or more), the probability that an attacker can decrypt a backup via HSM compromise is not much higher than the probability of simply guessing the client's PIN.

In modern backup systems, each HSM only needs to monitor the number of PIN attempts for a small subset of users, but because of our location-hiding encryption primitive, every HSM needs to be able to verify the number of PIN attempts for every user. To maintain this information scalably, the HSMs use a new type of distributed log. Third parties can monitor this log to alert users whenever a backup-recovery attempt is underway. Since a compromised service provider may see which HSMs a mobile device interacts with during recovery (and could compromise those HSMs to recover the users' backed-up data), HSMs revoke their ability to decrypt backups after completing the recovery process. Implementing this revocation requires adapting "puncturable encryption" [38] to storage-limited HSMs. While our prototype is focused on PIN-protected backups, these primitives have potentially broader applicability to problems such as private storage in peer-to-peer systems and cryptocurrency "brain wallets."

We implemented SafetyPin on low-cost SoloKey HSMs [72]. We evaluate the system using a cluster of 100 SoloKeys (Figure 1) and an Android phone (representing the client device). Generating a recovery ciphertext on the client, excluding the time to encrypt the disk image, takes 0.37 seconds. To process 1B recoveries a year, or 123K recoveries per hour, we estimate that we would need 3,100 SoloKeys. In a SafetyPin deployment of 3,100 HSMs, tolerating the compromise of 6% of the HSMs (i.e., 194 HSMs), the client must interact with a cluster of 40 HSMs during recovery. Running our backup-recovery protocol across a cluster of this size takes 1.01 seconds.

Limitations. A limitation of SafetyPin is that the set of HSMs a device uses for recovery can leak information about the user's PIN. In particular, an attacker who controls the data center can learn a salted hash of the user's PIN during recovery. This is unfortunate in the common case that people re-use the same PIN after recovery [23, 42, 32, 70]. We discuss one mitigation in Section 8. Also, while it is possible to detect when PINs can safely be re-used, we have not yet implemented this functionality.

In addition, SafetyPin is more expensive than today's PIN-based backup systems. SafetyPin requires the data center operator to operate a much larger fleet of HSMs (roughly 50 – 100× larger) than the standard HSM-based backup systems require. SafetyPin clients must also download roughly 2MB of keying material per day in a SafetyPin deployment supporting one billion recoveries per year, due to the periodic rotation of large HSM keys. Even so, we expect that the cost of storing and transferring disk images (GBs/user) will dwarf these costs.

2 The setting

Entities. Our encrypted-backup system involves three entities, whose roles we describe here.

Client. Initially, the client holds (1) a username with the service provider, (2) a human-memorable passphrase or PIN, (3) a disk image to be backed up, and (4) the public keys of the service provider's HSMs. Later on, the client should be able to recover her backed-up data using only her username, her PIN, and access to the other components of the backup system.

In SafetyPin, as in today's PIN-based backup systems, security depends on the client having access to the HSMs' true public keys: If a malicious service provider can swap out the HSMs' true public keys for its own public keys without detection, the service provider can immediately break security. Using a distributed log (Section 6) can ensure that all clients see a common set of HSM public keys, to prevent targeted attacks. Hardware-attestation techniques, as used in the FIDO [67] and SGX [44] specs, can provide another defense.

We also assume the provider has traditional account authentication (e.g., Gmail passwords) to prevent random third parties from consuming PIN guesses, but we omit this from the discussion for simplicity.

Service provider. The service provider offers the encrypted-backup service to a pool of clients and it maintains the data centers in which the backup system runs. For example, the service provider could be a mobile-phone vendor, such as Apple or Google. The service provider's data centers contain the network infrastructure that connects the HSMs. They also contain large amounts of (potentially untrustworthy) storage and computing resources. Our security properties will hold against a service provider that becomes compromised at any point after the system is set up.

Hardware security modules (HSMs). The service provider's data centers contain thousands of hardware security modules. An HSM is a tamper-resistant computing device meant for storing cryptographic secrets. HSMs have fully programmable processors but are typically resource-poor (see Table 2). It is possible to lock an HSM's firmware before deployment,

Device	Price	g^x/sec.	Storage	FIPS
SoloKey [72]	$20	8	256 KB⋆	
YubiHSM 2 [84]	$650	14	126 KB	
SafeNet A700 [68]	$18,468	2,000	2,048 KB	✓
Intel i7-8569U (CPU)	$431	22,338	n/a	

Table 2: Hardware security modules offer physical security protections but are computationally weak compared to a standard CPU.

The g^x/sec is NIST P256 elliptic-curve point-multiplications per second. "FIPS?" refers to whether the device meets the FIPS 140-2 standard for HSMs. (⋆ The 256 KB storage on the SoloKey is shared between code and data.)

which makes remote compromise and key-extraction attacks more difficult. Each HSM has a public key and stores the corresponding secret key in its secure memory.

The attack scenario. The service provider (Apple, Google, etc.) spends vast amounts of money acquiring a large user base for products that store user data in the cloud. The provider risks reputational damage and journalistic scrutiny if it cannot ensure the durability and confidentiality of user data.

A service provider can deploy SafetyPin as a way to build trust among its user base and to protect its own infrastructure against future compromise. By enlisting third-party organizations to monitor the SafetyPin deployment's public distributed log, the provider can build further public trust in the system.

At some point after the provider deploys SafetyPin, a powerful attacker wishes to steal user data. The attacker may have malicious insiders working for the provider. It may physically compromise data centers to steal HSMs. It may intercept shipments to tamper with some of the HSMs on their way to the data center. The attacker could also be a state actor employing legal pressure to gain access to data centers. Nonetheless, the attacker is sensitive to both the cost of attacks and the risk of public exposure.

Both the attack cost and risk of exposure increase with the number of HSMs the attacker must compromise. For instance, while a malicious insider working at a data center may be able to abscond with a single HSM—passing the missing device off as a hardware failure—removing 100 HSMs is a much riskier proposition. A state actor who can order the provider to hand over HSMs may be dissuaded if doing so will attract press coverage either by making non-targeted clients' data unrecoverable or creating a damning public audit trail.

The attacker may compromise clients as well as the provider. For instance, the attacker may have a good guess at a target user's PIN, perhaps because of CCTV footage showing the user unlocking a mobile device. While SafetyPin cannot prevent the attacker from gaining access to the data with the correct PIN, the risk will be higher to the attacker if stolen PINs cannot be used without exposing the attack in SafetyPin's public distributed log.

Notation. The set $\mathbb{Z}^{>0}$ refers to the set of natural numbers $\{1,2,3,\dots\}$. For a positive integer n, we let $[n] = \{1,\dots,n\}$ and we use $\perp$ to denote a failure symbol. For strings a and b, we write their concatenation as $a\|b$. Throughout, we use λ to denote the security parameter, and we typically take $\lambda = 128$ (i.e., for 128-bit security).

3 System goals

SafetyPin implements an encrypted-backup functionality, which consists of two routines:

- the *backup* algorithm, which the client uses to produce its encrypted backup, and
- the *recovery* protocol, in which the client uses HSMs to recover the backup plaintext from ciphertext.

We define these protocols with respect to a number of HSMs $N \in \mathbb{Z}^{>0}$ and a finite PIN space $\mathcal{P} \subseteq \{0,1\}^*$. For convenience, we define the *master public key* mpk for a data center to be all N HSMs' public keys: $\mathsf{mpk} = (\mathsf{pk}_1,\dots,\mathsf{pk}_N)$. The syntax of an encrypted-backup system is then as follows:

$\mathsf{Backup}(\mathsf{mpk},\mathsf{user},\mathsf{pin},\mathsf{msg}) \to \mathsf{ct}$. Given the master public key mpk, a client username user, the client's PIN $\mathsf{pin} \in \mathcal{P}$, and a message $\mathsf{msg} \in \{0,1\}^*$ to be backed up, output a recovery ciphertext ct. This routine runs on the client and requires no interaction with HSMs. The client uploads the resulting ciphertext ct to the service provider.

$\mathsf{Recover}^{\mathcal{S},\mathcal{H}_1,\dots,\mathcal{H}_N}(\mathsf{mpk},\mathsf{user},\mathsf{pin},\mathsf{ct}) \to \mathsf{msg}$ or $\perp$. The client initiates the recovery routine, which takes as input the master public key mpk, a client username user, a PIN $\mathsf{pin} \in \mathcal{P}$, and a recovery ciphertext ct.

During the execution of $\mathsf{Recover}$, the client interacts with the service provider $\mathcal{S}$ and a subset of the HSMs $\mathcal{H}_1,\dots,\mathcal{H}_N$. Each HSM $\mathcal{H}_i$ holds the master public key mpk, and its secret decryption key sk_i. During recovery, the data center provides the client's username user to each HSM.

The recovery routine outputs a backed-up message $\mathsf{msg} \in \{0,1\}^*$ or a failure symbol $\perp$.

We now describe the security properties that such a system should satisfy. We work in an asynchronous network model; we use standard cryptographic primitives to set up authenticated and encrypted channels between the client, service provider, and HSMs.

Property 1: Security. If the client obtains the HSMs' true public keys, then even an attacker that:

- controls the service provider (in particular, is an active network attacker inside the data centers and has control of the service provider's servers and storage),
- compromises an f_{secret} (e.g., $f_{\mathsf{secret}} = \frac{1}{16}$) fraction of HSMs in the data center *before* the client begins the recovery process, and
- compromises all of the HSMs in the data center *after* the recovery protocol completes,

still should learn nothing about any honest client's encrypted message (in a semantic-security sense [35]) beyond what it

can learn by guessing that client's PIN.

Discussion: The adversary can inspect all clients' recovery ciphertexts and then choose to compromise a large set of HSMs that depends on these ciphertexts. Such attacks are relevant when, for example, a state actor with the power to compromise many HSMs targets the backed-up data of a specific set of users.

Two important caveats are: (1) SafetyPin does not protect against an attacker compromising HSMs while recovery is in progress (see Figure 4) and (2) as implemented, SafetyPin does not protect the PIN: an adversary that observes which HSMs the client contacts during recovery may learn a salted hash of the PIN after recovery completes. Section 6.3 discusses how to detect and mitigate this leakage by protecting the salt.

Property 2: Scalability. The recovery protocol should require the client to interact with a constant number of HSMs, independent of the number of HSMs in the data center. (This constant may depend on the security parameter and on the fraction of HSMs whose compromise the system can tolerate.) Hence, providers can deploy additional HSMs to scale capacity. Concretely, when we configure the system to tolerate the compromise of $f_{\mathsf{secret}} = \frac{1}{16}$ of the data center's HSMs, our protocol requires the client to communicate with 40 HSMs during recovery.

Property 3: Fault tolerance. Every client should be able to recover her encrypted message even if a constant fraction f_{live} (e.g. $f_{\mathsf{live}} = \frac{1}{64}$) of the HSMs in the data center fail-stop.

Setting parameters. For the remainder of this paper, we set the fraction of compromised HSMs that the system can tolerate to $f_{\mathsf{secret}} = \frac{1}{16}$ and the fraction of HSMs that can fail while still allowing the client to recover her backup to $f_{\mathsf{live}} = \frac{1}{64}$. This choice is reasonable because large companies have more than 16 data centers, while smaller companies can collaborate on a shared deployment with 16 physical security perimeters. By adjusting the other parameters, it is possible to achieve any $0 < f_{\mathsf{secret}} < 1$ or $0 < f_{\mathsf{live}} < 1$. (In Section 9.2, we discuss how the choice of these values affects other system parameters.)

4 Architecture overview

We now describe our encrypted-backup protocol (Figure 3) and explain how it satisfies the design goals of Section 3. We will discuss possible extensions and deployment considerations in Section 8.

4.1 The back-up process

The client begins the back-up process holding

- the public keys of all HSMs in the data center,
- its secret PIN, and
- a disk image to be backed up (the "message").

To back up its disk image, the client samples a subset of n HSMs out of the N total HSMs in the data center where $n \ll N$. The client chooses this subset by hashing (a) public information: the service name, its username, and a public salt the client chooses at random, and (b) its secret PIN. The client then encrypts its message with a random AES encryption key, and then splits this AES key into n threshold shares using Shamir secret sharing [69], such that any threshold t of the shares suffice to recover the AES key. The client prepends each share with the client's username to ensure that the ciphertexts are bound to the client's username. The client then encrypts one share to the public key of each HSM in its chosen subset.

The client's recovery ciphertext then consists of: its public salt, the AES-encrypted message, the n encrypted shares of the AES key, and a configuration-epoch number that the service provider can use to identify the set of HSMs that were in service at the time the client created its backup. The client computes the ciphertext locally and uploads it to the backup service provider, with no HSM interactions required.

To explain why this construction is scalable: since only a constant number of HSMs $n \ll N$ participate in the decryption process, the system scales well as the number of HSMs in the data center increases.

To explain why this construction should be secure: if the attacker cannot guess the client's PIN, the attacker does not know which set of n HSMs (out of the N total) it needs to compromise to recover the client's AES key. So, the best attacks are either to: guess the client's PIN or compromise a large fraction of the data center.

This argument requires that each individual key-share ciphertext leak no information about which HSM can decrypt it—a cryptographic property known as "key privacy" [8]. However, even key-private encryption schemes do not always remain secure against an adversary that adaptively compromises secret keys, which leads to our first technical challenge:

Challenge 1. *How can we ensure that the client's recovery ciphertext "leaks nothing" about which HSMs are required to decrypt the client's message, even against an attacker who can adaptively compromise HSMs?*

In Section 5, we explain how to solve this problem using *location-hiding encryption*, a new cryptographic primitive.

4.2 The recovery process

The client begins the recovery process holding:

- the public keys of all HSMs in the data center,
- its secret PIN, and
- its recovery ciphertext (which the client can fetch from the service provider).

First, the client asks the service provider to record its recovery attempt in the *append-only log*, implemented collectively by the service provider and HSMs. The log holds a mapping of identifiers to values. The service provider can insert new identifier-value pairs into the log but the service provider cannot modify or delete the values of defined identifiers, ensuring that there is at most one immutable value for each identifier.

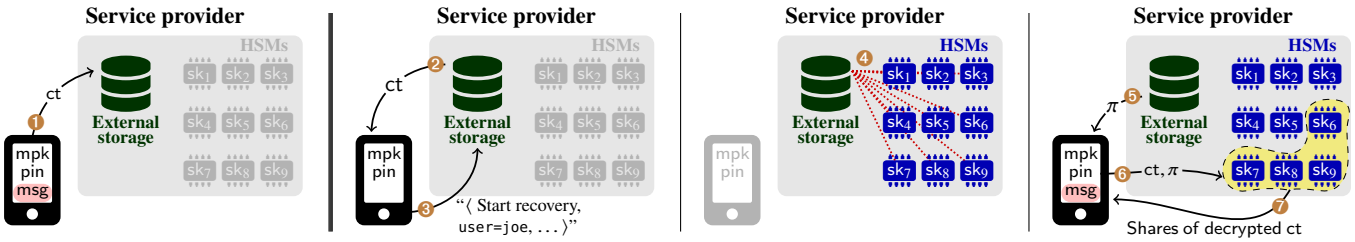


Figure 3: An overview of the recovery-protocol flow. Each HSM i holds a secret key sk_i. The client holds a vector mpk of all HSMs' public keys. ❶ During backup, the client uses its PIN and the master public key to encrypt its data msg into a recovery ciphertext ct. The client then uploads this recovery ciphertext ct to the service provider. ❷ During recovery, the client downloads its recovery ciphertext. ❸ The client asks the data center to log its recovery attempt. ❹ The service provider collects a batch of client log-insertion requests, updates the log, and aggregates the new log into a Merkle tree. The service provider and HSMs run a log-update protocol. At the end of this protocol, each HSM holds the root of the Merkle tree computed over the latest log. ❺ The service provider sends the client a Merkle proof π that the client's recovery attempt is included in the latest log (i.e., in the latest Merkle root). ❻ The client sends the recovery ciphertext ct and log-inclusion proof π to the subset of HSMs needed to decrypt the recovery ciphertext. ❼ The HSMs check the proof and return shares of the decrypted ciphertext to the client. The client uses these to recover the backed-up data msg.

The recovery attempt is logged as follows. The client begins by using public information (service name, username, and salt in the recovery ciphertext) along with its secret PIN to recover the subset of n HSMs it picked during backup. The client then hashes these values together with some randomness to produce a cryptographic commitment h to the identities of these HSMs and to its recovery ciphertext. The client then asks the service provider to insert the identifier-value pair (user, h) into the log, where user is the client's username. (In this discussion, we use the client's username as the key for simplicity. In practice, to preserve privacy, we might use an opaque device-install UUID.)

The service provider collects a batch of these log-insertion requests, produces a Merkle-tree [59] digest over the updated log, and runs a log-update protocol with the HSMs. At the end of this protocol, the HSMs hold the updated log digest. The service provider then returns to the client a Merkle proof π proving that the pair (user, h) appears in the latest log digest.

Since the service provider and HSMs run the log-update protocol periodically (e.g., every 10 minutes), the client will have to wait a few minutes on average to decrypt its backup. The client already has to download its large encrypted disk image, which will likely take minutes, so these steps can proceed in parallel.

The client then contacts its chosen set of n HSMs over an encrypted channel, such as TLS. The client sends to each HSM: its username, the opening of its commitment h (i.e., the values and randomness used to construct the commitment h), and the Merkle inclusion proof π. Each HSM

- recomputes the commitment h and checks the inclusion proof π (to confirm that the recovery attempt is logged), and
- decrypts its share of the client's AES key, confirms that the username in the decrypted plaintext matches the one provided by the client (which prevents user A from attempting to decrypt user B's ciphertext, in collusion with a malicious service provider).

If both of these checks pass, the HSM returns the AES-key share to the client.

Given any t of these decryption-key shares, the client can recover the AES key used to encrypt its backup. The client can then use this AES key to decrypt its backed-up message.

Since at most one log entry can exist per username, the use of the log ensures that each user can make at most one recovery attempt. In this way, the system defeats brute-force PIN-guessing attacks. With a slight modification, it is possible to allow each user to make a fixed number (e.g., 5) guesses, or a fixed number of guesses per time period (e.g., 5 per month).

A counter-intuitive property of this scheme is that the client never explicitly provides its PIN to the HSMs. The fact that the client knows which subset of the HSMs to contact implicitly proves the client's knowledge of the PIN because the set of n HSMs is much smaller than the total number of HSMs N.

This overview leaves some technical details unexplained. In particular:

Challenge 2. *How do the HSMs implement the append-only log without sacrificing scalability or security?*

A straightforward way to implement the log would be to have each HSM store the entire state of the log. But then every HSM would have to participate in every recovery attempt, which would not meet our scalability goals. Another implementation would be to have the data-center operators maintain the log, but then malicious data centers could violate the append-only property, and thus mount brute-force PIN-guessing attacks, without HSMs noticing.

In Section 6, we explain how the HSMs can collectively maintain such an append-only log in a scalable and secure manner. At a high level, the (potentially adversarial) data center maintains the state of the log, which we represent as a list of identifier-value pairs. Every time the data center wants to insert an identifier-value pair into the log, the data center must

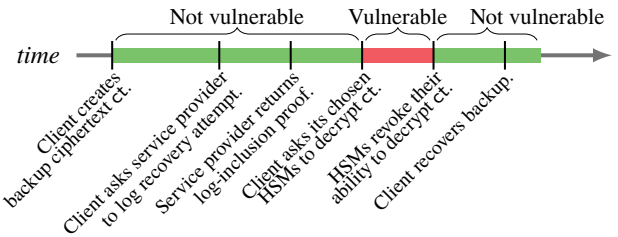


Figure 4: Since HSMs in SafetyPin revoke their ability to decrypt a client's recovery ciphertext, SafetyPin protects against HSM compromise attacks that take place before recovery begins and after it completes. An attacker who can compromise HSMs while recovery is in progress can break security.

prove to a random subset of the HSMs that the identifier to be inserted is undefined in the current log. Provided that at least one honest HSM audits each log-insertion, we can guarantee that the values associated with log identifiers are immutable (i.e., that we maintain the log's append-only property). In this way, (a) each HSM needs to participate in only a vanishing fraction of the recovery attempts and (b) even an attacker who can compromise many of the HSMs cannot break the append-only nature of the log.

One remaining issue is that an attacker who observes the data center network may see which HSMs a client interacts with during recovery and decide to compromise that exact set of HSMs after recovery completes.

Challenge 3. *For scalability, the client should only communicate with a small number of HSMs during recovery. But then how can we protect against an attacker who compromises these HSMs after recovery completes?*

Our idea is as follows: after a client runs the recovery protocol, each participating HSM *revokes* its ability to decrypt that client's recovery ciphertext. So, even if an after-the-fact attacker compromises the HSMs that participated in recovery, the attacker learns no useful information. The only window of vulnerability is at the moment after the client contacts its HSMs and before the HSMs complete revocation (Figure 4). Making this work on resource-limited HSMs requires new technical tools, which we describe in Section 7.

5 Protecting the mapping of users to HSMs with location-hiding encryption

In this section, we define and construct *location-hiding encryption*, which the client uses to encrypt its backup data.

The location-hiding encryption routine takes as input (1) a set of N public keys, (2) a short PIN, and (3) a message, and outputs a ciphertext. In our application, the N public keys are the public keys of the N HSMs in the data center.

The cryptosystem has three main properties, which we formalize in the full version [24]:

1. Security. To successfully decrypt the ciphertext, an attacker must either (a) guess the PIN or (b) control more than a constant fraction f_{secret} of the N total secret keys. This security property must hold even if the attacker can adaptively compromise an f_{secret} fraction of the N secret keys. In our application, this implies that unless an adversary can guess the PIN or compromise a constant f_{secret} fraction of the HSMs in the data center, it learns nothing about the client's backed-up data.

2. Scalability. Given the PIN used to encrypt the message, it is possible to decrypt the message using a small subset of the N secret keys corresponding to the N public keys used during encryption. In our application, a client who knows the correct PIN can recover its backup by interacting with only a small cluster of n HSMs (for some parameter $n \ll N$) out of the N total HSMs, So as N grows, each HSM needs to participate in a vanishing fraction of the total recovery attempts.

3. Fault tolerance. Given the PIN, it is possible to decrypt a ciphertext even if a random fraction f_{live} of all secret keys are unavailable. In our application, this implies clients can recover their backups even if an f_{live} fraction of all HSMs fail.

We call this primitive "location-hiding encryption" because there is a small set of n HSMs that the attacker could compromise to decrypt the ciphertext, but the cryptosystem *hides the location* of these HSMs within the larger pool of N HSMs.

Our construction

Our construction of location-hiding encryption is just a careful composition of existing primitives. However, it takes some analysis to prove that the composition provides the desired security properties. We describe our construction here in prose and we include the security definitions and proofs in the full version [24]. The construction makes use of a public-key encryption scheme (hashed ElGamal encryption [27, 15]) and an authenticated encryption scheme (e.g., AES-GCM).

Setup. In our construction, each HSM i, for $i \in [N]$, holds a keypair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for the public-key encryption scheme. Let $t \in \mathbb{Z}^{>0}$ be a threshold such that if each HSM fails with probability f_{live}, then in a random sample of n HSMs, there are at least t non-failed HSMs with extremely high probability. Our instantiation takes $t = n/2$ for $f_{\mathsf{live}} = \frac{1}{64}$.

Encryption. The encryption routine takes as input a list of N public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_N)$, a PIN, and a message msg. To encrypt the message using our location-hiding encryption scheme:

1. Sample a random AES key k and a random salt.
2. Split k into t-out-of-n-Shamir secret shares $k_1, \ldots, k_n$ [69].
3. Hash the PIN and salt and use the result as a seed to generate a list of n random indices $i_1, \ldots, i_n \in [N]$.
4. Encrypt each key-share k_j with public key pk_{i_j}.
5. Finally, return (a) the salt, (b) the n public-key ciphertexts, and (c) the AES encryption of msg under key k.

Decryption. To decrypt given the ciphertext and PIN:

1. Hash the salt and PIN to reconstruct the set of indices $i_1, \ldots, i_n \in [N]$ used during encryption.

2. Use secret keys $\mathsf{sk}_{i_1}, \ldots, \mathsf{sk}_{i_n}$ to decrypt the n shares of the AES key k. (In fact, only t of the shares are necessary.)
3. Using the recovery routine for Shamir secret sharing, recompute the AES key k from its shares.
4. Decrypt and return msg using the AES key k.

Notice that the decryption routine only uses the PIN to sample the set of secret keys used for decryption. In our application, this implies that the client never needs to explicitly provide its PIN (or even a hash of its PIN) to the HSMs; contacting the right subset of HSMs is enough to ensure that the client provided the correct PIN.

The intuition behind the security analysis is straightforward: with hashed ElGamal encryption, the ciphertext reveals no information about which n public keys (out of the N total where $n \ll N$) were used during encryption. Thus, the ciphertext reveals no information about which secret keys the attacker must compromise unless the attacker can guess the PIN. Without these secret keys, the attacker cannot learn anything about k, and therefore cannot decrypt the message.

In the full version [24], we formalize our location-hiding encryption scheme and prove that it is secure in the random oracle model when instantiated with the hashed ElGamal encryption (with certain constraints on n and N).

There are two reasons why the security analysis is non-trivial: First, we must ensure that the ciphertext leaks nothing about the n keys to which it was encrypted (i.e., that it is *key-private* [8]). Second, we must ensure that the encryption scheme remains secure even if an attacker can adaptively compromise secret keys. This is known as security under *selective-opening attack* [9, 29, 43]. Showing that both properties hold at once is the source of the technical complexity.

6 The distributed log

In SafetyPin, the HSMs collectively maintain a *distributed log*, which any external party can read and replay. The service provider maintains the log state and the HSMs monitor log insertions to ensure that the service provider does not violate the log's append-only property.

We use this log for two primary purposes:

1. **Limiting PIN guesses.** To prevent an attacker from brute-force guessing a client's PIN, we use the log (as described in Section 4) to enforce a global limit on the number of recovery attempts that the HSMs allow per username.
2. **Monitoring recovery attempts.** The service provider logs each recovery attempt, so any SafetyPin client can inspect the log to learn whether someone (e.g., a foreign attacker or snooping acquaintance) has tried to recover their backed-up data. A client could then take mitigating action—such as contacting their service provider, a law-enforcement agency, or the press.

A third use for the log—which comes directly from related work [30] and which we have not yet implemented—is to **manage HSM group membership**. Whenever the service provider wants to add or remove an HSM from the data center, the service provider operator could record this information in the log before the other HSMs will accept the change. All SafetyPin clients can thus verify that they are communicating with the same set of HSMs. In addition, clients can also detect suspicious changes in the set of HSMs in the data center. (For example, if the service provider replaces all HSMs in the data center over the course of a day.)

The log is simply a list of identifier-value pairs maintained by the service provider. Clients can insert identifier-value pairs in order to record recovery attempts, and HSMs maintain a digest of the log state. Our distributed log must satisfy the following key property:

> *If any honest HSM ever accepts that an identifier-value pair* $(\mathsf{id}, \mathsf{val})$ *is included in the log, the HSM should never accept that* $(\mathsf{id}, \mathsf{val}')$ *is included in the log, for any value* $\mathsf{val}' \neq \mathsf{val}$.

6.1 Underlying data structure

Terminology. The log L is a list of key-value pairs. Since we use the word "key" in this paper to refer to cryptographic keys, we call log keys "identifiers." We say that a log L' "extends" a log L if (a) L is a prefix of L' and (b) every identifier in L' appears at most once.

Our distributed log uses an authenticated data structure [75, 64, 77] that implements the following five routines:

$\mathsf{Digest}(L) \to d$. Return a constant-size digest d representing the current state of the log.

$\mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val}) \to \{\pi_{\mathsf{Inc}}, \perp\}$. Output a proof π_{Inc} that attests to the fact that the identifier-value pair $(\mathsf{id}, \mathsf{val})$ is in the log represented by digest $d = \mathsf{Digest}(L)$.

$\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}}) \to \{0, 1\}$. Return "1" iff π_{Inc} proves that the log that digest d represents contains $(\mathsf{id}, \mathsf{val})$.

$\mathsf{ProveExtends}(L, L') \to \{\pi_{\mathsf{Ext}}, \perp\}$. Output a proof π_{Ext} that $d' = \mathsf{Digest}(L')$ represents a log that extends the log that digest $d = \mathsf{Digest}(L)$ represents.

$\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) \to \{0, 1\}$. Return "1" iff π_{Ext} proves that the log that digest d' represents extends the log that digest d represents.

The inclusion and extension proofs must be complete (honest verifiers accept valid proofs) and sound (honest verifiers reject invalid proofs), as we define in the full version [24].

Implementing the data structure. Nissim and Naor [64] show that it is possible to implement these log primitives using only Merkle trees [59]. We summarize their construction in the full version [24]. At a very high level: the digest of the log is just the root of a Merkle tree computed over all of the entries of the log, represented as a binary search tree indexed by id. A log-inclusion proof π_{Inc} is a Merkle proof of inclusion relative to this root. A log-extension proof π_{Ext} is a proof that: (1) every identifier inserted to the new log did not exist in the old log and (2) the new digest represents the old log tree with the new values inserted. It is possible to prove both assertions

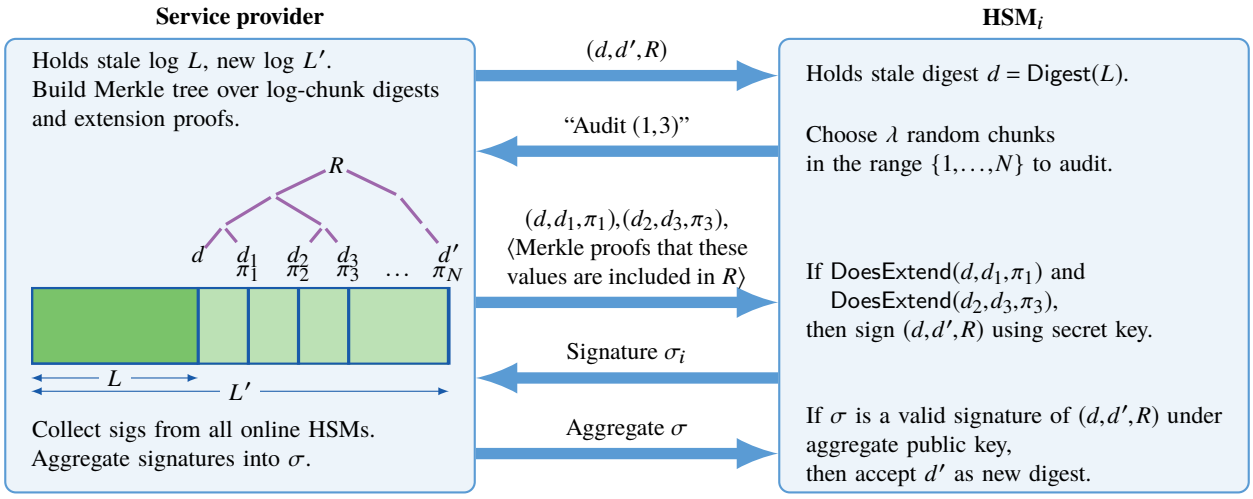


Figure 5: The protocol that the service provider and HSMs use to update the HSM's log digest.

using a number of Merkle proofs proportional to the number of log insertions.

6.2 Building a distributed log

We now explain how to use the primitives of Section 6.1 to build our distributed append-only log.

Initializing the log. The service provider maintains the entire state of the log L. Each HSM stores a log digest d which, in steady state, is the digest of the log L that the service provider holds. Initially, the log L is empty and each HSM holds the digest of the empty log.

Inserting into the log. A client can insert an entry (id,val) into the log by simply sending the pair to the service provider. The service provider adds this entry to its log state L.

Proving log membership to HSMs. Before the HSMs allow a client to begin the recovery process, the HSMs require proof that the client's recovery attempt is logged. Assume for the moment that the service provider holds a log L and all HSMs hold the up-to-date digest d = Digest(L). (We will explain how the HSMs get the latest log digest in a moment.) Then, a client can prove inclusion of any pair (id,val) in the log by asking the service provider for an inclusion proof. The service provider computes π_{Inc} = ProveIncludes(L,id,val) and returns the inclusion proof to the client. The client then sends (id,val,π_{Inc}) to the HSM, which can check DoesInclude(d,id,val,π_{Inc}) to be convinced that (id,val) is in the log represented by its digest d. This inclusion check is fast—logarithmic in the log length.

Updating the log digest at the HSMs. After a sequence of log-insertions, the service provider holds a log state L'. The HSMs will be holding a digest d = Digest(L) of a stale log L. If the service provider is honest, the new log L' extends the old log L.

To update the log digest at the HSMs, the service provider will first send the new digest d' = Digest(L') to every HSM. Next, the data center must convince each HSM that this new digest d' represents a log that extends the log L that the old digest d represents.

One non-scalable way to achieve this would be for the service provider to send an extension proof π_{Ext} = ProveExtends(L,L') to every HSM. The problem is that the time required to check this extension proof grows linearly with the number of new log entries. So if every HSMs checked the entire extension proof, the throughput of the system would not increase as the number of HSMs increases.

Instead, we use a randomized-checking approach, as in Figure 5. If there have been I insertions to the log since the last update, the service provider divides the updates into N chunks, each containing I/N insertions. The service provider then applies these chunks of updates to the old log L one at a time, producing a digest d_i and extension proof π_i for each of the N intermediate logs ($i \in \{1,\ldots,N\}$). The service provider then sends the root R of a Merkle-tree commitment to these digests to each HSM.

Each HSM then asks the service provider for a random λ-size subset of the intermediate digests and extension proofs, where λ is a security parameter. The service provider returns the requested digests and extension proofs and proves that these values are included in the Merkle root R. Each HSM checks its requested intermediate extension proofs using DoesExtend($\cdot$) and checks the Merkle proof relative to the root R. The HSMs auditing the first and last chunks also ensure that the intermediate digests match the old digest d and the new digest d', respectively.

If these extension and Merkle proofs are valid, each HSM signs the tuple (d,d',R) using an aggregate signature scheme [14], and returns the signature to the service provider. Once all online HSMs have signed, the service provider aggregates these signatures and broadcasts the aggregated signature to all HSMs. If any HSM fails during this process, the service

provider notifies the HSMs and they restart this log-update process. (In the full version [24], we describe how the log can make progress even if HSMs fail during the log-update protocol.)

The HSMs check the aggregate signature on (d, d', R) relative to the HSMs' aggregate public key. If the signature is valid, the HSMs accept the new digest d'.

Security. If there are at most f_{secret} compromised HSMs, then even if f_{secret} honest HSMs are slow, $(1 - 2f_{\mathsf{secret}})N$ honest HSMs will participate in any successful protocol execution. If each of these HSM audits C chunks, then the probability that no honest HSM audits a particular log chunk is

$$\Pr[\text{fail}] = \left(1 - \tfrac{1}{N}\right)^{(1-2f_{\mathsf{secret}})N \cdot C} \leq \exp\left((2f_{\mathsf{secret}} - 1) \cdot C\right).$$

(Here, we use the fact that $(1 - x) \leq \exp(-x)$.) If each HSM audits $C = \lambda \approx 128$ chunks, this failure probability is $\ll 2^{-128}$. In other words, some honest HSM will catch a cheating service provider with overwhelming probability. In addition, since all honest HSMs will expect a signature from all honest HSMs, this will cause the updating operation to fail and the system to halt. For this analysis, we assume that the adversary cannot adaptively compromise HSMs while the recovery protocol is running without taking them offline.

Scalability. Each HSM must check the extension proofs on λ chunks, where each chunk contains a $1/N$ fraction of the total updates in each epoch. Thus each HSM checks a vanishing fraction ($\frac{\lambda}{N}$) of log insertions. Each HSM checks one aggregate signature, which requires time independent of the number of HSMs [14]. Thus, the total work that each HSM performs per epoch decreases as the number of HSMs N increases.

Because we use the log primarily to limit the number of PIN attempts, garbage collection is straightforward. The service provider simply creates a new empty log, effectively resetting the number of PIN attempts for every user (old copies of the log can still be inspected to monitor recovery attempts). To ensure that the service provider does not run garbage collection and clear the state too frequently, each HSM will run garbage collection for a fixed number of times (e.g. the expected number of garbage collections over two years) before refusing to respond to further requests. This bounds the number of times the service provider can garbage collect the log.

6.3 Transparency and external auditability

Our log design allows *anyone* to audit the log to ensure that the service provider correctly maintains the log's append-only property. Additional auditors only add to the security of the system by adding another layer of protection, as they can detect log corruptions in the event that more than f_{secret} HSMs are compromised. In particular, for any two log digests d and d', an auditor can ask the data center for the entire logs L and L' corresponding to both of these digests. The auditor confirms that d is the root of the log tree for L and that d' is the root of the log tree for L'. Finally, the auditor checks that L' extends L.

As an extra precaution, users could specify external parties (e.g., Let's Encrypt) as designated auditors during backup. During recovery, the HSMs would only complete the recovery if these auditors sign the latest log digest. In this way, mounting a brute-force PIN-guessing attempt against a user would require compromising the user's external auditors as well.

The transparency log can also help with PIN re-use. As discussed in Section 8, instead of storing the salt directly with the service provider, the salt itself can be encrypted using a second round of location-hiding encryption and a null PIN. After recovery, the salt will be destroyed as discussed in the next section. Once the salt has been destroyed, the device restoring a backup can use the log to determine if anyone else has ever fetched the salt. If not, then it is safe for the user to re-use the old PIN.

As described in Section 4, the log contains usernames, which could be sensitive. To prevent leaking usernames, we would replace usernames with random device identifiers that are rerandomized when the device is factory reset. However, even with this modification, the log still leaks information about when and how often users restore backups, which the service provider may not wish to make public. While we hope that organizations would make their logs public, we acknowledge that some may only share their logs with several hand-picked organizations for auditing or may not share their logs at all. In these cases, our security guarantees still hold, although some of the transparency benefits are lost.

7 Forward security by puncturable encryption

We would like our encrypted-backup system to provide forward secrecy [18]. During the recovery process, the client reveals the identity of the $n \ll N$ HSMs that can decrypt its backup. Without forward secrecy, an attacker can break into these n HSMs to recover the client's backed-up data. Forward secrecy ensures that after recovery, an attacker, even one who compromises all HSMs in the data center, learns no information about the client's backup.

One seemingly straightforward way to provide forward secrecy would be to use a new keypair for each backup. However, because the client cannot interact with the HSMs it is encrypting to during backup (as this would reveal their identities), using a unique keypair for every backup would require every HSM in the data center to generate a new keypair for every backup, running counter to our scalability goals.

7.1 Background: Puncturable encryption

We instead achieve forward secrecy using puncturable public-key encryption [38, 39, 25, 21, 19, 26]. A puncturable encryption scheme is a normal public-key encryption scheme $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$, with one extra routine:

$\mathsf{Puncture}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{sk}_{\mathsf{ct}}$. Given a decryption key sk and a ciphertext ct, output a new secret key $\mathsf{sk}_{\mathsf{ct}}$ that can decrypt all ciphertexts that sk could decrypt except for ct.

Puncturable encryption for forward secrecy. To achieve forward security in SafetyPin, after an HSM decrypts its share of a client's recovery ciphertext ct, the HSM *punctures* its secret decryption key. The punctured key allows the HSM to decrypt all ciphertexts except for ct. Thus, if an attacker compromises *all* HSMs in the data center after a client has recovered its backup, the attacker will be unable to decrypt any backup images that clients have already recovered. Furthermore, if an attacker compromises at most $f_{\mathsf{secret}} \cdot N$ HSMs total, where f_{secret} is a parameter of the system that we define in Section 3, then the attacker will not be able to recover any backed-up data whatsoever.

Existing tool: Bloom-filter encryption. Our implementation uses a puncturable encryption scheme called Bloom-filter encryption [25]. There are only two details of Bloom-filter encryption that are important for this discussion.

1. *The secret key is large.* If a key supports $P \in \mathbb{Z}^{>0}$ punctures and we want decryption to fail with probability at most $2^{-\lambda}$, then the secret key for Bloom-filter encryption is an array of roughly λP elements of a cryptographic group $\mathbb{G}$. After P punctures, the secret key may no longer decrypt messages and it is necessary to rotate encryption keys.
2. *Puncturing is simple.* Puncturing the secret key just requires deleting λ elements in the data array that comprises the secret key.

Concretely, when we set the Bloom-filter-encryption parameters to suitable values for experimental evaluation, each Bloom-filter encryption secret key has size over 64 MB. Even high-end HSMs have only 1–2 MB of storage (Table 2), so storing such large keys on an HSM would be impossible.

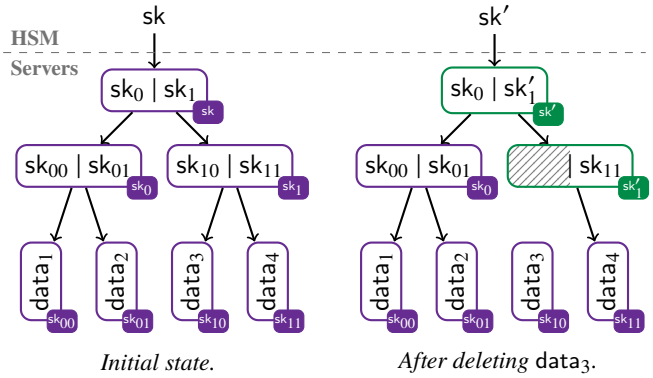
7.2 Outsourced storage with secure deletion

We show how to efficiently outsource the storage of this large secret key in a way that preserves forward secrecy of the punctured key. In particular, the HSM can outsource the storage of its secret-key array to the untrustworthy service provider, while still retaining the ability to delete portions of the key. Our technique applies to outsourcing the storage of any data array—not just secret keys—so we describe our secure-deletion approach in general terms.

Desired functionality. At a high level, the HSM has access to (a) a small amount of internal storage and (b) a large external block store, run by the service provider. The HSM wants to store an array of D data blocks at the provider $(\mathsf{data}_1, \ldots, \mathsf{data}_D)$. The HSM should be able to subsequently *read* or *delete* these blocks.

The following security properties should hold, even if the attacker, controlling the service provider, may choose the data-array and sequence of operations the HSM performs:

- **Integrity.** If the service provider tampers with the stored data in a way that could cause a *read* to return an incorrect result, the read operation outputs $\perp$. Otherwise, the read operation for a block i returns the value of the last data that the client wrote to block i.
- **Secure deletion.** If the service provider compromises the HSM after the HSM has run the *delete* operation for the ith data block, the attacker learns nothing about the data stored in block i. (This property implies a confidentiality property: the service provider learns nothing about the outsourced data.)



Initial state. *After deleting* data_3.

Figure 6: Our outsourced-storage scheme uses a tree of keys. An arrow $a \rightarrow b$ denotes that value b is stored encrypted under key a. A service provider that stores all values it sees and later compromises the HSM state (sk') still does not learn the deleted data_3 value.

For efficiency, the HSM storage requirements must be small (constant size) and the *read* and *delete* routines should run quickly (in time logarithmic in the size D of the data array). Unlike in ORAM [33, 34], our goal is not to hide the HSM's data-access pattern from the service provider. We aim only to hide the contents of the array.

7.3 Our secure outsourced storage scheme

We explain our construction here in prose. See the full version [24] for a more formal description.

Running the setup phase. During the setup phase, our outsourced-storage scheme builds a binary tree with D leaves. Every node of the tree contains a fresh symmetric encryption key. During setup, for each node in the tree with key sk_i, we encrypt the keys of the child nodes sk_{i0} and sk_{i1} with sk_i and store this ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{sk}_{i0} \| \mathsf{sk}_{i1})$ in outsourced storage. At the leaves of the tree, we encrypt the ith data block with the key sk_i at the ith leaf and we store the ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{data}_i)$ in outsourced storage.

For example, in Figure 6, we use sk_0 to encrypt sk_{00} and sk_{01} and we store the result in outsourced storage. We use key sk_{01} to decrypt data item 2. Thus, knowing the root key sk is enough to decrypt the entire tree and access every data element in the array.

Reading a data block. To retrieve the data block at index i, the HSM reads in the ciphertexts along the path from the tree root to leaf i. The HSM then decrypts the chain of ciphertexts from the root down to recover the data block at index i. For

example, in Figure 6, to retrieve data block 3, the HSM can use sk to decrypt sk_1, and sk_1 to decrypt sk_{10}, which it can use to decrypt data item 1.

Deleting a data block. To delete the data block at index i, the HSM recovers (as in retrieval) the keys along the path from the root to leaf i. At the node containing the key to decrypt data block i, the HSM deletes the key. It then chooses a fresh key and re-encrypts the other key at that node using the fresh key. To maintain the ability of the parent key to decrypt the child ciphertext, the HSM updates the parent of that node to contain the fresh key for its child and re-encrypts the parent's keys under a new key. It continues this up the path to the root, where the HSM chooses a new key sk' to encrypt the root. The HSM replaces sk with sk', deleting the old sk, and then sends the new ciphertexts along the path from the root to leaf i back to the service provider. For example, in Figure 6, to delete data item 3, the HSM decrypts the keys $(\mathsf{sk}_0 \| \mathsf{sk}_1)$ and $(\mathsf{sk}_{10} \| \mathsf{sk}_{11})$. The HSM then deletes sk_{10}, chooses a new key sk'_1 to encrypt sk_{11}, and then chooses a new key sk' to encrypt sk_0 and sk'_1. The HSM then replaces sk with sk'.

Efficiency. The setup time is linear in the size of the data array D. The runtimes of retrieval and deletion are both logarithmic in D, and require only symmetric-key operations. The HSM stores only the constant-sized root encryption key sk.

Security intuition. An HSM can always recover the keys necessary to decrypt a data item, provided the HSM did not previously delete any of the keys necessary for decryption. Integrity follows immediately from the security of the underlying authenticated encryption scheme. Finally, we ensure secure deletion by deleting the key necessary to decrypt a certain data item and updating the root key. Without the old root key, it is impossible to access the key necessary to decrypt the deleted data item.

Putting it together. To summarize: the HSMs use a puncturable encryption scheme to prevent the compromise of HSM secrets at time T from allowing an adversary to learn about backed-up data that was recovered any time before T. We implement puncturable encryption using Bloom-filter encryption and outsource the storage of the large secret decryption key using our new technique for outsourcing with secure deletion.

8 Extensions and deployment considerations

The full SafetyPin implementation has to deal with a number of additional issues, which we discuss now.

Failure during recovery. As discussed in Section 7, after participating in recovery, HSMs revoke their ability to decrypt the recovered ciphertext. One consequence is that a client cannot recover the same backup ciphertext twice. This raises the question of what happens if a replacement device fails during or shortly after recovery, or if a communication failure during recovery prevents the new device from receiving the replies from the HSMs.

To solve this problem, when a client initiates recovery, it first generates a fresh per-recovery keypair $(\mathsf{sk}, \mathsf{pk})$ for a public-key encryption scheme. The client backs up this secret key sk using SafetyPin before initiating its recovery. Next, the client sends the public key pk to each HSM and then begins the backup-recovery process. Each HSM encrypts its replies to the client under pk, and each HSM sends a copy of each reply to the data center. If a client device fails during recovery, a second, replacement client device can retrieve the backed-up secret key sk and use these to decrypt the replies stored at the data center. This scheme nests arbitrarily, thereby handling any number of consecutive device failures during recovery.

Incremental backups. In practice, mobile devices often generate incremental backups rather than encrypting the entire disk image for each backup. SafetyPin supports incremental backups in the following way. The user uses SafetyPin to store a single AES key, which the user also keeps on her phone. The user can then encrypt incremental backups under this AES key and upload the resulting ciphertext to the data center. When the user recovers, she recovers her AES key and can use this key to decrypt the incremental updates.

Multiple recovery ciphertexts. Clients back up their phones regularly (e.g., every three days), and will thus generate a series of recovery ciphertexts. We want to ensure that after a client recovers her backup from time t, the HSMs involved in recovery puncture their secret decryption keys so that they cannot decrypt that client's backups from earlier times $t' < t$, even if an attacker compromises all HSMs in the data center. To achieve this, in the puncturable-encryption step (Section 7), we have the client use the same salt for each recovery ciphertext it generates. In this way, the client will encrypt its series of backups to the same set of HSMs. When these HSM puncture their secret keys during the recovery process, they will destroy their ability to decrypt any previous recovery ciphertexts from the given client. After recovery, the client chooses a new salt to generate subsequent backups on its new device.

Preventing post-recovery PIN leakage. As we have discussed, an attacker that watches the client recover can learn a salted hash of the user's PIN, which can be used to mount an offline brute-force attack to learn the user's PIN.

One approach to protect against this attack would be to have each user store their salt in secret-shared form at a random set S_{salt} of HSMs, where S_{salt} is included in the client's recovery ciphertext. Then, provided that the attacker does not compromise this set of HSMs, the attacker would learn no useful information on the user's PIN, even after recovery. An attacker could always compromise every HSM in S_{salt}, but an attacker that can compromise only a f_{secret} fraction of HSMs in the data center would not be able to mount this attack against too many clients' salts. We hope to model and prove this multi-user PIN-protection property in future work.

Operation	Ops/sec
Pairing	0.43
ECDSA ver	5.85
ElGamal dec	6.67
$g^x \in \mathbb{G}_{P256}$	7.69

	Operation	Ops/sec
	HMAC-SHA256	2,173.91
	AES-128	3,703.70
I/O	RTT, HID (32b)	71.43
I/O	RTT, CDC (32b)	2,277.90
I/O	Flash read (32b)	≈166,000

Table 7: Microbenchmarks on SoloKey. Pairing is on BLS12-381 curve using the JEDI library [49]. Other public-key operations use NIST P256 curve.

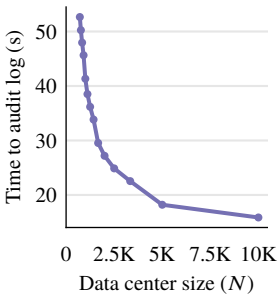


Figure 8: Log-audit time after inserting 10K recovery attempts for a log with roughly 100M recovery attempts. We only measure the auditing time for 100 HSMs as we only had 100 SoloKeys; we distribute the work as if there were N HSMs.

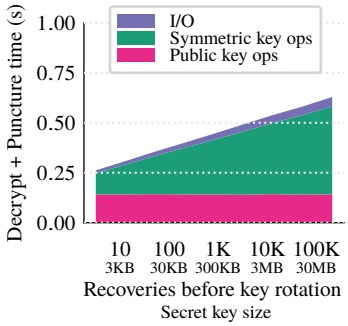


Figure 9: Time to run puncturable encryption on a single HSM as the maximum number of allowed punctures (and also secret key size) grows. The cost of our outsourced storage scheme dominates, though the access time is logarithmic in the size of the key.

9 Implementation and evaluation

We implemented SafetyPin on an experimental data cluster of 100 hardware security devices (Figure 1).

HSM. For the HSMs, we used SoloKeys [72], a low-cost open-source USB FIDO2 security key. SoloKeys use a STM32L432 microcontroller with an ARM Cortex-M4 32-bit RISC core clocked at 80MHz and 265KB of memory. The device is not side-channel resistant, but has a true random number generator and can lock its firmware. We add roughly 2,500 lines of C code to the open-source SoloKey firmware [71].

By default, SoloKeys communicate with the USB host via USB HID, an interrupt-based USB class used typically for keyboards and mice that has a maximum throughput of 64KBps. To improve performance, we rewrote parts of the firmware to use USB CDC, a high-throughput USB class commonly used for networking devices. This gave a roughly 32× increase in I/O throughput (Table 7).

For the puncturable-encryption scheme (Section 7.1), we use a variant of Bloom-filter encryption [25] that avoids the need for pairings [13] but increases the size of the HSMs' public keys. For the aggregate signature scheme needed for the log, we use BLS-style multisignatures [12] over the JEDI [49] implementation of the BLS12-381 curve.

Our implementation does not encrypt communication between the client and HSMs. Based on the time to run AES-128 and ElGamal encryption on the SoloKeys, we estimate that transport-layer encryption would add two ElGamal decryptions and 2KB of AES operations per recovery, increasing recovery time by approximately 0.3 seconds, or 30%. This overhead is comparatively high because processing a recovery only requires a handful of symmetric and public key operations.

Service Provider. Our service provider host is a Linux machine with an Intel Xeon E5-2650 CPU clocked at 2.60GHz. Our service-provider implementation is roughly 3,800 lines of C/C++ code (excluding tests) and uses OpenSSL.

Client. Our client device is a Google Pixel 4. Our implementation is roughly 2,300 lines of C/C++ code (excluding tests) and uses OpenSSL.

9.1 Microbenchmarks

Log. Figure 8 demonstrates how increasing the number of HSMs reduces the log-digest update time. We assume that the log is periodically garbage collected (i.e., approximately once a month), so that it holds at most a hundred million recovery attempts at once. If the HSMs run the log-update process every 10 minutes, each HSM spends approximately 11% of its active cycles auditing the log. The choice of how often to update the log is a tradeoff between how long users must wait to recover their backups and the total number of write cycles to non-volatile storage permitted by the hardware.

Puncturable encryption. Figure 9 shows the cost of performing a decrypt-and-puncture operation as the number of supported punctures increases. The AES operations associated with our scheme for outsourced storage with secure deletion (Section 7.2) dominate the cost.

Another way to implement outsourced storage with secure deletion would be to have the HSM store the outsourced array encrypted under a single AES key k. To delete an item, the HSM would read in the entire array, delete the item, and write out the entire array encrypted under a fresh key k'. With this approach, a deletion takes 48 minutes for a 64 MB array (the size of our outsourced secret keys). Our scheme thus improves system throughput by roughly 4,423×.

Each HSM punctures its secret key (Section 7.1) once after each decryption it performs. Since our puncturable-encryption scheme only supports a fixed number of punctures, each HSM must periodically rotate its encryption keys. We configure our puncturable-encryption scheme to allow each HSM to perform roughly 2^{18} decryptions before it must rotate its keys (rotation is triggered when half of the elements of the secret key have been deleted). Key rotation is expensive: we estimate (based on the number of public-key operations required) that key rotation on our HSMs will take roughly 75 hours. Each HSM spends approximately 139.4 hours processing recoveries and maintaining the log between key rotations. Therefore, each HSM spends roughly 56% of its cycles rotating its keys, and

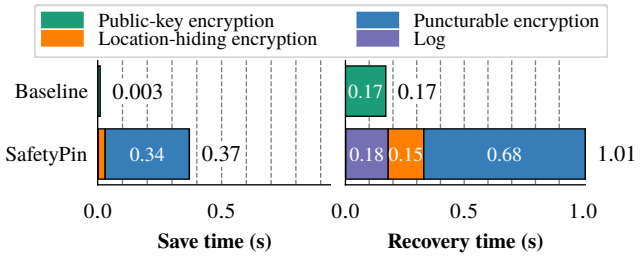


Figure 10: Breakdown of time to save (on Android Pixel 4 phone) and recover (using our SoloKey cluster). We do not consider the time to encrypt or decrypt disk images.

each HSM can process 1,503.9 recoveries per hour on average.

9.2 End-to-end costs

Parameters. We estimate that on average, each user will run recovery once a year. (There are 3.8B smartphone users [73] and 1.5B smartphones sold annually [74], so we expect $1.5/3.8 = 0.39 \ll 1$ recovery/user/year.) We calculate that a SafetyPin deployment of $N = 3{,}100$ HSMs could support one billion users. So, we treat our small cluster of 100 HSMs as a representative slice of a larger data center of $N = 3{,}100$ HSMs. Within this larger data center, each client shares its recovery keys among a cluster of $n = 40$ HSMs. This choice of n is based on the size of the data center N and PINs with six decimal digits, and is dictated by bounds we prove in the full version [24]. We set the puncturable encryption keys to allow 2^{20} punctures, as we found this provides a reasonable tradeoff between the time to decrypt and puncture and the time between key rotations. With these parameters, we maintain secrecy if at most an $f_{\mathsf{secret}} = \frac{1}{16}$ fraction of the HSMs are compromised (or $f_{\mathsf{secret}} \cdot N \approx 194$ total). We allow data recovery if at most an $f_{\mathsf{live}} = \frac{1}{64}$ fraction fail due to benign hardware failures (or $f_{\mathsf{live}} \cdot N \approx 48$ total).

Baseline. We compare against an encrypted-backup system modeled on the ones that Google and Apple use [82, 47]. To backup, the client selects a fixed cluster of five HSMs and encrypts her recovery key and a hash of her PIN under the cluster's public key. At recovery, the client sends the recovery ciphertext and a hash of her PIN to the cluster, and any HSM in the cluster can decrypt the ciphertext, check that the PIN hashes match, and return the recovery key. To defeat brute-force PIN-guessing attacks, each HSM independently limits the number of recovery attempts allowed on a given ciphertext.

Client overhead. Figure 10 gives the overhead of generating a backup in SafetyPin, compared to the baseline. The backup process takes 0.37 seconds. SafetyPin recovery ciphertexts are 16.5KB, versus 130B for our baseline, though we expect encrypted disk image to dominate the ciphertext size.

SafetyPin increases the bandwidth cost at the client. In the baseline scheme, the client downloads five public keys—one from each of its five chosen HSMs. In SafetyPin, the client must fetch a copy of all HSMs' public keys. (This way, the service provider does not learn the subset of HSMs to which the client is encrypting its backup.) So, when a client first joins the system, the client must download all these keys (11.5MB). Whenever an HSM rotates its puncturable-encryption keys, clients must download the HSM's new public key. In a deployment of $N = 3{,}100$ HSMs supporting one billion recoveries annually, we estimate that each SafetyPin client must download 1.97MB of keying material daily. Increasing the puncturable encryption failure probability would decrease client bandwidth, although this would require decreasing the fraction of HSMs allowed to fail, f_{live}. If a client goes offline for several days, it must download the rotated public keys for each day it spent offline (roughly 2MB/day), up to a maximum of 11.5MB (the size of all HSMs' keys). However, the client only needs to store the public keys for the n HSMs comprising its chosen recovery cluster which amounts to 9.02KB.

Recovery time. At a cluster size of $n = 40$ HSMs, Figure 11 shows that the end-to-end recovery time takes 1.01 seconds. Puncturable-encryption operations dominate recovery time (Figure 10), since these require expensive elliptic-curve operations for ElGamal decryption and many I/O and AES operations in order to perform secure deletion (Section 7.2).

Tail latency. In a deployment of SafetyPin, it will be important to consider not only the average throughput of the SafetyPin cluster, but also the request latency. Since recovery requests will arrive concurrently and in a bursty fashion, we will need to overprovision the system slightly to ensure that request tail latency does not grow too high, even under large transient loads. In Figure 13, we model how many HSMs are required to achieve various 99th-percentile latencies, while handling different average throughputs. We compute these values by modeling incoming requests using a Poisson process and each HSM using a M/M/1 queue with service times derived from our experimental results. As the figure demonstrates, by increasing the total number of HSMs, we can reduce the tail latency even when accounting for request contention. We anticipate that recovery time will in practice be dominated by the time to download the encrypted disk image, and so as long as the tail latency is less than or close to this time, any delay is unlikely to be noticed by the user.

Financial cost. Figure 12 shows how throughput scales as the outlay on HSMs increases and Table 14 presents dollar-cost estimates for SafetyPin deployments with different types of HSMs. For a configuration that tolerates the compromise of 50 high-quality HSMs, we estimate that adding SafetyPin to an unencrypted backup system would increase the system's dollar cost by 2.5%.

10 Related work

Today's encrypted-backup systems rely either on the security of hardware security modules [37, 47], secure microcontrollers [3], or secure enclaves [55, 57]. Vulnerabilities in these hardware components leave encrypted-backup systems

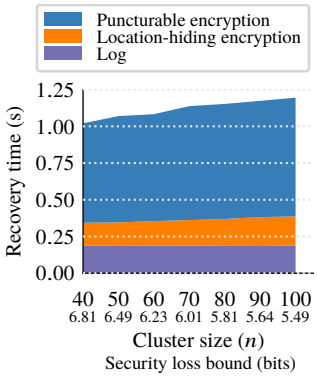


Figure 11: Recovery time grows slowly as cluster size n increases. The bits of security lost refers to the difference between the advantage of an attacker against a SafetyPin deployment with the given value of n and an attacker trying to guess the user's PIN.

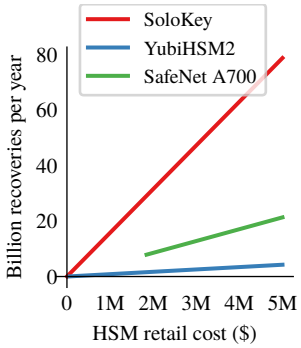
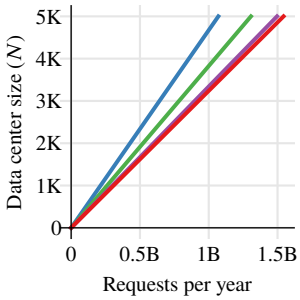


Figure 12: Estimated number of SafetyPin-protected recoveries per year supported by clusters of different HSM models and costs. We use g^x/sec to compute the expected throughput of more powerful HSMs based on our measurements using SoloKeys (Table 2).



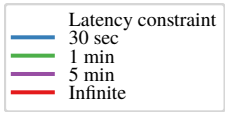


Figure 13: Data center sizes necessary to process different request rates with various 99th-percentile latency requirements.

open to attack. And there is ample evidence of vulnerabilities in both HSMs [66, 60, 63, 41, 46, 17, 31, 2] and enclaves [79, 16, 36, 53, 28, 80, 81, 61, 20, 40, 52, 11], and reason for concern about hardware backdoors as well [76, 83, 7, 48].

Many companies including Anchorage [5], Unbound Tech [78], Curv [22], and Ledger Vault [51], offer systems for secret-sharing cryptocurrency secret keys across multiple hardware devices. Unlike SafetyPin, these solutions use a small fixed set of HSMs, so they cannot simultaneously provide scalability and protection against adaptive HSM compromise.

In recent theoretical work, Benhamouda et al. show how to scalably store secrets on proof-of-stake blockchains when an adversary can adaptively corrupt some fraction of the stake [10]. They face many of the same cryptographic challenges that we tackle in Section 5; their theoretical treatment complements our implementation-focused approach. While they use proactive secret sharing to periodically re-share the secret and hide the secret from an adversary controlling some fraction of the stake, our approach allows a party with some low-entropy secret to recover the high-entropy secret.

Transparency logs inspire our log design [6, 50, 58, 1, 54]. While these logs allow a powerful auditor to verify correctness,

	HSM Qty.	f_{secret}	N_{evil}	Cost
SoloKey [72]	3,037	1/16	189	$60.7K
YubiHSM2 [84]	1,732	1/16	108	$1.1M
SafeNet A700 [68]	40	1/20	2	$738.7K
– 10 evil HSMs	320	1/32	10	$3.0M
– 50 evil HSMs	800	1/16	50	$14.8M
Estimated cost of storing 4GB $\times 10^9$ *users per year:*				$600M

Table 14: The estimated hardware cost of a SafetyPin deployment supporting one billion users, if each user recovers once per year. The N_{evil} number is how many corrupt HSMs the deployment tolerates.

We estimate the storage cost using AWS S3 infrequent access [4] ($0.0125 per GB/month). We estimate YubiHSM2 and SafeNet HSM throughput using their data sheets (Table 2). When computing the number of HSMs necessary to service a billion users, we account for key-rotation time. A cluster of 40 SafeNet HSMs can meet the throughput demands of one billion users, so we also consider larger deployments tolerating more compromised HSMs.

they do not easily allow distributing the work of auditing across many less powerful participants. The proofs we provide to the HSMs about the state of the log draw on work on authenticated data structures [75, 56, 65] and cryptocurrency light clients [62]. Kaptchuk et al. show how public ledgers can be used to build stateful systems from stateless secure hardware [45], and they show how their techniques can be applied to Apple's encrypted-backup system. This work is complementary to ours, as they show how to securely manage state in cases where HSMs do not have secure internal non-volatile storage (an assumption we make in SafetyPin).

11 Conclusion

SafetyPin is an encrypted backup system that (a) requires its users to only remember a short PIN, (b) defeats brute-force PIN-guessing attacks using hardware protections, and (c) provides strong protection against hardware compromise. SafetyPin demonstrates that it is possible to reap the benefits of hardware security protections without turning these hardware devices into single points of security failure.

Acknowledgments. We would like to thank Raluca Ada Popa and Bryan Ford for their support throughout this project. Albert Kwon, Anish Athalye, Christian Mouchet, David Lazar, Dima Kogan, and Lefteris Kokoris-Kogias, offered thoughtful criticism on drafts of this work. We thank our shepherd Sebastian Angel for his work reviewing our camera-ready. We thank Dan Boneh for useful suggestions on how to simplify the security analysis of our location-hiding encryption scheme, we thank Vinod Vaikuntanathan for the suggestion to avoid pairings in our puncturable-encryption scheme, and we thank Keith Winstein for early brainstorming on passwords and PINs. Conor Patrick and Nicolas Stalder gave helpful suggestions for modifying the SoloKey firmware to support USB CDC, and Vivian Fang provided advice on assembling the system. Finally, we thank the anonymous reviewers of USENIX Security 2020 and OSDI 2020 for their thorough and detailed feedback. This research is also supported in part by the RISELab, a Facebook Research Award, and a NSF GRFP fellowship.

References

[1] Trillian. https://github.com/google/trillian.

[2] CVE-2015-5464. Available from MITRE, CVE-ID CVE-2015-5464., February 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5464.

[3] Titan in depth: Security in plaintext. Google, August 2017. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[4] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/, Accessed 12 February 2020.

[5] Anchorage. https://anchorage.com/, Accessed 25 May 2020.

[6] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.

[7] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *CHES*. Springer, 2013.

[8] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2001.

[9] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In *EUROCRYPT*, pages 1–35. Springer, 2009.

[10] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a blockchain keep a secret? *IACR Cryptology ePrint Archive*, 2020.

[11] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security*, pages 1213–1227, 2018.

[12] Dan Boneh, Manu Drijvers, and Gregory Neven. BLS multi-signatures with public-key aggregation. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html, Accessed 23 May 2020, 03 2018.

[13] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[14] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432. Springer, 2003.

[15] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. 2020.

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX WOOT*, 2017.

[17] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. Bios chronomancy: Fixing the core root of trust for measurement. In *CCS*, pages 25–36. ACM, 2013.

[18] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, pages 255–271, 2003.

[19] Ran Canetti, Srinivasan Raghuraman, Silas Richelson, and Vinod Vaikuntanathan. Chosen-ciphertext secure fully homomorphic encryption. In *IACR International Workshop on Public Key Cryptography*, pages 213–240. Springer, 2017.

[20] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE: Stealing intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[21] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. *SIAM*, 47(6):2157–2202, 2018.

[22] Curv. https://www.curv.co/, Accessed 25 May 2020.

[23] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, 2014.

[24] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. Safetypin: Encrypted backups with human-memorable secrets. *arXiv preprint arXiv:2010.06712*, 2019.

[25] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In *EUROCRYPT*, pages 425–455. Springer, 2018.

[26] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: forward secrecy, improved security, and applications. In *IACR International Workshop on Public Key Cryptography*, pages 219–250. Springer, 2018.

[27] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[28] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[29] Serge Fehr, Dennis Hofheinz, Eike Kiltz, and Hoeteck Wee. Encryption schemes secure against chosen-ciphertext selective opening attacks. In *EUROCRYPT*, pages 381–402. Springer, 2010.

[30] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *Symposium on Operating Systems Design and Implementation*, pages 233–248, 2006.

[31] Jean-Baptiste Bédrune Gabriel Campana. Everybody be Cool, This is a Robbery! Blackhat, 2019. https://www.blackhat.com/us-19/briefings/schedule/#everybody-be-cool-this-is-a-robbery-16233.

[32] Shirley Gaw and Edward W Felten. Password management strategies for online accounts. In *SOUPS*, pages 44–55. ACM, 2006.

[33] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[35] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[36] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *European Workshop on Systems Security*, pages 1–6, 2017.

[37] Matthew Green. Is Apple's Cloud Key Vault a crypto backdoor?, 2016. https://blog.cryptographyengineering.com/2016/08/13/is-apples-cloud-key-vault-crypto/.

[38] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *Security and Privacy*. IEEE, 2015.

[39] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[40] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, pages 44–60. Springer, 2018.

[41] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A bad dream: Subverting trusted platform module while you are sleeping. In *USENIX Security*, pages 1229–1246, 2018.

[42] SM Haque, Matthew Wright, and Shannon Scielzo. A study of user password strategy for multiple accounts. In *Data and application security and privacy*, pages 173–176. ACM, 2013.

[43] Dennis Hofheinz and Andy Rupp. Standard versus selective opening security: separation and equivalence results. In *Theory of Cryptography Conference*, pages 591–615, 2014.

[44] Intel. Strengthen enclave trust with attestation. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html, Accessed 5 February 2020.

[45] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *NDSS*, 2019.

[46] Bernhard Kauer. Oslo: Improving the security of trusted computing. In *USENIX Security*, volume 24, page 173, 2007.

[47] Ivan Krstic. Behind the scenes with iOS security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[48] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept. 2014.

[49] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. In *USENIX Security*, pages 1519–1536, 2019.

[50] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. https://tools.ietf.org/html/rfc6962.

[51] Ledger vault. https://www.ledger.com/vault, Accessed 25 May 2020.

[52] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[54] S. Li, C. Man, and J. Watson. Delegated Distributed Mappings. Internet-Draft draft-watson-dinrg-delmap-02, Internet Engineering Task Force, April 2019. https://tools.ietf.org/html/draft-watson-dinrg-delmap-02.

[55] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/.

[56] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[57] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 10(1), 2013.

[58] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[59] Ralph Merkle. A certified digital signature. In *CRYPTO*, pages 218–238. Springer, 1989.

[60] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[61] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Security and Privacy*. IEEE, 2020.

[62] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[63] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of Coppersmith's attack: Practical factorization of widely used RSA moduli. In *CCS*, pages 1631–1648. ACM, 2017.

[64] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security Symposium*, 1998.

[65] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *International conference on information and communications security*, pages 1–15. Springer, 2007.

[66] Ryan Paul. Infineon DRM/encryption chip succumbs to physical attack. Ars Technica, 2010. https://arstechnica.com/information-technology/2010/02/infineon-drmencryption-chip-succumbs-to-physical-attack/.

[67] Adam Powers. FIDO TechNotes: The truth about attestation. https://fidoalliance.org/fido-technotes-the-truth-about-attestation/, Accessed 25 May 2020.

[68] SafeNet Luna network hardware security modules A700 - cryptographic accelerator. https://www.insight.com/en_US/shop/product/908-000366-001-000/GEMALTO/908-000366-001-000/SafeNetLunaNetworkHardwareSecurityModulesA700-Cryptographicaccelerator-GigE-1U-rack-mountable/, Accessed 5 February 2020.

[69] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[70] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *SOUPS*, page 2. ACM, 2010.

[71] Solokeys. Solo. https://github.com/solokeys/solo, Accessed 5 February 2020.

[72] Solokeys. https://solokeys.com/, Accessed 5 February 2020.

[73] Statista. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[74] Statista. Number of smartphones sold to end users worldwide from 2007 to 2020. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/, Accessed 23 May 2020.

[75] Roberto Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[76] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design Test of Computers*, 27(1), Jan 2010.

[77] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, pages 1299–1316, 2019.

[78] Unbound tech. https://www.unboundtech.com/, Accessed 25 May 2020.

[79] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[80] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[81] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[82] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[83] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog malicious hardware. In *Security and Privacy*. IEEE, May 2016.

[84] YubiHSM 2. https://www.yubico.com/product/yubihsm-2, Accessed 5 February 2020.

A Artifact Appendix

A.1 Abstract

The SafetyPin implementation is split into two components:

- **HSM:** The HSMs (hardware security modules) are used to recover user secrets. Our implementation uses SoloKeys, which are low-cost HSMs. We add roughly 2,500 lines of C code to the open-source SoloKey firmware.
- **Host:** The host implements functionality for the user and data center, including saving secrets, maintaining the log, and coordinating HSMs. Our implementation is roughly 3,800 lines of C/C++ code.

We implement the protocol described in the paper above. To improve performance, we rewrote parts of the SoloKey firmware to use USB CDC, a high-throughput USB class commonly used for networking devices. This results in roughly a 32× increase in I/O throughput. Our artifact is available at:

`https://github.com/edauterman/SafetyPin`

A.2 Artifact check-list

- **Hardware:**
  - 100 SoloKeys
  - 10 Anker SuperSpeed USB 3.0 hubs
  - 2 4-port USB PCIe controller cards
  - Linux machine with Intel Xeon E5-260 CPU clocked at 2.60GHz
- **Compilation:** The ARM compiler for the SoloKeys, and `gcc` for the host.
- **Metrics:** Latency
- **Experiments:** Log-audit time, puncturable encryption overhead, breakdown of recovery time, cluster size vs. recovery time
- **Required disk space:** 14MB
- **Expected experiment run time:** 50 minutes
- **Public link:** `https://github.com/edauterman/SafetyPin`
- **Code licenses:** Apache v2

A.3 Description

A.3.1 How to access

We provide reviewers with credentials to remotely access our system. Instructions for assembling a similar system are available here:

`https://github.com/edauterman/SafetyPin/blob/master/SETUP.md`.

A.3.2 Hardware dependencies

Our artifact uses SoloKeys as low-cost HSMs. We use 100 SoloKeys for our experiments, although other deployments

could use a different number of HSMs. Because of limitations in the Intel XCHI controller, which supports a maximum of 96 endpoints (each USB 3.0 device has 3 endpoints), we installed PCIe cards to support additional endpoints. This is not necessary for smaller-scale deployments, but larger deployments should choose a host that supports installing such PCIe cards or find another solution. We also recommend USB hubs with an external power source such as the Anker hubs.

A.3.3 Software dependencies

The firmware for the SoloKeys builds on the original SoloKey firmware, which already includes several libraries for cryptographic primitives on embedded devices:

`https://github.com/solokeys/solo`.

To support pairings for aggregate signatures, we use the jedi-pairing library for embedded devices:

`https://github.com/ucbrise/jedi-pairing/`.

For USB HID support, we use the Signal11 library:

`https://github.com/signal11/hidapi`.

We implement our cryptographic primitives that do not require pairings at the host using OpenSSL.

A.4 Installation

Instructions for building the host are available under `host/`. Instructions for building the firmware and flashing the SoloKeys are available under `hsm/`. The SoloKey documentation provides additional details and troubleshooting for building and flashing the SoloKeys:

`https://docs.solokeys.io/`.

When experimenting with SafetyPin, you should not boot SoloKeys in DFU mode, as this locks the firmware and will prevent you from modifying the firmware later (e.g. to load an updated version of the SafetyPin source).

A.5 Experiment workflow

Reviewers can remotely access our machine and run all experiments by executing `./runAll.sh` in `bench/`. More detailed instructions for running individual experiments are available here:

`https://github.com/edauterman/SafetyPin#instructions-for-artifact-evaluation`.

A.6 Evaluation and expected result

Run all experiments by executing `./runAll.sh` in `bench/`. This will produce figures in `bench/out` that match Figure 8, Figure 9, Figure 10, and Figure 11. Note that we only reproduce the recovery time breakdown in Figure 10. Additionally, the configuration we set up for the reviewers only uses 90 HSMs for Figure 8 and Figure 11. We do this to keep different firmware on the remaining 10 HSMs to measure the breakdown in puncturable encryption time as the secret key size increases (Figure 9). For the experiments we show in the body of the paper, we re-flashed HSMs between experiments so that we could use all 100 HSMs to generate Figure 8 and Figure 11.

A.7 Experiment customization

The experiment for Figure 8 can be modified to measure different data center sizes without changing the firmware on the HSMs. The experiment for Figure 9 can likewise be modified to measure different secret key sizes, although this requires changing HSM firmware. If we had more HSMs, we could easily expand Figure 11 to show the effect of larger cluster sizes. We do not measure cluster sizes less than 40 because our analysis shows that our security guarantees begin to break down below this point.

A.8 Notes

To switch between USB CDC and USB HID, change the HID flag on both the host and the HSMs (this requires loading new firmware on the HSMs). More detailed instructions are available here:

`https://github.com/edauterman/SafetyPin/blob/master/SETUP.md`.

Note that rather than generating puncturable encryption secret keys on the HSM (a process we estimate would take roughly 75 hours), to run our experiments efficiently, we generate the secret key on the host (for security, a real-world deployment would need to generate this secret key on the HSM).

A.9 AE Methodology

Submission, reviewing and badging methodology:

`https://www.usenix.org/conference/osdi20/call-for-artifacts`