SafetyPin: Encrypted Backups with Human-Memorable Secrets

Emma Dauterman (UC Berkeley), Henry Corrigan-Gibbs (EPFL and MIT CSAIL), and David Mazières (Stanford)

OSDI 2020
Mobile backups today

1. User only needs to remember her **screen-lock PIN**.
   + PINs are easy to remember and not known to the service provider.
   - PINs have low-entropy.

2. **Secure hardware** prevents brute-force attacks against PIN.
   + Hardware security modules (HSMs) are resistant to physical attacks.
   - HSMs are not perfect.
State-of-the-art vs. SafetyPin: security

State-of-the-art

1 compromise = Millions of backups

SafetyPin

< N/16 compromises = 0 backups

SafetyPin tolerates many HSM compromises.
State-of-the-art vs. SafetyPin: **scalability**

**State-of-the-art**
- Client talks to a **single HSM**.

**SafetyPin**
- Client talks to a **few HSMs** (e.g. 40).

SafetyPin doesn’t sacrifice **scalability**.
State-of-the-art vs. SafetyPin: fault tolerance

Tolerates many failures between backups (e.g. N/64).
More HSMs improve security and performance

Claim: compromising more HSMs is more expensive

- The cost of **physical attacks** scales linearly with the number of HSMs.
- Physically attacking more HSMs increases the **risk of exposure**.
- Some protection against software bugs with diverse HSMs.
Attack scenario

Service provider (e.g. Google) sets up the system correctly.

Later, attacker wants to obtain user backups.

Attacker can:
• observe/modify network traffic,
• control the storage server, and
• adaptively compromise many HSMs.
Attack scenario

Service provider (e.g. Google) sets up the system correctly.

Later, attacker wants to obtain user backups.

Attacker can:
- observe/modify network traffic,
- control the storage server, and
- adaptively compromise many HSMs.
Attack scenario

Service provider (e.g. Google) sets up the system correctly.

Later, attacker wants to obtain user backups.

Attacker can:
• observe/modify network traffic,
• control the storage server, and
• adaptively compromise many HSMs.
Attack scenario

Service provider (e.g. Google) sets up the system correctly.

Later, attacker wants to obtain user backups.

Attacker can:
- observe/modify network traffic,
- control the storage server, and
- adaptively compromise many HSMs.
Outline

1. SafetyPin design
   • Overview
   • Scalable rate-limiting

2. SafetyPin evaluation
SafetyPin design idea

**Idea:** Hide the identities of a small, fixed set of HSMs a user can recover with.

- **Scalability:** The user can recover with a small set of HSMs.
- **Security:** Without the PIN, the attacker can’t identify the set of HSMs.

[System parameterized to support 1B recoveries per year with 128-bit security]
Backup [simplified]

1. Select $n \approx 40$ HSMs using a PIN.

2. Break message into $n \approx 40$ shares $m_1, \ldots, m_n$ such that $n/2$ needed to recover.

3. Encrypt a share $m_i$ to each selected HSM $i$. 

message
Backup [simplified]

1. Select $n \approx 40$ HSMs using a PIN.

2. Break message into $n \approx 40$ shares $m_1, \ldots, m_n$ such that $n/2$ needed to recover.

3. Encrypt a share $m_i$ to each selected HSM $i$. 

$H_{\text{user}}(\text{pin})$

message
Backup [simplified]

1. Select $n \approx 40$ HSMs using a PIN.

2. Break message into $n \approx 40$ shares $m_1, \ldots, m_n$ such that $n/2$ needed to recover.

3. Encrypt a share $m_i$ to each selected HSM $i$. 

---

**Diagram:**
- A smartphone labeled $H_{user}(pin)$ connected to a server.
- The server has multiple slots, one of which is highlighted with a red rectangle.
- Messages $m_1$ to $m_n$ are shown.
Backup [simplified]

1. Select $n \approx 40$ HSMs using a PIN.

2. Break message into $n \approx 40$ shares $m_1, \ldots, m_n$ such that $n/2$ needed to recover.

3. The attacker can't identify the set of HSMs without the PIN because
   - client doesn't interact with HSMs, and
   - ciphertexts don't leak identities of HSMs.

$H_{user}(pin)$

message
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of \( n \approx 40 \) HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message from responses.
Recovery [simplified]

1. Retrieve ciphertexts.
2. Compute the set of $n \approx 40$ HSMs.
3. Send the ciphertexts to the HSMs for decryption.
4. Assemble the message.

Attacker watching user recover learns information about the PIN.
Outline

1. SafetyPin design
   - Overview
   - Scalable rate-limiting

2. SafetyPin evaluation
Scalable rate-limiting

Each HSM must enforce a global PIN attempt limit for every user.

**Problem:** Introduces a scalability bottleneck.

**Tool:** Public append-only log maintained by HSMs.

[Simplified version that allows a single recovery attempt]
Distributed log design

Log structured as set of **key-value pairs**.

**Core invariant:** If a HSM accepts (key, value), it will not accept (key, value’) where value !≠ value’.
Distributed log design

Log structured as set of **key-value pairs**.

**Core invariant:** If a HSM accepts (key, value), it will not accept (key, value’) where value != value’.

**Application:** scalable-rate limiting
Each key is a username, and each value commits to a recovery attempt.

Maintains log
Each HSM stores log digest
Distributed log implementation

Log structured as a binary search tree

• ordered by key (username), where

• leaves are values (commitments to recovery attempts).

Alice    Charlie
**Core invariant:** If a HSM accepts (key, value), it will not accept (key, value’) where value != value’.
Updating the log

Core invariant: If a HSM accepts (key, value), it will not accept (key, value’) where value != value’.
Updating the log

Periodically, service provider

• computes a Merkle tree over the leaves, and
• sends a new Merkle root to the HSMs.

HSMs must check that the new log head extends the old log head.
Updating log digest at the HSMs with distributed auditing

Data center

Old log $L$, new log $L'$

HSM

Old digest $d$, new digest $d'$
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Old digest $d$, new digest $d'$
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Old digest $d$, new digest $d'$

Request $\lambda \approx 128$ chunks
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Request $\lambda \approx 128$ chunks

Old digest $d$, new digest $d'$

If each requested chunk keeps $\leq 1$ recovery attempt per user, sign new digest $d'$. 
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Request $\lambda \approx 128$ chunks

If each requested chunk keeps $\leq 1$ recovery attempt per user, sign new digest $d'$.
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Request $\lambda \approx 128$ chunks

Signature

Old digest $d$, new digest $d'$

If each requested chunk keeps $\leq 1$ recovery attempt per user, sign new digest $d'$. 

Aggregate signatures from all HSMs [BGLS03].
Updating log digest at the HSMs with distributed auditing

Old log \(L\), new log \(L'\)

Divide updates into \(N\) chunks.

Request \(\lambda \approx 128\) chunks

If each requested chunk keeps \(\leq 1\) recovery attempt per user, sign new digest \(d'\).

Aggregate signatures from all HSMs [BGLS03].
Updating log digest at the HSMs with distributed auditing

Old log $L$, new log $L'$

Divide updates into $N$ chunks.

Old digest $d$, new digest $d'$

Request $\lambda \approx 128$ chunks

If each requested chunk keeps $\leq 1$ recovery attempt per user, sign new digest $d'$.

Aggregate signatures from all HSMs [BGLS03].

Signature

If signature verifies, accept new digest $d'$. 

Aggregate signature
Distributed auditing properties

- **Scalability:** Each HSM checks $\lambda \approx 128$ chunks and verifies one signature.

- **Security:** The attacker corrupts the log undetected with probability $2^{-128}$.

- **Transparency:**
  - **Clients** can monitor recovery attempts made on their behalf.
  - **External auditors** can audit the log.
    - An attacker that later compromises all HSMs can’t erase evidence.
Making SafetyPin practical for resource-limited hardware (see paper)

At recovery, the attacker sees which HSMs to compromise to obtain backup.

• **Challenge:** Revoking ability to decrypt requires large HSM secret keys

• **Idea:** Outsourced storage with secure deletion
Outline

1. SafetyPin design
   - Overview
   - Scalable rate-limiting

2. SafetyPin evaluation
Implementation setup

https://github.com/edauterman/SafetyPin

Android Pixel 4

Linux machine

100 $20 SoloKeys

100 SoloKeys = slice of cluster that can process 1B recoveries/year
Implementation setup

https://github.com/edauterman/SafetyPin

100 SoloKeys = slice of cluster that can process 1B recoveries/year
Evaluation summary: experimental results

Overhead with resource-limited HSMs

• Recovering a backup takes 1.01s.

Client cost

• Generating a recovery ciphertext takes 0.37s.
• Client must download 2MB of HSM keys each day.
Evaluation summary: system estimates

Total system cost

- Supporting 1B recoveries per year costs
  - $61K with SoloKeys (hardware we used), and
  - $15M with high-quality HSMs.
- Cost of storing 4GB for 1B users: $600M.
Secure hardware doesn't need to be trusted hardware

- Today, secure hardware is often a single point of failure.
  - This doesn’t have to be the case.

- We should never settle for reduced security.
  - Computational limits of HSMs are not an excuse.
Secure hardware doesn't need to be trusted hardware

• Today, secure hardware is often a single point of failure.
  • This doesn’t have to be the case.

• We should never settle for reduced security.
  • Computational limits of HSMs are not an excuse.

Thanks!

Emma Dauterman
edauterman@berkeley.edu
https://github.com/edauterman/SafetyPin