Secret Key Recovery in a Global-Scale End-to-End Encryption System

Graeme Connell, *Signal Messenger;* Vivian Fang, *UC Berkeley;* Rolfe Schmidt, *Signal Messenger;* Emma Dauterman and Raluca Ada Popa, *UC Berkeley*

https://www.usenix.org/conference/osdi24/presentation/connell

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of Science and Technology

Secret Key Recovery in a Global-Scale End-to-End Encryption System

Graeme Connell[*]
Signal Messenger

Vivian Fang[*]
UC Berkeley

Rolfe Schmidt[*]
Signal Messenger

Emma Dauterman
UC Berkeley

Raluca Ada Popa
UC Berkeley

Abstract

End-to-end encrypted messaging applications ensure that an attacker cannot read a user's message history without their decryption keys. While this provides strong privacy, it creates a usability problem: if a user loses their devices and cannot access their decryption keys, they can no longer access their message history. To solve this usability problem, users should be able to back up their decryption keys with the messaging provider. For privacy, the provider should not have access to users' decryption keys. To solve this problem, we present Secure Value Recovery 3 (SVR3), a secret key recovery system that distributes trust across different types of hardware enclaves run by different cloud providers in order to protect users' decryption keys. SVR3 is the first deployed secret key recovery system to split trust across heterogeneous enclaves managed by different cloud providers: this design ensures that a single type of enclave does not become a central point of attack. SVR3 protects decryption keys via rollback protection and fault tolerance techniques tailored to the enclaves' security guarantees. SVR3 costs $0.0025/user/year and takes 365ms for a user to recover their key, which is a rare operation. A part of SVR3 has been rolled out to millions of real users in a deployment with capacity for over 500 million users, demonstrating the ability to operate at scale.

1 Introduction

End-to-end encrypted messaging applications like Signal [85], WhatsApp [24], and Messenger [58] are used by hundreds of millions to billions of users. They provide end-to-end encryption: user devices (the "ends") encrypt user messages so application servers receive only encrypted messages without decryption keys. Only the users in a conversation can decrypt the messages locally on their devices. This paradigm protects user messages even if the application provider or cloud infrastructure is compromised.

To provide this guarantee, end-to-end encrypted messaging application providers must ensure that their users' secret keys are protected against a wide range of attacks by malicious employees, cloud provider administrators, or other privileged agents. Unfortunately, this creates a usability problem: if a user loses their secret keys, for example by losing their devices, the user loses access to their account and message history because these keys are necessary to decrypt the user's chat history and metadata (e.g., address book, social graph). The application provider cannot directly store user secret keys because it could then decrypt user messages, violating the core principle of end-to-end encryption. Therefore, users who lose their devices should be able to recover their secret keys without the provider getting access to their secret keys.

Shortcomings of many existing key recovery systems. A potential strawman is to allow the user to download their secret keys (e.g., print them on a piece of paper) and store them in a safe place [40, 46, 59], but this places extra burden on the user [76]. A more user-friendly approach to this problem is to allow a user to use a password or a PIN to encrypt their key [33]. Unfortunately, these are often vulnerable to brute-force dictionary attacks [82, 83]. Furthermore, standard safeguards (e.g., forcing the attack to be performed online) can easily be circumvented by the application provider.

Current deployed systems [4, 43, 51, 88, 96, 98] prevent brute-force attacks by using secure hardware to limit the number of PIN guesses. This approach provides a strong protection against service provider administrators and cloud providers. While these systems all represent significant advances in password-based key recovery, they rely on the security guarantees of a *single* type of secure hardware. Although secure hardware is a powerful tool for enhancing the security of systems, it can eventually be subverted—attackers have extracted user secrets from secure hardware in the past [12, 14, 31, 35, 62, 75, 79, 87, 90, 91, 94, 95]. In these systems, compromising just one type of secure hardware enables an attacker to recover many users' secret keys, which is a catastrophic scenario for any popular encrypted system.

[*] Equal contribution.

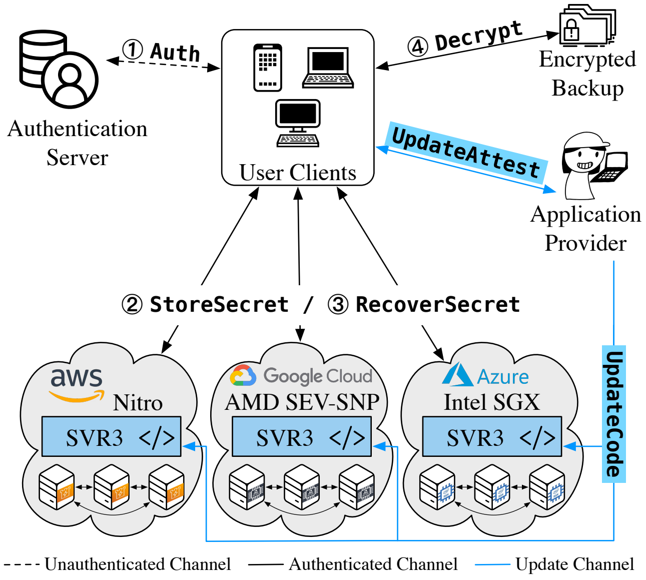


Figure 1: System architecture for $n = 3$ enclave clusters, with each cluster using a different type of hardware enclave.

Key recovery without a single point of security failure. In this paper, we contribute **S**ecure **V**alue **R**ecovery **3**[1], a PIN-based secret key recovery system that prevents any one type of enclave or cloud provider from becoming a central point of attack. Our security properties are informed by the observation that many vulnerabilities are quickly patched, and so it is challenging for an attacker to find vulnerabilities *simultaneously* on different enclave architectures. SVR3 proposes a layered architecture, illustrated in Figure 1, consisting of a tailored cryptographic multi-server key recovery protocol that distributes trust across three different enclaves from three distinct hardware vendors on three major clouds: Intel SGX in Microsoft Azure, AMD SEV-SNP in Google Cloud, and Nitro in AWS. SVR3 ensures that even if an attacker simultaneously compromises two of these enclave types and the respective clouds, the attacker cannot reconstruct the user's secrets due to the cryptographic protocol. The attacker needs to simultaneously compromise the security of all of the clouds and all of the enclave types to reach user secrets.

We implemented SVR3 as a production-ready system embedded in Signal Messenger [85], an end-to-end encrypted messaging application with tens of millions of users. We have already deployed an initial version of SVR3's implementation to millions of users globally, and the fully featured system is in the process of deployment at the time of publication. A third-party auditor, NCC Group, audited the deployment of Signal's SVR2, a predecessor system currently in production and using SVR3's consensus protocol on a single trust domain. SVR3 is open source [84] and can be used by any end-to-end encrypted system that needs secret key recovery (e.g., encrypted messaging [24, 85], email [72, 74], or storage [99]). To the best of our knowledge, SVR3 is the first deployed cross-enclave, cross-cloud secret key recovery system. The servers for SVR3 cost only \$0.0025/user/year and it takes 365ms for a user to recover their key, which is a rare operation.

Design decisions. Our design choices were guided by the goal of developing a real-world PIN-based key recovery system that prevents dictionary attacks, is easy and affordable to maintain, and provides security even if a particular enclave or cloud provider is vulnerable. We summarize the key decisions below.

A layered security architecture (§2–§3). We aim to protect users' secrets against three major classes of attackers: cloud attackers, an internal application provider attacker, and external hackers. To achieve this, one strawman is to distribute trust across multiple organizations. However, finding reliable and trustworthy such organizations is difficult and expensive [21, 50]. Instead, we introduce an architecture that layers cryptographic security on top of hardware security by using different types of enclaves in different clouds. The hardware enclaves enable creating three separate trust domains, and the cryptographic tools split secret keys across the trust domains.

PPSS to distribute trust (§4). Password Protected Secret Sharing (PPSS) [5] provides password-based key recovery while distributing trust across multiple backends and limiting attackers to online dictionary attacks. Different PPSS schemes have different deployment consequences, and we select the construction by Jarecki et al. [37] primarily because it requires no cross-trust domain communication and the server design enables clients to use different secret sharing schemes if they wish. We use this protocol to construct our one-round key recovery protocol, where the servers receive no information about whether the PIN guess was correct, and the servers unconditionally delete key material after a fixed number of PIN guesses (which can be refreshed by the clients). This is in contrast to existing works [85, 88, 98], which rely on password-based authentication and require multiple communication rounds.

Rollback protection through enclave memory and consensus (§5). Like Signal's original SVR1 system [85], SVR3 protects against *software* rollback attacks by keeping all data (e.g., guess counts) inside enclave memory. In order to prevent data loss, we replicate data across multiple enclaves in the same cloud. SVR1 uses the original Raft consensus protocol [66], which is not safe under *physical* rollback attacks. In principle, an attacker with physical access (e.g., a DIMM interposer [89]) to a single server in a vanilla Raft replica group could take control of the group and roll back log entries. To defend against such attacks, we develop a modified Raft [66] protocol, Raft$^{\circlearrowleft}$, that provides safety under physical rollback attacks, as specified in §3.2. We prove its safety under a formal TLA+ [45] model in the face of physical rollback attacks.

[1] This is the third generation of Signal's SVR service and succeeds SVR1 [51], which did not distribute trust across multiple types of secure hardware. (SVR2 was a transition system consisting of a partial SVR3 design.)

Secure code updates via auditing (§6). To enable code updates while providing strong security, we allow clients to audit the deployed code and explicitly disallow sharing of data between different (server) binary versions. Data migration between binary versions flows through the client, and clients can determine whether or not to store their secret value on each version of the binary.

Limitations. SVR3 relies on the underlying security guarantees of the enclaves it employs; supporting a new enclave or a new version of an existing enclave would require carefully reasoning about how it fits into the threat model. Splitting infrastructure across multiple cloud providers also incurs higher monetary costs than deploying on a single provider, but offers stronger security assurances. Finally, SVR3 does not support recovering the user PIN that is used in secret key recovery (i.e., if a user forgets their PIN, they cannot recover their key). We mitigate this in practice by periodically prompting the user to re-enter their PIN on the messaging client to prevent permanent lockout.

2 System overview

2.1 System architecture

Figure 1 shows the system architecture for an SVR3 deployment with three cloud providers, with the following entities:

Enclave clusters. The application owner deploys n enclave clusters (in our deployment, $n = 3$). To strengthen security, each enclave cluster should run on a different type of enclave in a different cloud environment (see §3). We will refer to each enclave cluster running on different hardware in a different cloud as a trust domain. Enclave clusters maintain replicated storage and respond to messages from clients. Each enclave cluster consists of a load balancer, a discovery service, and a geographically distributed replica group.

Authentication server. The authentication server establishes authenticated channels between clients and enclave clusters. The authentication server prevents malicious clients from exhausting PIN attempts for honest users because a client needs to authenticate to the authentication server (e.g., via an SMS code) before interacting with the enclave clusters.

Clients. Clients (e.g., mobile phones or laptops) interact with the authentication server and nodes in the enclave clusters in order to back up and recover their secret keys.

Application provider. The application provider will update the software and run monitoring and maintenance to ensure that the system is available and healthy.

2.2 System API

As shown in Figure 1, SVR3 exposes the following client API:

- Auth(client_id, client_cred) → auth_token: Establishes authenticated channel between client and server.

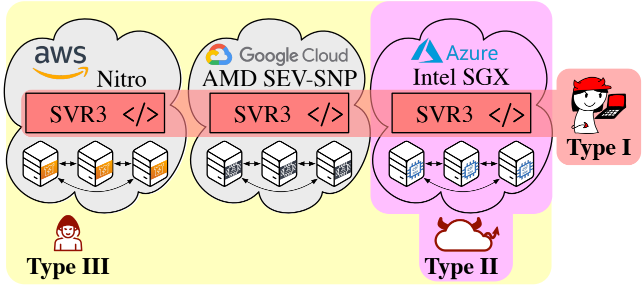


Figure 2: Types of attackers SVR3 protects against.

- StoreSecret(client_id, auth_token, val, pin): Backs up a value val for an authenticated client using a human-memorable PIN value pin and an authentication token auth_token.
- RecoverSecret(client_id, auth_token, pin) → {secret, ⊥}: Recovers the value secret for client if (and only if)
  - auth_token is valid for client_id,
  - pin matches the PIN provided at StoreSecret time for client_id, and
  - the number of unsuccessful RecoverSecret attempts for client_id does not exceed a set guess limit.

  Otherwise, outputs ⊥.

The client can use their recovered secret to locate, authorize access to, and decrypt their encrypted backup.

We describe how the developer updates SVR3 in §6.

3 Threat model and guarantees

SVR3's goal is to protect users's secrets. SVR3 provides different security guarantees against three types of server attackers, shown in Figure 2:

- **Type I (Internal).** This attacker compromises the organization deploying SVR3 (e.g., a malicious employee). This attacker does not have physical access to the cloud deployment, but can freely spin up and bring down machines and modify the software being run.
- **Type II (Cloud).** This attacker represents an entity with control over the physical infrastructure SVR3 is deployed on (e.g., a single cloud provider). While this attacker does not have the same degree of access to the entire multi-cloud system deployment, it can leverage physical access and tamper with the hardware running SVR3.
- **Type III (External).** This attacker is external to the deployment of SVR3 (e.g., a hacker), and attacks all parts of an organization's surface.

We express SVR3's security guarantees at two levels: (1) at the level of trust domains (§3.1), defining security in terms of which trust domains are not compromised, and (2) at the level of enclaves inside a trust domain (§3.2), specifying the conditions under which a trust domain is not compromised.

Like other end-to-end encrypted systems [72,74,98], if a user's device is compromised, SVR3 provides no guarantees to that user. For an uncompromised user device, we rely on the trustworthiness of client code released by Signal; we enable the community to scrutinize the client code and build trust in it by making it open-source [55–57].

SVR3 does not hide the identity of clients or the timing of backup and recovery requests.

3.1 Security across trust domains

SVR3 protects users' secret keys if at most t out of n trust domains are compromised. We assume that the odds of an attacker identifying and exploiting vulnerabilities *simultaneously* across $> t$ trust domains is low, which motivates our threat model. By simultaneous, we mean within the time period it takes to become aware of a vulnerability and replace the enclaves in the trust domain impacted by that vulnerability.

In our deployment of SVR3, we set $t = 2$ and $n = 3$, so we ensure security as long as at least one trust domain is not compromised. We limit PIN guesses by selecting a parameter u, a server *usage limit*.

Theorem 1 (Informal)**.** *In an SVR3 deployment configured with n trust domains, threshold t, and a usage limit u, assuming a password-protected secret sharing scheme (defined in §4.2), if an attacker compromises $\leq t$ trust domains, then SVR3 ensures that, for each secret key, the attacker only has $\left\lfloor \frac{nu}{t+1} \right\rfloor$ PIN attempts and, after that, cannot recover the secret key.*

We describe how SVR3 achieves Theorem 1 in §4.2.

3.2 Security within a trust domain

We now describe the threat model we consider when instantiating the trust domains assumed in §3.1. Recall that each trust domain consists of an enclave cluster and that each trust domain should use a different type of enclave.

3.2.1 Enclave threat model

SVR3's design is not tied to some specific enclave implementations. Different enclaves vary in design, so we abstract out the security properties that we require from the enclaves employed for SVR3's security guarantees (§3.2.2) to hold. An *uncompromised enclave* must provide:

(E1) *Application-level attestation*. The enclave can prove that certain code is running before other systems interact with it.

(E2) *Access control*. Enclave memory is encrypted, and access control is hardware-enforced to prevent all non-enclave access.

(E3) *Page-level rollback granularity*. The attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating global memory integrity. We assume that an attacker cannot mount these attacks at a sub-page granularity (e.g., address level) either because the enclave protects this or other protection mechanisms are used in the enclave (see below).

Deviations from enclave threat model. We describe the properties of different enclaves and how they fit our threat model in §A of the full version [17]. Some recent enclaves use AES-XTS, which encrypts in 16B increments [15]. While our design currently targets enclaves that can only be rolled back at the page-level granularity (E3), we can implement atomic regions (regions that are guaranteed to run without interruption by an attacker) by utilizing the interrupt handler introduced by AEX-Notify [18]. We describe how to do so in §5.3. Given the changing landscape of enclave implementations and the possibility that enclaves may not adhere to (E1)–(E3) in the future, we assume that alternative mechanisms like AEX-Notify can be developed to address such discrepancies between real-world enclaves and our enclave threat model.

Attacks on enclaves. Enclaves are susceptible to attacks. We list four categories here and then discuss when SVR3 hardens a trust domain against them.

(A1) *Memory access pattern attacks*. Enclaves do not hide memory access patterns, enabling a large class of side-channel attacks, including but not limited to cache attacks [9,32,61,80], branch prediction [48], paging-based attacks [93, 100], and memory bus snooping [47].

(A2) *Software rollback attacks*. Enclaves are also susceptible to rollback attacks, also referred to as freshness or replay attacks [69]. Software rollback attacks occur from rolling back persisted state outside of the enclave's memory (**Type I** attacker).

(A3) *Hardware rollback attacks*. An attacker with physical access to the system bus can roll back enclave memory at the page level without detection (**Type II** attacker), for example, using a DIMM interposer [89].

(A4) *Other attacks*. Certain physical attacks allow an attacker to break guarantees (E1)–(E3) of enclaves (e.g., leakage due to power consumption [14, 62, 87] or denial-of-service attacks due to memory corruptions [31, 35]). Transient execution attacks [12, 75, 79, 90, 91, 94, 95] exploit speculative execution to leak secret data.

3.2.2 Security guarantees

SVR3 *hardens* a trust domain against a set of attacks, rendering the trust domain uncompromised despite those attacks. We describe the conditions below:

(H1) SVR3's memory-access patterns do not depend on user secret content, and hiding *which* user is recovering their key is a non-goal for SVR3, so it does not suffer from memory-access patterns side-channel attacks (A1).

(H2) SVR3 defends against software rollback attacks (A2).

(H3) SVR3 defends against hardware rollback attacks (A3) as long as $\leq s$ nodes in each cluster are simultaneously rolled back, where s is a fault-tolerance ("supermajority") parameter defined in §5.2.5. In our production deployment, we set $s = 2$.

(H4) Within a trust domain, SVR3 does not guarantee protection against other attacks (A4), which could render the trust domain compromised. In this case, SVR3 still offers the cross-trust domain security guarantees in §3.1.

3.3 Availability

Like other end-to-end encrypted systems [72, 98], Signal prioritizes security over availability of secret key recovery because users' secret keys are extremely sensitive and crucial to safeguard in an end-to-end encrypted system. Nevertheless, SVR3 provides availability to clients when at least $t+1$ trust domains are operating correctly. By correct operation, we mean that enclaves in the trust domain are online and none of the enclaves in the trust domain are under attack. Therefore, we expect the system to be available under normal operation.

SVR3 also does not defend against denial-of-service (DoS) attacks from a **Type I** attacker (since this is the organization that deploys SVR3 itself) or the authentication server.

SVR3 ensures that a malicious client cannot deny availability for an honest user (e.g., by exhausting the number of PIN attempts allowed) assuming that the attacker did not compromise the client credentials or the authentication server (used to Auth in Figure 1), and it did not otherwise compromise the servers beyond the availability threshold above.

It is important to consider what users would experience if trust domain(s) were to fail, leading to secret value loss. While this is a significant event when viewed from the perspective of the application provider, it will not lead to secret value loss for the majority of clients in practice: clients cache their SVR3-protected secret, and so clients can simply create a backup at the new deployment. Thus data loss is only a concern for users who lose their devices after the old deployment fails and before migration to the new deployment completes.

4 Secret key backup and recovery protocols

We now describe the cryptographic protocols in SVR3.

4.1 Establishing enclave sessions

To interact with the SVR3 servers, the client must first authenticate with the authentication server. If the user has lost their devices, then the authentication server sends the client an SMS code, and then the user enters the SMS code to receive a token. This process allows the authentication server to prevent malicious clients from denying service to honest users by exhausting all of their PIN attempts. Notably though, the authentication server does not have any information about user PINs. The client then uses this token to establish a secure channel with a replica in each trust domain. As part of the process of establishing a secure channel, the client runs remote attestation [16] with the enclaves to ensure that it is communicating with the expected enclaves.

4.2 PIN-protected secret sharing

In existing deployed PIN-based backup systems [43,51,96,98], a secure hardware device has access to users' secret keys and PINs or PIN-derived information in order to authenticate users. This design means that an attacker that compromises the secure hardware can, either directly or via a brute-force attack, learn user PINs. This property is particularly problematic when we consider the fact that many users re-use PINs across services.

As a result, when designing our cross-enclave cross-cloud solution, we cannot simply instantiate the above mechanism in each trust domain. Any one compromised trust domain would have access to the PIN, enabling the attacker to recover the user's secret key. Instead, we leverage the class of cryptographic protocols called *password-protected secret sharing (PPSS)* [5] protocols, which ensure that:

- An attacker that compromises $\leq t$ trust domains is still limited to an online dictionary attack.
- If an attacker fully compromises $> t$ trust domains, the attacker does not immediately learn client secrets. The attacker still must perform an offline dictionary attack on user PINs.

Identifying a suitable PPSS scheme for SVR3. Different PPSS schemes have different tradeoffs [1, 5, 36–38], so we worked to identify the most suitable scheme for SVR3 and then tailor it to our setting. Some prior work optimizes for metrics that are not important to our deployment, but sacrifices properties that are important to us.

For example, many of these works aim to reduce the number of exponentiations to improve efficiency [1, 36–38]. However, the number of exponentiations is not a bottleneck in our setting, especially because the number of trust domains (3) is small. The scheme with the fewest exponentiations [38] also requires coordinated server initialization and necessitates choosing secret sharing parameters at deployment time. Coordinated initialization could require us to redeploy all trust domains every time a single trust domain requires a security upgrade, and cross-trust-domain communication with security against **Type I** attackers is difficult. Choosing a secret sharing scheme at deployment time tightly couples PPSS parameters with clients and servers, removing the flexibility to modify client PPSS parameters without also changing the servers.

With these priorities in mind, we identified the PPSS from Jarecki et al. [37] as the most suitable because it is particularly simple: each backend generates a new secret key for a client when the client creates a new backup and then uses this key to evaluate an oblivious pseudorandom function (OPRF) [28] during secret reconstruction. Informally, a pseudorandom function (PRF) is a keyed function $F_k(\cdot)$ that, for a randomly chosen key k, appears to be random (indistinguishable from a function chosen uniformly at random from all functions with the same domain and range), even though it is deterministic and efficiently computable. An *oblivious* PRF is a two-party protocol where the server holds k and the client holds some input x. The protocol enables the client to learn $F_k(x)$ without the server learning anything about x or $F_k(x)$.

This PPSS scheme has several properties that are appealing for a real-world deployment:

- The protocol is one-round and concretely efficient.
- Different trust domains do not communicate with each other.
- Servers need minimal configuration. In particular they do not need any information about the threshold scheme being used, and different clients can use the same server with different threshold schemes.
- The protocol can use a standards-track OPRF with optional verifiability [23].

We note that the WhatsApp key recovery system uses a password-authenticated key agreement (PAKE) scheme [24, 98], and SVR3 does not. While PAKE protocols are a commonly cited application for PPSS schemes, we do not need to establish a session between our client and a server. We only need to recover a secret key, which is a simpler problem. Since branching while fetching secret shares is not sensitive, we do not need to layer oblivious data retrieval on top [22, 60].

Augmenting PPSS with usage limiting. Limiting attackers to a fixed number of password guesses is a core requirement for SVR3. While the application provider can use an authentication server for access control and rate limiting, this only restricts external users. SVR3 must limit powerful attackers with full administrative and physical access to the servers to the same finite number of guesses.

We solve this by leveraging our distributed-trust setting to enforce a *usage quota* on OPRF evaluations. A standard OPRF [28] allows a server with a PRF key to evaluate a PRF on a client input without learning the input. SVR3 allows the client to set a usage limit, u, at registration time, and each honest trust domain will delete that client's OPRF key after u OPRF evaluations. In order to instantiate an honest trust domain, we use enclaves to ensure that the server enforces the usage limit. Note that the security guarantees provided by PPSS and the heterogeneous enclaves are tightly coupled: the enclaves are critical for instantiating trust domains, and PPSS enables splitting a secret value across different trust domains.

In the below proposition, we bound the number of total OPRF evaluations based on the threshold t and trust domains n, providing the protection described in Theorem 1.

Proposition 1. *For a (t,n) instance of PPSS [37] with a usage-limited OPRF configured to allow u evaluations, an adversary has at most $\lfloor \frac{nu}{t+1} \rfloor$ PIN attempts before the secret cannot be recovered.*

Proof. Only nu OPRF evaluations are possible in the system. $t+1$ evaluations are needed to perform one PIN attempt. After $\lfloor \frac{nu}{t+1} \rfloor$ PIN attempts, $(t+1)\lfloor \frac{nu}{t+1} \rfloor$ OPRF evaluations have been used. Only $(t+1)\{nu/(t+1)\} < t+1$ more evaluations are possible, where $\{\}$ denotes the fractional part, that is, $\{x\} = x - \lfloor x \rfloor$. This is not enough to reconstruct the secret. □

5 Building a SVR3 backend

We now describe SVR3's system design within one trust domain. Per our threat model in §3, each uncompromised SVR3 trust domain consists of a cluster of machines, which we assume behave correctly except for possible physical rollback attacks and crash failures within a specified bound.

5.1 Design decisions

We first provide an overview of the design decisions behind SVR3's design to ensure fault tolerance and the security guarantees in §3.2.2.

Use of enclaves. In order to protect server secrets and allow clients to check the code that is processing their data, we run the core part of the service in an attested, confidential enclave.

In-memory database to avoid sealing. Data sealing is a mechanism whereby an enclave can encrypt internal state with a key that is unique to the platform and enclave, persist the encrypted data to disk, and then recover it if the enclave is torn down and restarted. As noted in prior work [26, 97], applications in commercially available enclaves that use data sealing to store state externally and recover from crashes are vulnerable to simple, software-based rollback attacks. Since a core function of SVR3 is to faithfully maintain a per-user OPRF evaluation count, rollback attacks would undermine the system and could allow an attacker unlimited online password guesses. To prevent this and achieve (H2), the enclave that stores the database of client secrets and usage counters is kept entirely in enclave-protected memory; it is *never* sealed and written to untrusted memory or disk. We show that the database fits entirely in memory without sharding users in §8.1.

Distributed consensus. Without a data persistence mechanism (e.g., data sealing), the servers cannot recover from crashes, and data in any failed server will be lost. To ensure that data is not lost, we build the service as a geographically distributed database. To ensure split-brain or other attacks do not allow excess PIN guesses, we use a distributed consensus protocol, modified from Raft [66]. We give a high-level overview of vanilla Raft in §5.2.1. Our modified Raft protocol, $\text{Raft}^{\circlearrowleft}$, which we describe in §5.2.3, hardens vanilla Raft against physical rollback attacks and ensures that client requests and usage count changes are committed before responding to client queries. We describe in §5.3 how we use Raft$^{\circlearrowleft}$ to achieve global integrity across the database when assuming page-level rollback granularity of enclaves (E3), achieving (H3).

5.2 Rollback-resistant consensus protocol

SVR3 already protects against the class of rollback attacks that arise from storing state outside of the enclave by keeping all state in memory. However, as discussed, machines can fail, and so in order to tolerate failures without losing data, we use Raft$^{\circlearrowleft}$, a modified version of vanilla Raft across enclaves from a cloud provider. A full TLA+ description of Raft$^{\circlearrowleft}$ is available in §E of the full version [17], and we provide a proof of safety based on the TLA+ specification in §D.

In this paper, we use n to refer to the number of trust domains and m to refer to the number of replica machines *within* a trust domain.

5.2.1 Vanilla Raft background

Raft [66] is a consensus algorithm that manages a replicated log across multiple nodes (replicas). It elects a single leader replica that receives and replicates log entries to the other follower replicas. The leader handles all client requests by appending new log entries and sending an AppendEntriesRequest to each follower for the duration of its *term*. Follower replicas respond to requests from the leader to replicate log entries. If the leader fails, a new leader is elected through a leader election process. Log entries are identified by `<index, term>`, where `index` is the log position and `term` is the current term number. There is at most one leader in any given term. A leader forces the followers' logs to duplicate its own: conflicting entries in follower logs (with some `term` t) will be overwritten with entries from the leader's log if the leader's term t' is $\geq t$. For f crash failures, Vanilla Raft requires $m \geq 2f+1$ replicas in order to provide safety and liveness.

5.2.2 The physical rollback problem

While keeping the database in memory protects against software rollback attacks, an attacker with physical access to the system bus could roll back enclave memory at the page level. Since such an attack is more expensive to perform than software-based rollback attacks, we can significantly improve security by requiring an attacker to perform these attacks simultaneously on multiple enclave replicas. With this context, we note that the vanilla Raft protocol [66], as specified, will allow an attacker who can roll back a Raft leader to make an unlimited number of PIN attempts: the Raft protocol does not look at log contents, so if a leader is rolled back and sends an AppendEntriesRequest for a new `<index, term>` log entry at an old log index, followers will accept it and allow the leader to commit.

Prior work [26, 97] has addressed a problem close to this one, but with important differences. First, they are designed for data-sealing rollbacks, which do not affect SVR3 because we do not use data sealing. Second, Raft$^{\circlearrowleft}$ also defends against physical rollback attacks, which prior works do not consider in their threat model. Physical rollback attacks are more difficult to detect than data-sealing rollback attacks: after a crash recovery, the new enclave has to execute code that decrypts the sealed data to rebuild the internal state and every data-sealing rollback needs to have the enclave go through this code path. The RR protocol [26] takes advantage of this process to detect data-sealing rollback attacks. Finally, existing protocols aim to ensure liveness in the face of rollback attacks, and this is an explicit non-goal for SVR3 as mentioned in §3.3.

5.2.3 Rollback prevention in Raft$^{\circlearrowleft}$

Together, the following additions to the Raft protocol enable us to prove safety of Raft$^{\circlearrowleft}$ in the presence of an attacker who can simultaneously mount physical rollback attacks against $\leq s$ nodes. For m Raft$^{\circlearrowleft}$ servers in a trust domain, s must be strictly smaller than m to ensure safety (§5.2.4). However, to ensure fault tolerance and liveness in the face of crash failures, s should be even smaller (§5.2.5).

Hash chain. Instead of using `<index, term>` to identify a log entry, as in Raft, we use `<index, term,` $\texttt{hash}_{\texttt{index}}$`>` where $\texttt{hash}_{\texttt{index}} = \texttt{Hash(entrydata, index, term, hash}_{\texttt{index-1}})$, `entrydata` is the contents of the log entry, and `Hash` is a cryptographic hash function. When a follower receives an AppendEntriesRequest, it computes the expected hash chain value and verifies that it matches the value in the request. If the values do not match, the follower rejects the request.

This prevents the simple rollback attack on Raft described in §5.2.1. However, it is still possible for an attacker who can roll back one server to gain unlimited password guesses by triggering an election with a quorum of servers that did not see the log entry for the first client request.

Supermajority. To ensure that an attacker capable of rolling back a single server cannot gain extra password guesses by triggering an election, we require quorums to have a supermajority of replicas so that the intersection of any two quorums contains more than s replicas, where s is a configurable parameter that is included in the server's attestation. This allows clients to be certain of the value of s used by the service and decide whether to accept it. We prove that an attacker must be able to roll back more than s enclaves to roll back a log entry that was committed by this Raft$^{\circlearrowleft}$. This supermajority parameter is comparable to PBFT's Byzantine nodes value [10].

Promise round. We add a *promise round* to the protocol. We discuss reasoning for why we add a promise round in the full version [17]. Once a quorum of servers acknowledges seeing a log entry, the leader will "promise" this entry by advancing its `promise_idx` to the index of this entry. A promised entry is not committed, but no replica will delete an entry that has been promised. This completes the first round.

The leader now sends its `promise_idx` to all followers in its next AppendEntriesRequest, and followers will update their own `promise_idx` to match the leader's when they process the message. From this point, these followers have promised the log entry and will not delete it. The followers send their current `promise_idx` with each AppendEntriesResponse. Once a quorum of replicas has promised an entry, it is committed.

5.2.4 Safety

In order to achieve safety, the number of machines in the enclave cluster must be larger than the number of rollback attacks we want to tolerate ($m > s$). As liveness under rollback attacks is a non-goal for SVR3 (an attacker with physical access can easily deny service), we decouple the constraints on m with respect to rollback attacks (s) and crash failures (f_c). We describe how s impacts liveness under crash failures in §5.2.5. We prove that Raft$^{\circlearrowleft}$ is safe under a bounded number (s) of physical rollback attacks within a trust domain.

Theorem 2 (Informal)**.** *Let M_R be the maximum number of machines in an enclave cluster that can be rolled back and s be our supermajority configuration parameter. If $M_R \leq s$, then under standard cryptographic assumptions, for every log entry* <*index, term, hash*$_{index}$> *that has been applied to the state machine of a server i, server i will never apply a different log entry at this index.*

Proof sketch. The argument follows the proof of safety in Ongaro [65] and relies on the observation that any two quorums will have an intersection that includes at least one server that has not been rolled back. We must address the fact that in the presence of rollbacks, Lemma 3 in Ongaro [65] does not hold. This poses a significant challenge, and forces us to introduce a new concept of *live committed* entries that is subtly different from the prior notion of committed [65]. With our definition, future leaders may not have a live committed entry in their log, but if they do not then they will be unable to commit new entries, so we retain safety at the expense of liveness. The major point where the argument from Ongaro [65] breaks down in our setting is in points 7.c.ii.B and 7.c.iii.B in the proof of their Lemma 8. Our argument uses the hash chain and promise index to show that there is a voter in the intersection of two quorums that has not been rolled back and will not replace the log entry. The complete proof of safety is in §D of the full version [17].

5.2.5 Liveness

We do not provide liveness for a trust domain under the setting of an attacker mounting physical rollback attacks, as the attacker could trivially deny client requests by taking the entire enclave cluster offline. When assuming no attacks within a trust domain, Raft$^{\circlearrowleft}$ requires $f_c \leq \lfloor (m-s)/2 \rfloor$ *crash* failures to be live under normal connectivity conditions, where m denotes the number of replicas in a trust domain (enclave cluster) and s denotes the supermajority parameter described in §5.2.3. This is due to the quorum size being $\lfloor (m+s)/2 \rfloor + 1$ enclaves. It remains an open problem to prove liveness of Raft in this setting (e.g., by formal verification [34]). Nevertheless, as discussed in §3.3, SVR3 still provides availability to clients when at least $t+1$ trust domains are operating correctly.

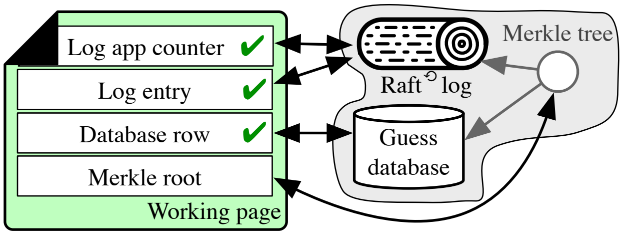


Figure 3: Integrity across database. In order to achieve global integrity, updates are only applied when all state on the working page validates under the same Merkle tree root.

5.2.6 Self-healing for simple maintenance

We implement the process for replica group membership changes described in the Raft paper [65] and add a layer of automation. In Raft$^{\circlearrowleft}$, a replica group has a configured target number of voting members. For a healthy configuration, a replica group in our system will have this number of voting members as well as several non-voting members that stay up to date and service client requests. If some voting member is not seen by the leader after a configurable timeout, the leader will initiate a membership change that demotes the missing replica to non-voting status. After an additional timeout, it will remove the replica from the group entirely.

Furthermore, whenever the number of voting members is below the configured target, the leader will check to see if a non-voting member is present and initiate a membership change promoting a non-voting member to voting status.

With these mechanisms in place, administrators simply need to launch new instances and direct them to the discovery service with group information. The new server will then request to join the group, be brought up to date by a peer, and become a non-voting member. As needed, the voting members may then promote this new replica to voting status.

5.3 Integrity across the database

Raft$^{\circlearrowleft}$ provides protection against rollback attacks on the contents of the log. However, our threat model (§3) assumes *page-level* rollback granularity on memory inside the enclave, which means that the attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating *global* memory integrity.

In order to protect against rollback attacks on the backing in-memory database, SVR3 keeps a Merkle tree across the Raft$^{\circlearrowleft}$ log, database, and log application counter.

5.3.1 Merkle tree

The log application counter keeps track of the latest log entry that has been applied to the database. The Merkle tree contains every database row, the hashchain of the most recently committed log entry, and the log application counter. The hashchain of the last committed log entry, as described in §5.2.3, can be used to verify this entry and earlier entries in the log. As shown in Figure 3, the Merkle leaves for database rows and log application counter are updated each time the underlying object changes, and the update only succeeds if the current state of the Merkle tree is consistent with the previous value of that data.

5.3.2 Applying committed log entries

We describe how we process committed log entries in Algorithm 1. The executing thread holds a lock on the database, log, and log application counter throughout execution, so no honest process will have a thread outside this process change the Merkle tree during that execution. When applying a commited log to the local database, a replica will begin by reading the log application counter lac, the log entry at that index entry, and the database row row referenced by that log entry onto a single memory page, which we will call the *working page*. When reading each of these items, it will verify its Merkle proof $(\Pi_{\mathsf{lac}}, \Pi_{\mathsf{entry}}, \Pi_{\mathsf{row}})$ and also copy the root of the Merkle tree for each read onto the working page. After copying this data, we verify that the Merkle roots associated with each read are equal, determine whether the number of uses of this row has surpassed the configured maximum, and update the row by incrementing the usage count and deleting the OPRF secret if the maximum usage count has been exceeded. We then update the row in the database and increment the log application counter, updating the Merkle tree entries for both, then proceed with evaluating the OPRF, if the key is present, and finally respond to the client.

If the attacker rolls back the database row to the contents of a previous timestep, it first has to roll back every entry from the row to the Merkle tree root. However, the root also covers the log entries and log application counter, which are modified when a database row is modified (how SVR3 achieves atomicity of this operation is described above). Thus, the attacker will have to roll back the log as well; rolling back the log is exactly what Raft$^{\circlearrowleft}$ protects against.

Atomic regions. Because all of our working memory fits on a single page, operations are atomic with respect to the attacker's ability to rollback memory at the page granularity. In order to support more modern enclaves that only have cache line granularity (e.g., 16B), we need to implement atomic regions that are guaranteed to run without interruption by an attacker. We describe in detail how to implement atomic regions on SGX and SEV-SNP in §C of the full version [17] by utilizing the interrupt handler in AEX-Notify [18]. AEX-Notify mitigates SGX-Step, an attack framework that makes it possible to single-step enclave programs [92]. It does so by introducing an instruction set architecture extension to support a custom handler on interrupt. The SGX-Step mitigation leverages this handler to speed up the next instruction so that the attacker is statistically unlikely to 'hit' the next instruction's execution with an APIC timer. This mechanism also allows us to implement atomic regions, in a similar fashion to restartable sequences [8]. At a high level, we set a flag in a fixed register when an interrupt occurs, and we check this flag at the end of the atomic region to determine whether to restart the atomic region. If the flag is set, we restart and retry until it runs without any interrupt. We leave optimizing this approach in a secure manner to future work.

Algorithm 1 Applying a committed log entry. We describe in text how we process committed log entries in §5.3.2.

```
$\mathsf{workspace}_R \leftarrow (\mathsf{lac}, \Pi_{\mathsf{lac}}, \mathsf{entry}, \Pi_{\mathsf{entry}}, \mathsf{row}, \Pi_{\mathsf{row}})$
Atomic region.
                                        ▷ Abort on any Verify failure.
    $\mathsf{failure} \leftarrow 0$
    $\mathsf{Verify}(\Pi_{\mathsf{lac}}.\mathsf{root} \overset{?}{=} \Pi_{\mathsf{entry}}.\mathsf{root} \overset{?}{=} \Pi_{\mathsf{row}}.\mathsf{root})$
    $\mathsf{Verify}(\mathsf{entry.clientid} \overset{?}{=} \mathsf{row.clientid})$
    $\mathsf{Verify}(\mathsf{lac}, \Pi_{\mathsf{lac}}); \mathsf{Verify}(\mathsf{entry}, \Pi_{\mathsf{entry}});$
    $\mathsf{Verify}(\mathsf{row}, \Pi_{\mathsf{row}})$
    if $\mathsf{row.guess\_cnt} < \mathsf{max\_guesses}$ then
        $\mathsf{evaluated} \leftarrow \mathsf{OPRFEval}(\mathsf{row.sk}, \mathsf{blinded})$
        $\mathsf{row.guess\_cnt} \leftarrow \mathsf{row.guess\_cnt} + 1$
    else
        $\mathsf{failure} \leftarrow 1$
        $\mathsf{row.sk} \leftarrow 0, \mathsf{row.guess\_cnt} \leftarrow \mathsf{UINT\_MAX}$
    end if
    $\mathsf{workspace}_W \leftarrow (\mathsf{row}, \mathsf{UpdatePrf}(\mathsf{row}, \Pi_{\mathsf{row}}))$
$\Pi'_{\mathsf{row}} \leftarrow \mathsf{UpdatePrf}(\mathsf{row}); \Pi'_{\mathsf{lac}} \leftarrow \mathsf{UpdatePrf}(\mathsf{lac})$
Check that leaves on path in $\Pi'_{\mathsf{row}}, \Pi'_{\mathsf{lac}}$ match $\Pi_{\mathsf{row}}, \Pi_{\mathsf{lac}}$.
if failure then return MISSING
else return (OK, evaluated)
end if
```

6 Operations

Production systems need upgrades. This is a challenge for us because we want to defend against malicious administrators: a secure system can become completely insecure if a malicious administrator can push arbitrary code to the system. At a high level, we defend against malicious code updates by ensuring that users can audit the code that is running; the code is open source, and enclaves attest to the security-relevant server code and configurations running.

Adding new servers. When a new server is launched in a trust domain, it connects to a discovery service and registers a new group if no replica group is registered. If there is an existing replica group, the new server will select a peer in that group, validate that its enclave measurements match, and create an attested connection with that peer. By checking that enclave measurements match, SVR3 ensures that an administrator cannot add a server running different code. The new server then requests to join the group, and the existing server transfers all log entries and database rows to the new server. This is done over a Noise protocol [71] channel with key resetting and hybrid post-quantum forward secrecy [70] to provide robust forward secrecy. Once the transfer is complete, the replica group goes through the membership change process to add the new server (which requires a quorum).

Sometimes security-required microcode updates need to be applied to all servers. Since all data is kept in volatile enclave memory, there is no way to reboot the machine without losing all replica data. In this situation, *all members of the cluster must be replaced.* This can be done by sequentially adding new servers on patched hardware, then terminating old servers.

Clients. Android, iOS, and desktop clients are deployed through app stores with auditable, open-source code. Each client contains hard-coded information about which enclave measurements (for remote attestation), platform versions, and cluster configurations to accept. If a client attempts to connect to a SVR3 cluster and finds unexpected measurements or configuration, it will abort the connection.

Service upgrades and data migration. Since server enclaves can only communicate with peers that share the same enclave measurements, there is no mechanism to migrate data directly from an old version of an enclave-backed service to a new one. Instead, data migration flows through the client. To accomplish this, when a new version of a client is released that contains measurements for the new enclave, this client will recover its secret from the old servers (if it is not cached in local storage), and then it will back up its secret to the next version of the service. It takes approximately 90 days for a new client software release to fully reach the user base, so the new enclave-backed service must run alongside the older version during this 90-day window.

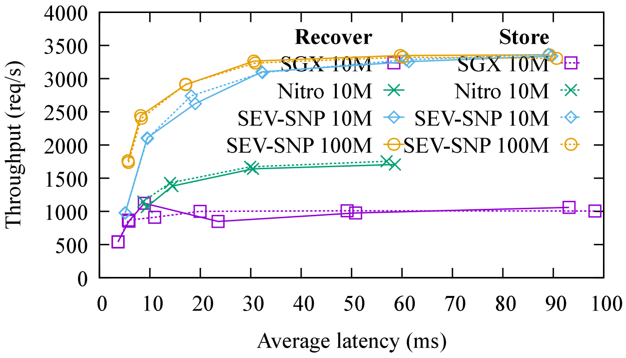


Figure 4: Average latency vs. throughput.

7 Implementation

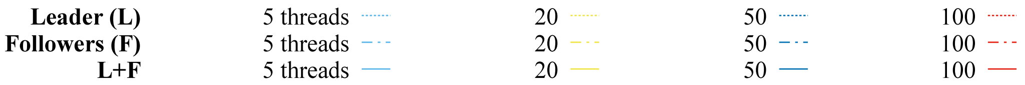
We implemented SVR3 in ~8,800 lines of C++ for the enclave and ~5,300 lines of Go for the untrusted host. For the SGX deployment we use the OpenEnclave framework v0.19 [67] and Intel SGX v2.22. For the Nitro deployment we use the Nitro Security Module library v0.4 [63]. We use a Noise protocol [71] channel on top of TCP for communication between replicas and websockets for communication with clients. We use protobuf [73] to define formats for all wire messages. In addition to handling client and peer requests, the host offers a control interface for administration as well as sophisticated metrics collection that is integrated with our internal monitoring and reporting systems. Our implementation assumes enclave page-level integrity, and we estimate overheads for supporting 16B-level rollback granularity in §8.1. The implementation is open source and the consensus system is already in production use. The full system is being deployed to production at the time of publication. Production deployments use 7 geographically distributed servers and a supermajority parameter of 2. Further details about the production deployment are in §B of the full version [17].

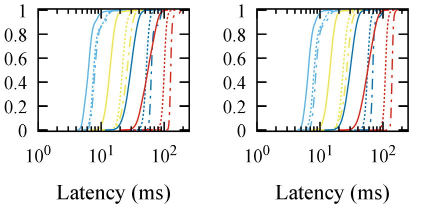
8 Evaluation

We investigate the overheads of running SVR3 (§8.1) and the performance perceived by the end user (§8.2).

Evaluation setup. For the purposes of this paper, we evaluate end-to-end performance on our organization's staging system, configured to handle 10 million users. This limit is due to available enclave memory, not compute. Staging clusters are configured with a supermajority parameter of 1 and consist of 3 environments (trust domains), each with 5 replicas deployed in the same region:

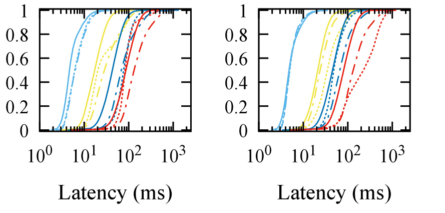
- AWS Nitro: `m5.xlarge` instances with 2 cores and 10 GB RAM per enclave ($142/month/server).
- Intel SGX at Azure: `DC2s_v3` instances with 2 cores and 8 GB EPC RAM per enclave ($140/month/server).





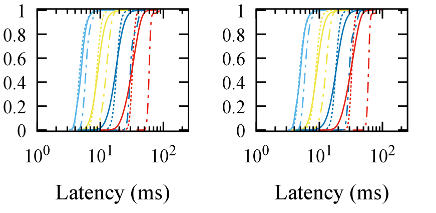
(a) StoreSecret (b) RecoverSecret

Figure 5: Request latency CDF for AWS Nitro, varying number of client threads, 10M users.



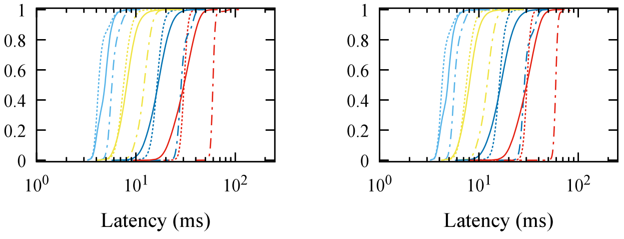
(a) StoreSecret (b) RecoverSecret

Figure 6: Request latency CDF for Intel SGX, 10M users.



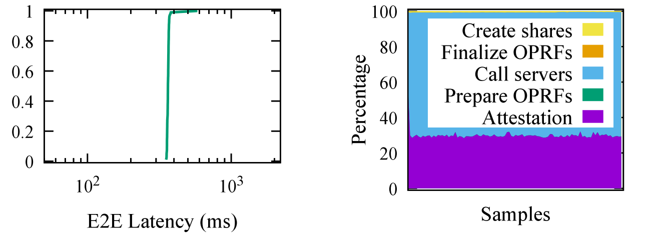
(a) StoreSecret (b) RecoverSecret

Figure 7: Request latency CDF for AMD SEV-SNP, 10M users.



(a) StoreSecret (b) RecoverSecret

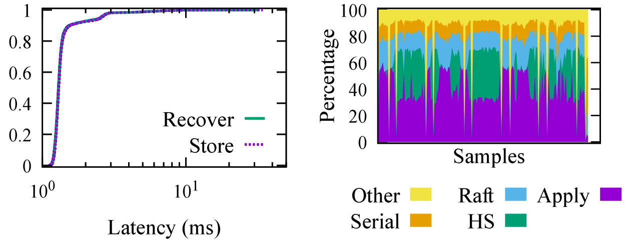
Figure 8: Request latency for AMD SEV-SNP, 100M users.



(a) Request (StoreSecret) latency CDF.

(b) Client request latency breakdown.

Figure 11: End-to-end performance.



(a) Request latency CDF.

(b) Request latency breakdown. HS = Noise handshake, Serial = serializing/deserializing protobufs, Apply = applying log entry (§5.3.2).

Figure 9: SVR3 performance without network latency from Raft⥀.

Enclave	Network (B/user)			
	StoreSecret		RecoverSecret	
	C ↔ S	S ↔ S	C ↔ S	S ↔ S
SGX	20,717	288–1,276	20,717	224–1,212
SEV-SNP	4,406	288–1,276	4,406	224–1,212
Nitro	4,593	288–1,276	4,593	224–1,212

Table 10: Network usage for a single client request to a 3-replica cluster. S=server, C=client. C ↔ S for SEV-SNP is an estimate.

- AMD SEV-SNP at GCP: 2 `n2d-standard-2` instances per enclave (one "confidential" and one for the untrusted host) with 2 cores and 8 GB RAM ($2 \cdot (\$70) = \$140$/month/server).

In total, the staging cluster costs $2,110/month to run ($0.0025/user/year). For microbenchmarking, we evaluate on a testing cluster with the same machine types as our staging cluster but with 3 replicas per trust domain instead of 5 and a supermajority parameter of 0 instead of 1.

Our production infrastructure has more replicas (with more cores and RAM per replica) and is set up to handle over 500 million users (more details in §B of the full version [17]). We provision for 1 req/s/1M users and ~256B of RAM/user. Our experience operating this system gives us confidence that evaluating on the staging infrastructure is meaningful and that SVR3 scales gracefully. To validate this claim, we also evaluate on an AMD SEV-SNP cluster with 100 million users using `n2d-standard-4` instances (4 cores and 16 GB RAM).

8.1 Microbenchmarks

Throughput. We plot an average latency vs. throughput curve for write and recovery requests in Figure 4. We generate each point by varying the number of client threads and measuring the average latency and throughput of requests. Requests are spread out across all 3 servers. For the 10M-user deployments, the throughput of recovery requests levels off around 1,700 req/s for Nitro, 1,000 req/s for SGX, and 3,300 req/s for SEV-SNP (for both 10M-user and 100M-user deployments).

Latency. We plot CDFs of the latency of write and recovery requests in Figure 5, Figure 6, and Figure 7 for Nitro, SGX, and SEV-SNP, respectively. Within each figure, we plot the latency when requests are sent only to the leader, when requests are sent only to followers, and when requests are sent to all 3 servers. Requests sent to followers are forwarded to the leader, so the average latency of requests at followers is higher than at the leader. The latency distribution of requests when sending requests to all 3 servers improves compared to sending requests to only followers. The latency distribution is better than sending requests to only the leader for Nitro and SGX, and the tail latency is worse than sending requests to only the leader for SEV-SNP. At 100 client threads, the average latency for requests sent to all servers for key recovery is 56.9ms for Nitro, 98.3ms for SGX, and 32.3ms for SEV-SNP. We also plot the CDFs of recovery request latency for the 100M-user SEV-SNP deployment in Figure 8. The latency distribution of the 100M-user deployment is very similar to the 10M-user deployment and the average latency of the requests sent to all 3 servers for key recovery is 30.9ms.

We note that a majority of the latency is due to network latency when appending to the Raft$^{\circlearrowleft}$ log, which we validate in Figure 9. We run the same experiment as above, but with 1 client thread and 1 SGX node (effectively disabling the network requests of Raft$^{\circlearrowleft}$). We plot the CDF of request latencies under this regime in Figure 9a, and the average latency of these requests is 1.47ms. We also profile the server and plot the percentage of CPU ticks in Figure 9b. On average, the Noise handshake is about 35%, applying the log entry is about 21%, and 13% is encrypting peer messages for Raft$^{\circlearrowleft}$. The yellow spikes are due to periodic updating of environment statistics, which also contributes to the long tail request latencies in SGX (Figure 6).

Impact of supporting 16B-granularity. Informed by latency measurements, we can upper-bound the impact of latency from achieving page-level integrity from 16B-granularity using atomic regions (§5.3.2). Applying the log entry (which we will conservatively make an entire atomic region) takes $1.47 \cdot 0.21 = 0.3$ms. We could be interrupted by the APIC timer, the end of a thread scheduling quantum, or by a page fault from a memory access, of which there are $5 \cdot \log_2(100,000,000) = 120$ (from the Merkle tree accesses in Algorithm 1). In the worst case, we would repeat execution of the atomic region 122 times, resulting in a worst-case additional latency of 36.6ms. Note that this is a (very) loose upper bound and is still below user perceptibility.

Network usage. We measure the network usage of SVR3 running on each enclave type for a 3-replica cluster in Table 10. There is a range of network usage for Server ↔ Server because it depends on how many requests have been batched into a single Raft$^{\circlearrowleft}$ append request. The network usage between servers also depends on the number of servers in the cluster, growing proportionally to $m-1$ for m servers. From a deployment perspective, we are more concerned with the Client ↔ Server bandwidth, which is under 20KB for all enclave types. This is because exchanging more data between the client and the server can become a usability issue for users with limited data plans.

Memory usage. We measured the memory usage of SVR3 on SGX, varying the number of users in the system. Note that we expect the memory usage to be similar for all enclave types, since they are storing the same amount of data for each user. We find that memory usage grows by ~450B/user until we start truncating the log at 100MB and then settles into a steady 170B/user added. At 100 million users, SVR3 uses 18.5GB of memory on each server, which is 185B/user/server.

8.2 End-to-end performance

We measure the end-to-end performance of SVR3 by running a client that stores its secret key by sending a (sequential) request to a server in each enclave cluster. For a more representative deployment, we geographically distribute the SGX cluster (`centralus`, `eastus`, `eastus2`, `southcentralus`, `westus`), the SEV-SNP cluster (`us-central1`, `europe-west3`, `asia-southeast1`, `europe-west4`, `europe-west3`), and the Nitro cluster (`us-east-1`, `us-east-2`, `us-west-1`, `us-west-2`, `eu-north-1`). The performance for recovering a key is almost identical to the performance for storing a key, so we only report the performance for storing a key. We plot the CDF of the latency of these requests in Figure 11a. The average end-to-end latency is 365ms, which is reasonable for a user to wait for a key recovery or key backup request. We plot the breakdown of the latency in Figure 11b. The majority of the latency is from waiting for servers to respond (69.3%), followed by remote attestation with the servers (29.9%).

9 Related work

Secret recovery systems. A number of companies have deployed secret recovery systems using secure hardware: Apple protects user iCloud data using hardware security modules (HSMs) [4, 43], Google protects Android backups using secure microcontrollers [96], and WhatsApp protects message histories using HSMs [98]. WhatsApp runs vanilla Raft [65] on a geographically distributed cluster of HSMs and uses OPAQUE [39] for key recovery. WhatsApp's consensus only requires one round trip between the leader and the replicas while SVR3 requires an extra round of communication (to guarantee safety in the face of rollbacks). Davies et al. analyzed the security of the WhatsApp encrypted backup protocol [24].

Like SVR3, all of these systems use secure hardware to allow a user to recover a cryptographic secret using a low-entropy secret (e.g., a 4-digit PIN). Unlike SVR3, they rely on a single type of secure hardware: the compromise of one secure hardware device can compromise many users' secrets.

Juicebox [88] is a key recovery protocol that distributes trust across one type of secure hardware and multiple trust domains in the traditional manner (across organizations). SVR3 has a simpler protocol that is not a multi-round PAKE as our servers never learn whether the PIN is guessed correctly or not (keys are deleted unconditionally when guesses run out). Secret shares are also stored directly on the servers in Juicebox. Thus, to prevent an attacker who compromises a threshold number of trust domains from reconstructing all the secrets without needing to mount a dictionary attack, they must mix the reconstructed secret with the PIN to create an encryption key that is then used to encrypt the target secret.

SafetyPin [20] is a PIN-based end-to-end encrypted backup system that defends against an attacker that can adaptively compromise some percent of HSMs. While SafetyPin protects against a more powerful attacker model, it requires a comparatively large number of HSMs.

Tutamen [78], Acsesor [11], and CanDID [52] split trust across multiple entities to allow users to recover their secrets (among other operations). Chen et al. [13] use cloud storage for secret recovery. These systems do not use secure hardware; the use of enclaves in SVR3 provides additional security and requires us to design for their limitations (e.g., rollback attacks). CALYPSO [42] also shards user secrets across different entities but, unlike SVR3, uses a blockchain. PreVeil [72] shards secret keys across other peers in a social or work graph, but requires manual setup from the user.

Another line of work has taken a more theoretical approach to the problem of secret key backups. Benhamouda et al. [7] use a proof-of-stake blockchain to allow users to store secrets while protecting against an attacker that can adaptively compromise a percent of the stake. Subsequent work improves efficiency in this model via batching [30].

Orisini et al. [68] also describe a scheme for end-to-end encrypted backups, but in their scheme, the user does not need to remember a PIN or something similar. Instead, users continuously monitor for illegitimate recovery attempts, allowing an honest user to thwart malicious recovery attempts but later recover their backup. While this approach is appealing in that it eliminates the PIN, it does not work for our setting where clients may go offline for extended periods of time.

Multi-party computation and secure hardware. Cryptocurrency wallets protect user secrets by distributing them across hardware enclaves or HSMs [27, 29, 41, 77, 81]. Cryptocurrency wallets are designed to avoid materializing the key in a single location rather than to enable users to recover secrets. Myst provides security by splitting trust across many hardware devices and operations like signing and decryption [54]. More broadly, prior work has examined composing multi-party computation and secure hardware for efficiency [6, 25, 44, 64]. Our use of secure hardware with multi-party computation is tailored to encrypted backups and, while this line of work uses secure hardware to reduce the costs of multi-party computation, we use it to augment the security of the system. In prior work [21], we observed that heterogeneous secure hardware hosted by different clouds can be useful for deploying systems that split user secrets, including encrypted backups, but we had not yet worked through and built out such a deployment.

Rollback prevention in enclaves. There has been a rich line of work on preventing rollback attacks in enclaves. Memoir [69] and Ariadne [86] store a small amount of state inside non-volatile memory (NVRAM) and use that to reconstruct application state during recovery. Both approaches are scoped to single machines, and do not provide availability in the event of a machine permanently failing. ROTE [53] uses a broadcast algorithm across enclaves to maintain a distributed counter, but requires NVRAM to update group membership, whereas we use our Raft$^{\circlearrowleft}$ log to update membership. Additionally, the abstraction that ROTE offers is one of a counter instead of generic log entries. Engraft [97] examines the safety issues of running off-the-shelf consensus inside enclaves. They use an underlying broadcast protocol similar to ROTE to maintain a distributed counter and introduce additional mechanisms to support node recoverability. However, in our setting, we can simply start a new node in the event of a node failure, so we do not need to support node recoverability.

Nimble [3] is a lightweight replication protocol that provides a freshness-guaranteed ledger. The ledger can be used to keep track of the state of untrusted storage, enabling applications that run on enclaves to persist their state to external (untrusted) storage and detect potential rollbacks on that storage. Note that our system is already protected against the class of rollback attacks on external storage described in §1 of [3] because *all data is stored and maintained in memory*. Nimble's threat model does not include physical rollback attacks on the enclave (both endorser and application). However, minimizing SVR3's trusted computing base (TCB) is an interesting and important future direction, and we discuss potential design decisions and open challenges in §10.

TrInc [49] shows that a secure log can be implemented with a secure counter. However, realizing a secure counter on enclaves is difficult. We cannot write PCRs to the TPM from inside an SGX enclave, and additionally, TPMs can limit the speed of counter updates (§6.1.1, [86]). CPU registers are written to the SSA, which can be rolled back. On SGX there is no CPU register where only an enclave can write to it. We are unaware of an (efficient) secure counter primitive on newer enclaves after consulting with Intel.

Consensus protocols. As Dinis et al. [26] point out, rollback behavior can be considered a subset of Byzantine behavior, so the Byzantine fault tolerant (BFT) model is stronger than necessary for our setting. Consequently, Raft$^{\circlearrowleft}$ is lighter weight than BFT flavors of Raft protocols like Tangoroa [19] which requires $O(m^2)$ communication scaling in the number of replicas. The supermajority parameter in Raft$^{\circlearrowleft}$, which increases the quorum size, is comparable to PBFT's [10] Byzantine nodes value. Engraft [97] and RR (TEEMS) [26] address data-sealing (software) rollback attacks. SVR3 not only defends against these data-sealing rollback attacks, but also defends against physical rollback attacks.

10 Discussion

Consensus in the enclave. Nimble [3] is able to maintain a secure log while removing the consensus mechanism from the TCB, and an important future direction for SVR3 would be to similarly minimize its TCB. However, it is not entirely straightforward, and there are interesting design and engineering challenges to address. First, Nimble will need to be hardened against physical rollback attacks, which seems straightforward to do. More significant is that since this log—which contains OPRF secrets—will be held in untrusted storage, it must be encrypted. This has important consequences for our system as we describe below, and addressing them may result in significant additional complexity (and thus increase the TCB).

First, we note that we will need enclaves similar to the ones we have today to handle client requests. These enclaves will now need to share a common encryption key to encrypt and decrypt these log messages. This shared key becomes a new single point of failure for the system. To maintain the forward secrecy we have today due to our use of Noise protocol [71] channels with rekeying between enclaves, it seems the enclaves will need to participate in some sort of continuous group key agreement (CGKA) [2] to rotate the key periodically and on membership changes.

Second, if this new system aims to keep the TCB small by maintaining the database state outside of the enclave, as with Juicebox [88] or WhatsApp [98], then the encryption key for the database becomes another single point of failure, but in this case it is not clear how we can achieve forward secrecy without periodically re-encrypting the entire database. If, on the other hand, we maintain the database in enclave memory, as we do now, then the use of CGKA to protect the encrypted log means that new members of a replica group will not be able to read old log messages to construct the database state. While we have a state transfer mechanism in our current system to handle truncated logs, we will need to refine it to ensure that new members are correctly initialized.

Taken together, we see removal of the consensus mechanism from the TCB as a project that requires careful design and analysis and significant engineering work that adds its own complexity. We note that the consensus protocol is a relatively small (1,541 LOC in C++) and well-understood part of our current codebase, so we need—and hope to find—clear rationale for its removal.

In-memory vs. disk-based storage. While disk-based storage solutions are cheaper than keeping the entire database of key recovery shares in memory, they are more susceptible to rollback attacks because the secrets are taken out of the enclave, and even enable rollback attacks that are software-based and can be performed without physical access.

Data privacy compliance. In general, a multi-cloud deployment may complicate compliance with data privacy laws. The design of SVR3, however, keeps compliance simple since by preventing any user data from being processed by our servers and blocking our administrators from accessing sensitive keys.

Malicious clients. SVR3 provides security guarantees for users using our clients, which we assume are well-behaved. Our client code is open source [55–57], and scrutinized by the community. If the user's client is compromised and malicious (e.g., the user has malware), it can affect the security of that user, but not the security or experience of other users with uncompromised clients.

Honest cloud providers? If we could assume that most cloud operators are honest, then that could change the parameterization of SVR3 (e.g., setting the number of trust domains that can be compromised t to 1), though this would also require assuming that the enclaves were not susceptible to any future vulnerabilities that could be exploited remotely. We would still use enclaves to prevent malicious system administrators from running arbitrary server code.

11 Conclusion

SVR3 demonstrates the potential of systems that provide security through a combination of cryptography and a diverse set of hardware enclaves, without putting trust in any single hardware component. Using different types of enclaves leads to an array of deployment challenges stemming from heterogeneous and shifting attacker models. SVR3 is a powerful defense against the evolving landscape of enclave security: by distributing trust across enclaves, even if a new threat arises in one type of enclave, user secrets are still secure. SVR3 costs \$0.0025/user/year and takes 365ms for a user to recover their key, which is a rare operation.

Acknowledgments. We are very grateful to our shepherd, Jay Lorch, for his detailed feedback, as well as to the collective OSDI reviewers for their suggestions, which greatly improved the presentation of this paper. We thank Trevor Perrin for the pointer to the PPSS construction [37]. Mark Johnson helped develop the early stages of this project. Ravi Khadiwala contributed significantly to SVR3's implementation. We thank Natacha Crooks, Christopher Fletcher, Jack Humphries, and Sky security students for their helpful feedback. The UC Berkeley authors are supported by NSF Graduate Research Fellowships, a Microsoft Ada Lovelace Fellowship, and gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, MBZUAI, Samsung SDS, SAP, Uber, and VMware.

References

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS*, 2016.

[2] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *TCC*, 2020.

[3] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback protection for confidential cloud services. In *OSDI*, 2023.

[4] Apple. iCloud Keychain security overview, 2021. https://support.apple.com/guide/security/icloud-keychain-security-overview-sec1c89c6f3b/.

[5] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS*, 2011.

[6] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In *FC*, 2017.

[7] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *TCC*, 2020.

[8] Brian N Bershad, David D Redell, and John R Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS*, 1992.

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

[10] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.

[11] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. *Cryptology ePrint Archive 2022/1729*, 2022.

[12] 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*, 2019.

[13] Long Chen, Ya-Nan Li, Qiang Tang, and Moti Yung. End-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage. In *USENIX Security*, 2022.

[14] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security*, 2021.

[15] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[16] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. In *ISeCure*, 2011.

[17] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a global-scale end-to-end encryption system. *Cryptology ePrint Archive 2024/887*, 2024.

[18] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *USENIX Security*, 2023.

[19] Christopher Copeland and Hongxia Zhong. Tangaroa: a Byzantine fault tolerant Raft, 2016. https://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf.

[20] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with human-memorable secrets. In *OSDI*, 2020.

[21] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *HotNets*, 2022.

[22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[23] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious pseudorandom functions (OPRFs) using prime-order groups. https://www.ietf.org/id/draft-irtf-cfrg-voprf-21.html.

[24] Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. *Cryptology ePrint Archive 2023/843*, 2023.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security*, 2014.

[26] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. RR: A fault model for efficient TEE replication. In *NDSS*, 2023.

[27] Fireblocks. https://www.fireblocks.com/platforms/mpc-wallet/.

[28] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[29] Gemini. Cold storage, keys & crypto: How Gemini keeps assets safe. https://www.gemini.com/blog/cold-storage-keys-crypto-how-gemini-keeps-assets-safe.

[30] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *PKC*, 2022.

[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE S&P*, 2018.

[32] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[33] Feng Hao and Paul C van Oorschot. SoK: Password-authenticated key exchange–theory, practice, standardization and real-world lessons. In *AsiaCCS*, 2022.

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[35] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via rowhammer attack. In *SysTEX*, 2017.

[36] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, 2014.

[37] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, 2016.

[38] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS*, 2017.

[39] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[40] Your Keybase account. `https://book.keybase.io/account`.

[41] Knox. Knox custody. `https://www.knoxcustody.com/security`.

[42] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. *Cryptology ePrint Archive 2018/209*, 2018.

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

[44] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *IEEE S&P*, 2020.

[45] Leslie Lamport. Specifying systems: The TLA+ language and tools for hardware and software engineers. 2002.

[46] Ledger. How Ledger device generates 24-word recovery phrase. `https://support.ledger.com/hc/en-us/articles/4415198323089-How-Ledger-device-generates-24-word-recovery-phrase`, November 2023.

[47] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*, 2020.

[48] 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*, 2017.

[49] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[50] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. The deployment dilemma: Merits & challenges of deploying MPC, 2023. `https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html`.

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

[52] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, Sybil-resistance, and accountability. In *IEEE S&P*, 2021.

[53] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *USENIX Security*, 2017.

[54] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. A touch of evil: High-assurance cryptographic hardware from untrusted components. In *CCS*, 2017.

[55] Signal Messenger. Signal Android client. `https://github.com/signalapp/Signal-Android`.

[56] Signal Messenger. Signal desktop client. `https://github.com/signalapp/Signal-Desktop`.

[57] Signal Messenger. Signal iOS client. `https://github.com/signalapp/Signal-iOS`.

[58] Meta. End-to-end encryption on Messenger explained, 2024. `https://about.fb.com/news/2024/03/end-to-end-encryption-on-messenger-explained/`.

[59] Microsoft. Bitlocker whitepaper Windows 10. `https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_firmware/pdf_3/Bitlocker_White_Paper_Windows_10.pdf`, 2018.

[60] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, 2018.

[61] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[62] 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 *IEEE S&P*, 2020.

[63] Nitro secure module. `https://github.com/aws/aws-nitro-enclaves-nsm-api/tree/v0.4.0`.

[64] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[65] Diego Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014.

[66] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[67] Open Enclave SDK. `https://github.com/openenclave/openenclave/tree/v0.19.0`.

[68] Chris Orsini, Alessandra Scafuro, and Tanner Verber. How to recover a cryptographic secret from the cloud. *Cryptology ePrint Archive 2023/1308*, 2023.

[69] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *IEEE S&P*, 2011.

[70] Trevor Perrin. KEM-based hybrid forward secrecy for Noise. 2018. https://github.com/noiseprotocol/noise_hfs_spec/blob/master/output/noise_hfs.pdf.

[71] Trevor Perrin. The Noise protocol framework. 2018.

[72] PreVeil: Encrypted email and file sharing. https://www.preveil.com/.

[73] Protocol buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

[74] Proton Mail. https://proton.me/mail.

[75] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *IEEE S&P*, 2021.

[76] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *SOUPS*, 2019.

[77] Riddle&Code. Hardware security modules vs. secure multiparty computation in digital asset custody: The drawback of choosing just one and what happens when you combine them. https://www.riddleandcode.com/blog-posts/hardware-security-modules-vs-secure-multi-party-computation-in-digital-asset-custody.

[78] Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald. Tutamen: A next-generation secret-storage platform. In *SoCC*, 2016.

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[80] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[81] Sepior. https://sepior.com/products/advanced-mpc-wallet.

[82] Pavitra Shankdhar. Popular tools for brute-force attacks. https://resources.infosecinstitute.com/topics/hacking/popular-tools-for-brute-force-attacks/, 2020.

[83] Rob Shirley. Internet security glossary, version 2. https://datatracker.ietf.org/doc/html/rfc4949.

[84] Signal Messenger. Secure Value Recovery Service v2/3. https://github.com/signalapp/SecureValueRecovery2.

[85] Signal Messenger. https://signal.org/.

[86] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[87] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security*, 2017.

[88] Nora Trapp. Key to simplicity: Squeezing the hassle out of encryption key recovery, 2024. https://www.juicebox.xyz/blog/key-to-simplicity-squeezing-the-hassle-out-of-encryption-key-recovery.

[89] Anna Trikalinou and Dan Lake. Taking DMA attacks to the next level. 2017.

[90] 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*, 2018.

[91] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, 2020.

[92] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

[93] 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*, 2017.

[94] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[95] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. *arXiv preprint arXiv:2006.13353*, 2020.

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

[97] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *CCS*, 2022.

[98] WhatsApp. Security of end-to-end encrypted backups, 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[99] Kyle Wiggers. Apple launches end-to-end encryption for iCloud data. *TechCrunch*, 2022.

[100] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.