Fast Batched DPSS and its Applications

Vipul Goyal[1,2], Abhiram Kothapalli[1], Elisaweta Masserova[1], Bryan Parno[1], and Yifan Song[1]

[1] Carnegie Mellon University
[2] NTT Research

Abstract. A *secret sharing* scheme enables one party to distribute shares of a secret to n parties and ensures that an adversary in control of t out of n parties will learn no information about the secret. However, traditional secret sharing schemes are often insufficient, especially for applications in which the set of parties who hold the secret shares might change over time. To achieve security in this setting, *dynamic proactive secret sharing* (DPSS) is used. DPSS schemes proactively update the secret shares held by the parties and allow changes to the set of parties holding the secrets. We propose FaB-DPSS (FAst Batched DPSS) – a new and highly optimized batched DPSS scheme. While previous work on batched DPSS [BDLO15] focuses on a single client submitting a batch of secrets and does not allow storing and releasing secrets independently, we allow multiple *different* clients to dynamically share and release secrets. FaB-DPSS is the most efficient robust DPSS scheme that supports the highest possible adversarial threshold of $\frac{1}{2}$. We prove FaB-DPSS secure and implement it. All operations complete in seconds, and we outperform a prior state-of-the-art DPSS scheme [MZW$^+$19] by over 6×.

Additionally, we propose new applications of DPSS in the context of blockchains. Specifically, we propose a protocol that uses blockchains and FaB-DPSS to provide conditional secret storage. The protocol allows parties to store secrets along with a release condition, and once a (possibly different) party satisfies this release condition, the secret is privately released to that party. This functionality is similar to *extractable witness encryption.* While there are numerous compelling applications (e.g., time-lock encryption, one-time programs, and fair multi-party computation) which rely on extractable witness encryption, there are no known efficient constructions (or even constructions based on any well-studied assumptions) of extractable witness encryption. However, by utilizing blockchains and FaB-DPSS, we can easily build all those applications. We provide an implementation of our conditional secret storage protocol as well as several applications building on top of it.

1 Introduction

In recent years, secret sharing schemes have received considerable attention. While traditional secret sharing is a well-known cryptographic primitive which has been extensively used in the context of secure multi-party computation, recently *proactive secret sharing* has become increasingly important. Similar to traditional secret sharing, proactive secret sharing schemes enable one party to distribute shares of a secret to n parties such that any $t+1$ shares are enough to reconstruct the secret, and an adversary in possession of t out of n shares learns no information about the secret. In contrast to traditional secret sharing, proactive secret sharing additionally considers the setting where the adversary may eventually corrupt *all* participants over time, while corrupting no more than a certain threshold at any given time. In the context of blockchains, proactive secret sharing has proven a useful alternative to central storage for securing secret keys (which are used to sign transactions, access cryptocurrency wallets, etc.). As pointed out in CHURP [MZW$^+$19], since blockchain nodes are typically allowed to freely leave or join a system at any time, in this context it is critical to allow for *dynamic* changes in the secret sharing committee. This is supported by *dynamic proactive secret sharing* (DPSS) [DJ97, MZW$^+$19, SLL08, BDLO15, ZSVR05, WWW02] which proactively updates the secret shares held by the parties and allows changes to the set of parties holding the secrets.

Scheme	**Dynamic setting**	**Adversary**	**Threshold**	**Network**	**Comm. (amort.)**	**Comm. (non-amort.)**
[HJKY95]	No	Active	$t/n < 1/2$	synch.	$O(n^2)$	$O(n^2)$
[CKLS02]	No	Active	$t/n < 1/3$	asynch.	$O(n^4)$	$O(n^4)$
[DJ97]	Yes	Passive	$t/n < 1/3$	asynch.	$O(n^2)$	$O(n^2)$
[WWW02]	Yes	Active	$t/n < 1/2$	synch.	$exp(n)$	$exp(n)$
[ZSVR05]	Yes	Active	$t/n < 1/3$	asynch.	$exp(n)$	$exp(n)$
[SLL08]	Yes	Active	$t/n < 1/3$	asynch.	$O(n^4)$	$O(n^4)$
[BDLO15]	Yes	Active	$t/n < 1/2 - \epsilon$	synch.	$O(1)$	$O(n^3)$
[MZW$^+$19]	Yes	Active	$t/n < 1/2$	synch.	$O(n^2)$	$O(n^2)$
This work	Yes	Active	$t/n < 1/2$	synch.	$O(n)$	$O(n^2)$

Fig. 1: Comparison of PSS Schemes. The Comm. columns show the communication cost/secret in a hand-off round.

We introduce FaB-DPSS (FAst Batched DPSS) – a highly optimized batched DPSS scheme. Especially in the context of secret storage on blockchains, batching is crucial as thousands of secrets might be stored and updated in parallel at any given time. FaB-DPSS improves over prior work in multiple dimensions. In contrast to previous work on batched DPSS [BDLO15], which focuses on a single client submitting a batch of secrets and does not allow storing and releasing secrets independently, we allow multiple *different* clients to dynamically and independently share and release secrets. Among the robust schemes which allow the highest-possible adversarial threshold of $\frac{1}{2}$ (see Figure 1), our protocol has the best communication complexity. It is also the most concretely efficient scheme – all operations complete in seconds (§8), and we outperform a prior state-of-the-art DPSS scheme [MZW$^+$19] by over 6×. These improvements are possible because of our entirely new approach to the hand-off phase of the DPSS – instead of relying on bivariate polynomials as is done in prior work [MZW$^+$19], we use a technique we dub "coupled sharings".

In addition to FaB-DPSS, we propose a number of blockchain-based DPSS applications, thus expanding the reach of DPSS in the context of blockchains. The most important one can be seen as a blockchain-based alternative to *extractable witness encryption*. Introduced by Garg et al. [GGSW13], a witness encryption scheme is, roughly, a primitive that allows one to encrypt a message with respect to a problem instance. Such a problem instance could be a sudoku puzzle in a newspaper or an allegedly bug-free program, or more generally, any NP search problem. If the decryptor knows a valid witness for the corresponding problem instance, such as a sudoku solution or a bug in the program, she can decrypt the ciphertext. Moreover, if a witness encryption scheme is *extractable*, then an adversary able to learn any non-trivial information about the encrypted message is also able to provide a witness for the corresponding problem instance.

Unfortunately, existing proposals for extractable witness encryption typically rely on *differing-inputs* obfuscation [BGI+12, ABG+13], a technique that is computationally expensive and relies on strong cryptographic assumptions. Responding to the lack of extractable witness encryption schemes based on standard assumptions, Garg et al. [GGHW14] suggest that it may be impossible.

Building upon blockchains and FaB-DPSS with a threshold of $t/n < \frac{1}{2}$, we design eWEB – an efficient alternative to extractable witness encryption. It uses only standard cryptographic assumptions, while respecting prior impossibility results: Instead of resorting to expensive cryptographic machinery, it relies on interaction with a dynamic set of nodes with an honest majority. We believe this a favorable trade-off, as the honest majority setting has been repeatedly used in practice, most notably in blockchains [3]. For simplicity, in the following we will use the terms "dynamic set of nodes with an honest majority" and "blockchain" interchangeably, and the same for "nodes" and "miners". Roughly, we allow users to encode a secret along with a release condition. A predefined set of n nodes jointly and securely store the encoding and later *privately* release the secret to a user who demonstrably satisfies the release condition. We provide a formal proof of security of our construction, relying on the guarantees provided by the blockchain setting (specifically, we will assume a set of miners such that the majority of the selected miners are honest). As pointed out by Goyal and Goyal [GG17], one way to select such a set of miners is by selecting miners who were responsible for mining the last n blocks (where n is large enough). While it might seem like using secret sharing in combination with a blockchain directly provides a solution for conditional secret sharing, achieving a formally secure solution is subtle and requires careful design, as well as a few tricks we introduce in Section 5.2.

We note that the combination of blockchains and witness encryption has proven remarkably powerful. Indeed, Liu et al. [LJKW18] propose a time-lock encryption scheme that allows one to encrypt a message such that it can only be decrypted once a certain deadline has passed, without relying on trusted third parties or imposing high computational overhead on the receiver. The construction of Choudhuri et al. [CGJ+17] achieves fairness in multi-party computation

[3] Other instantiations are possible as well, see Section 5.1

against a dishonest majority. Goyal and Goyal [GG17] present the first construction for one-time programs (that run only once and then "self-destruct") that does not use tamper-proof hardware.

Using our DPSS- and blockchain based eWEB scheme, practitioners can easily implement all these applications. Note that many of them [LJKW18, CGJ+17, GG17] already rely on blockchains, implying that using eWEB does not add any additional assumptions. We explain in detail how a number of these applications can be achieved, and we implement and evaluate a few of them.

We also note that eWEB has already formed the basis of a follow-up work on non-interactive MPC [GMPS21].

1.1 Our results.

As explained above, one of our main contributions is a new and highly efficient DPSS scheme. More specifically, we achieve the following:

Theorem 1. *Assuming secure point-to-point channels and assuming that the t-SDH assumption holds, our construction satisfies the DPSS security definition (Definition 1) for a fully malicious adversary satisfying a corruption threshold $t < \frac{1}{2}n$, where n is the total number of parties holding the secrets. The adversary has the power to adaptively corrupt parties at any time. Our construction achieves amortized complexity of $O(n)$ and non-amortized complexity of $O(n^2)$.*

Here, Definition 1 is a DPSS security definition which is based on an ideal functionality, which behaves as follows:

- The functionality keeps track of the current committee.
- Upon receiving a secret storage request, the functionality stores the secret and notifies the current committee about the storage request.
- Upon receiving a release request from more than t (the adversarial threshold) number of parties in the current committee, the functionality either sends the corresponding secret to the client if the release request was private, or to the public otherwise.

Intuitively, this security definition requires that an adversary corrupting no more than the allowed threshold of parties does not learn any information about the secret through our protocol (secrecy) and cannot prevent an eligible party from learning the secret (robustness).

We present and formally prove secure FaB-DPSS in §3. We note that among the robust DPSS schemes which provide the highest-possible adversarial threshold of $t < \frac{1}{2}n$ our batched construction achieves the best amortized complexity – $O(n)$, while the state of the art CHURP [MZW+19] achieves $O(n^2)$ (see Figure 1 for comparison). Simultaneously, we achieve the same non-amortized complexity as CHURP – both works achieve $O(n^2)$. Our evaluation shows (see Figure 2 as well as Section 8.1) that FaB-DPSS outperforms CHURP in practice.

Next, we propose eWEB – a new cryptographic primitive which can be seen as a blockchain-based alternative to extractable witness encryption. We give a formal syntax for eWEB in §4.

Building upon FaB-DPSS and blockchains, we design and formally prove secure an eWEB construction. For this, we assume that an adversary controls some number of users and miners subject to the constraint that at any time the majority of miners who are eligible to participate in the protocol are honest (we explain how such a miner committee is chosen in §5.2). We assume that eWEB is a core functionality for the underlying blockchain. We use blockchain also for a PKI infrastructure and assume that each party has a unique identifier that is known to other parties. Finally, we assume authenticated IND-CCA secure encryption, collision-resistant hash functions and simulation extractable non-interactive zero knowledge proofs of knowledge.

We implement and evaluate our eWEB protocol (§7).

Finally, we explain how time-lock encryption, dead-man's switch, fair MPC, one-time programs and proofs of receipts can be achieved using eWEB (§6). As a more involved example, we propose an eWEB-based proof-of-concept voting protocol (§6.1). We implement and evaluate several of these applications (§8.3).

1.2 Related work

We elaborate on the prior work of both DPSS and conditional secret release.

Prior Work on DPSS Since the introduction of proactive secret sharing by Herzberg et al. [HJKY95], many PSS schemes have been developed. These schemes vary in terms of security guarantees, network assumptions (synchronous or asynchronous), communication complexity and whether they can handle dynamic changes in the committee membership. In Fig. 1 we provide a comparison.

Below, we compare FaB-DPSS in detail with the two constructions (Baron et al. [BDLO15], CHURP [MZW+19]) that are most closely related to our protocol, as they are also the most efficient robust DPSS schemes to date.

In the best case, CHURP [MZW+19] achieves communication complexity $O(n^2)$ plus $O(n) \cdot \mathcal{B}$ to refresh each secret in the hand-off phase, where n is the number of parties and $\mathcal{B}$ denotes the cost of broadcasting a bit. In the worst case (where some corrupted party deviates from the protocol), it requires $O(n^2) \cdot \mathcal{B}$ communication per secret.

In the single-secret setting, our protocol achieves the same asymptotic communication complexity as CHURP. However, our protocol achieves *amortized* communication complexity $O(n)$ plus $O(1) \cdot \mathcal{B}$ per secret in the best case, and $O(n^2)$ plus $O(n) \cdot \mathcal{B}$ per secret in the worst case. Batching is crucially important in eWEB since there may be thousands of secrets stored at any given time.

While CHURP uses asymmetric bivariate polynomials to refresh a secret during hand-off, we use a modified version of a technique of Damgård et al. [DN07] to prepare a batch of random secret sharings. Generating random secret sharings is much more efficient than generating bivariate polynomials. Specifically, each bivariate polynomial requires $O(n)$ communication per party; i.e., $O(n^2)$ in total. On the other hand, we can prepare $O(n)$ random sharings with the same communication cost as preparing one bivariate polynomial. To benefit from it, we use an entirely different way to refresh secrets.

The work of Baron et al. [BDLO15] focuses on a slightly different setting from ours: they consider unconditionally secure DPSS with a $(1/2-\epsilon)$ corruption threshold, where ϵ is a constant. Their scheme has $O(1)$ amortized communication per secret. However, in the single-secret setting, it requires $O(n^3)$ communication to refresh a secret. The authors use the party virtualization technique where every virtual party is simulated by a set of real parties running a maliciously secure MPC protocol. As discussed in [MZW$^+$19], a high constant is hidden in the big O notation. For example, if $\epsilon = 1/6$, i.e., the scheme is only secure with a $1/3$ corruption threshold, the virtualization requires to simulate at least 576 virtual parties with each using a set of 576 real parties running a maliciously secure MPC protocol, rendering the scheme very inefficient in practice.

Baron et al. use *packed* secret sharing [FY92] (in contrast to our *batched* secret sharing), which allows the same client to store $O(n)$ secrets in one sharing by encoding multiple secrets as distinct points of a single polynomial. Thus, refreshing each sharing effectively refreshes a batch of $O(n)$ secrets submitted by the same client at the same time. However, this means that secrets in the same batch come from a *single* client and can only be refreshed or reconstructed together. It is unclear if merging batches submitted by different clients is possible. Thus, even if in eWEB some secrets submitted by different clients were to be released at the same time, Baron et al.'s scheme would not allow us to join these secrets in one batch to profit from their low amortized communication complexity. Our scheme has one secret per sharing: only supplementary random sharings are generated in a batch. This allows to refresh (and release) each secret individually.

To reach $O(1)$ amortized communication per secret, Baron et al. need a $(1/2-\epsilon)$ corruption threshold. Our scheme does not suffer from this corruption threshold loss.

Extractable Witness Encryption and Conditional Secret Release The notion of witness encryption was introduced by Garg et al. [GGSW13]. Goldwasser et al. [GKP$^+$13] proposed extractable security for witness encryption. In their work a candidate construction was introduced that requires very strong assumptions over multilinear maps. According to Liu et al. [LJKW18], existing witness encryption schemes have no efficient extraction methods. Garg et al. [GGHW14] suggest that it even might be impossible to achieve extractable witness encryption with arbitrary auxiliary inputs.

Nevertheless, as mentioned in Sections 1 and 6, the notion of extractable witness encryption has been extensively used in various cryptographic protocols [CGJ$^+$17, GG17, BH15, LJKW18], especially in conjunction with blockchains.

Concurrently to our preprint, Benhamouda et al. [BGG$^+$20] published a manuscript that also proposes conditionally storing secrets on a blockchain. Unlike eWEB, their work is specific to proof-of-stake blockchains. Like us, they design a new DPSS scheme, but they target a very specific (albeit intriguing) use case — in their setting, the members of a committee must remain anonymous, even to the previous committee. They consider a stronger adversary who can corrupt and uncorrupt previously honest parties, but they only tolerate 25% corruption, versus 50% for our scheme. They do not explain how to release secrets without revealing witnesses to the miners. This is not trivial, as one would not want to release the secret publicly or allow an adversary to reuse an honest user's witness. Finally, they do not provide a formal security definition or implementation. Our work closes this gap.

Recently, a preprint by Gentry et al. [GHM+20] improved the adversarial threshold of Benhamouda et al., allowing it to tolerate $\frac{1}{2} - \epsilon$ corruptions – fewer than eWEB. While our amortized communication complexity is $O(n)$, the amortized complexity of Gentry et al. (building upon Benhamouda et al.'s work) is an unspecified polynomial. Their setting and focus is different from ours.

Kokoris-Kogias et al. proposed Calypso [KKAS+18], a verifiable data management solution that relies on blockchains and threshold encryption. Calypso targets a different use case than our eWEB system: it allows verifiable sharing of data to parties that are explicitly authorized (either by the depositor or by a committee of authorized parties) to have access rather than specifying a general secret release condition that allows anyone who is able to satisfy this condition to get the data. Kokoris-Kogias et al. do not provide a formal security definition or formal security proof of their system. The major part of their work focuses on the static committee of parties holding the secrets; the dynamic committee setting is only discussed very briefly.

eWEB could be seen as a special case of proactive secure multi-party computation (PMPC) [OY91, BEDLO14, EOPY18]. However, while our DPSS scheme could be used for PMPC, eWEB targets a different use case than general PMPC. This allows for a much more efficient and largely non-interactive construction compared to PMPC protocols, which typically have very high round complexity.

2 Preliminaries

In this section, we introduce the DPSS security definition. In the interest of space, we introduce further building blocks in the full version of this paper [GKM+20].

2.1 DPSS Security Definition

A dynamic proactive secret-sharing scheme (DPSS) scheme allows a client to distribute shares of a secret to n parties, so that an adversary in control of some threshold number of parties t learns no information about the secret. The set of parties holding secrets is constantly changing, and the adversary can "release" some parties (users regain control of their systems) and corrupt new ones.

A DPSS scheme consists of the following three phases.

Setup. In each setup phase, one or more independent clients secret-share a total of m secrets to a set of n parties, known as a committee, denoted by

$\mathcal{C} = \{P_1, \ldots, P_n\}$. After each setup phase, each committee member holds one share for each secret s distributed during this phase.

Hand-off. As the protocol runs, the hand-off phase is periodically invoked to provide the new committee with updated shares in such a way that the adversary cannot use information from multiple committees to learn anything about the secret. This process reflects parties leaving and joining the committee. After the hand-off phase, all parties in the old committee delete their shares, and all parties in the new committee hold a sharing for each secret s. The hand-off phase is particularly challenging, since during the hand-off a total of $2t$ parties may be corrupted (t parties in the old committee and t parties in the new committee).

Reconstruction. When a client (which need not be one who stored the secret) asks for the secret reconstruction, that client and the current committee engage in a reconstruction process to allow the client learn the secret.

At a high-level, the security of the DPSS scheme requires that it should always be possible to recover the secret, and an adversary should not learn any further information about the secret beyond what has been learned before running the protocol. We formally model the security in $\mathsf{Ideal}_{\text{safe}}$. Note that we slightly generalize the typical DPSS definition by supporting not only private release to a client, but also a public release of the secret. In this case, the secret is broadcast to all parties.

Ideal Secrecy: $\mathsf{Ideal}_{\text{safe}}$

1. $\mathsf{Ideal}_{\text{safe}}$ receives a list $\mathcal{C}$ of parties as the first committee, and a corruption threshold t. $\mathsf{Ideal}_{\text{safe}}$ initializes an empty list $\mathcal{L}$ for the secrets.
2. Upon receiving a storage request (store, s) from a client, $\mathsf{Ideal}_{\text{safe}}$ adds the secret s to the end of list $\mathcal{L}$, and sets the identifier id of the secret s to be the number of secrets in $\mathcal{L}$. $\mathsf{Ideal}_{\text{safe}}$ sends id to all parties in the current committee.
3. Upon receiving $(\mathsf{change\text{-}committee}, \mathcal{C}')$ from more than t parties in the current committee $\mathcal{C}$, $\mathsf{Ideal}_{\text{safe}}$ changes the committee to the parties in $\mathcal{C}'$ and sends the identities of $\mathcal{C}'$ to all the parties in $\mathcal{C}$.
4. Upon receiving $(\mathsf{release}, \mathsf{id}, \mathsf{client})$ from more than t parties in the current committee, $\mathsf{Ideal}_{\text{safe}}$ scans list $\mathcal{L}$ for the secret $s^\star$ that corresponds to the identifier id. If such a secret does not exist, $\mathsf{Ideal}_{\text{safe}}$ sets $s^\star = \perp$.
   - If $\mathsf{client} \neq \mathsf{public}$, $\mathsf{Ideal}_{\text{safe}}$ sends $s^\star$ to the client.
   - Otherwise, $\mathsf{Ideal}_{\text{safe}}$ broadcasts $s^\star$.

Definition 1. *A dynamic proactive secret-sharing scheme is secure if for any PPT adversary $\mathcal{A}$ and threshold t, there exists a simulator $\mathcal{S}$ with access to*

Ideal_{safe} *(described in Ideal Secrecy), such that the view of $\mathcal{A}$ interacting with $\mathcal{S}$ is computationally indistinguishable from the view in the real execution.*

2.2 DPSS Definition Discussion

Our definition is slightly different from the original definition of PSS [OY91]. In that definition, there is an additional "Recovery" phase where a party infected by a virus reboots itself to remove the virus and recovers its share jointly with all other parties. In our definition, however, we assume that a party regains control automatically when the adversary releases it, and such a party can use fresh randomness afterwards. To adapt to the original definition where a reboot is needed for a party to remove the virus, this party backs up its share and reboots itself before the next handoff phase. The backup guarantees that an honest party does not lose its shares during the reboot. If this party is corrupted before the reboot, then since the handoff phase will generate a new sharing for all parties, there is no need to recover its old share.

3 Technical Overview – FaB-DPSS

In the following section we give an overview of our FaB-DPSS scheme and security proof. We give the entire construction in the full version of this paper.

FaB-DPSS is based on Shamir Secret Sharing [Sha79]. In the following, we assume the corruption threshold for each committee is fixed to t. We use $[x]_d$ to denote a degree-d sharing, i.e., $(d+1)$-out-of-n Shamir sharing. It requires at least $d+1$ shares to reconstruct the secret, any d or fewer shares leak no information about the secret. Note that Shamir's scheme is additively homomorphic.

In the following, first we outline the adversarial model. Then we discuss the hand-off phase of our scheme in the semi-honest case (§3.1) and explain how it can be modified for the fully malicious case (§3.2). We solve the semi-honest case through the introduction of the techique we dub "coupled sharings" combined with the careful use of ideas from the MPC literature [DN07] and a few additional tricks which allow us to achieve good amortized communication complexity. For the fully malicious case, unlike in the MPC world, we must marry these techniques with polynomial commitment schemes. We present the setup phase (§3.3) as a special case of our hand-off phase, summarize our reconstruction phase (§3.4), and provide intuition for our construction's security proof (§3.5).

Adversary Model We consider a computationally bounded fully malicious adversary $\mathcal{A}$ with the power to adaptively choose parties to corrupt at any time. $\mathcal{A}$ can corrupt any number of clients distributing secrets and learn the secrets held by the corrupted clients. For each committee $\mathcal{C}$ with a threshold $t < |\mathcal{C}|/2$, $\mathcal{A}$ can corrupt at most t parties in $\mathcal{C}$. When a party P_i is corrupted by $\mathcal{A}$, $\mathcal{A}$ fully controls the behavior of P_i and can modify P_i's memory state. Even if $\mathcal{A}$ releases its control of P_i, its memory may have already been modified; e.g., P_i's share might have been erased.

For a party P_i in both the old committee $\mathcal{C}$ and the new committee $\mathcal{C}'$, if $\mathcal{A}$ has the control of P_i during the hand-off phase, then P_i is considered to be corrupted in both committees. If $\mathcal{A}$ releases its control before the hand-off phase in which the secret sharing is passed from $\mathcal{C}$ to $\mathcal{C}'$, then P_i is only considered corrupted in the old committee $\mathcal{C}$. If $\mathcal{A}$ only corrupts P_i after the hand-off phase, P_i is only considered corrupted in the new committee $\mathcal{C}'$.

For simplicity, in the following, we assume that there exist secure point-to-point channels between the parties and the corruption threshold is a fixed value t. Our protocol can be easily adapted to allow different thresholds for different committees (see the full version of this paper).

3.1 FaB-DPSS: Semi-honest Case

We first explain the high-level idea of our protocol in the semi-honest setting; i.e., all parties honestly follow the protocol. The foundational idea in FaB-DPSS is the introduction of so-called *coupled sharings*. By this, we mean two sharings $([r]_t, [\tilde{r}]_t)$ which have the same *value* $(r = \tilde{r})$, even though the particular shares which lead to this value are different for the two sharings. Now, imagine a coupled sharing $([r]_t, [\tilde{r}]_t)$ of a uniformly random value r. We let $[r]_t$ be held by the old committee, and $[\tilde{r}]_t$ be held by the new committee. Suppose the secret sharing we want to refresh is $[s]_t$, held by the old committee. Then the old committee will compute the sharing $[s+r]_t = [s]_t + [r]_t$ and reconstruct the secret $s+r$. Since r is uniformly random, $s+r$ does not leak any information about s. Now, the new committee can compute $[\tilde{s}]_t = (s+r) - [\tilde{r}]_t$. Since $\tilde{r} = r$, we have $\tilde{s} = s$. This whole process is split into *preparation* and *refresh* phases:

- In the preparation phase, parties in the new committee prepare the coupled sharing: two degree-t sharings of the same random value $r(= \tilde{r})$, denoted by $[r]_t$ and $[\tilde{r}]_t$. The old committee receives the shares of $[r]_t$ and the new committee holds the shares of $[\tilde{r}]_t$.
- In the refresh phase, the old committee reconstructs the sharing $[s]_t + [r]_t$ and publishes the result. The new committee sets $[\tilde{s}]_t = (s+r) - [\tilde{r}]_t$.

The rest of our protocol builds around this idea. We need to solve the following challenges:

- How can committees prepare the coupled sharings?
- How can this preparation step be done as efficiently as possible?
- How can these sharings be used efficiently during the refresh step?

We start by answering the first two questions. In the following, let $\mathcal{C}$ denote the old committee and $\mathcal{C}'$ denote the new committee. Intuitively, the straw man solution which allows to obtain one coupled sharing is the following:

1. Each party $P_i' \in \mathcal{C}'$ prepares a coupled sharing $([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$ of a random value and distributes $[u^{(i)}]_t$ to the old committee and $[\tilde{u}^{(i)}]_t$ to the new committee.
2. All parties in the old committee compute $[r]_t = \sum_{i=1}^{n} [u^{(i)}]_t$. All parties in the new committee compute $[\tilde{r}]_t = \sum_{i=1}^{n} [\tilde{u}^{(i)}]_t$.

Since for each i, $u^{(i)} = \tilde{u}^{(i)}$, we have $r = \tilde{r}$.

Unfortunately, this way of preparing coupled sharings is wasteful since at least $(n-t)$ coupled sharings are generated by honest parties, which appear uniformly random to corrupted parties. In order to get $(n-t)$ random coupled sharings instead of just 1, we borrow an idea from Damgård and Nielsen [DN07].

In their work, parties need to prepare a batch of random sharings which will be used in an MPC protocol. All parties first agree on a fixed and public Vandermonde matrix $\boldsymbol{M}^{\mathrm{T}}$ of size $n \times (n-t)$. An important property of a Vandermonde matrix is that any $(n-t) \times (n-t)$ submatrix of $\boldsymbol{M}^{\mathrm{T}}$ is *invertible*. To prepare a batch of random sharings, each party P_i generates and distributes a random sharing $[u^{(i)}]_t$. Next, all parties compute

$$([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(n-t)}]_t)^{\mathrm{T}} = \boldsymbol{M}([u^{(1)}]_t, [u^{(2)}]_t, \ldots, [u^{(n)}]_t)^{\mathrm{T}},$$

and take $[r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(n-t)}]_t$ as output. Since any $(n-t)\times(n-t)$ submatrix of $\boldsymbol{M}$ is invertible, given the sharings provided by corrupted parties, there is a one-to-one map from the output sharings to the sharings distributed by honest parties. Since the input sharings of the honest parties are uniformly random, the one-to-one map ensures that the output sharings are uniformly random as well [DN07].

Note that any linear combination of a set of coupled sharings is also a valid coupled sharing. Thus, in our protocol, instead of computing $([r]_t, [\tilde{r}]_t) = \sum_{i=1}^{n}([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$, parties in the old committee can compute

$$([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(n-t)}]_t)^{\mathrm{T}} = \boldsymbol{M}([u^{(1)}]_t, [u^{(2)}]_t, \ldots, [u^{(n)}]_t)^{\mathrm{T}}$$

and parties in the new committee can compute

$$([\tilde{r}^{(1)}]_t, [\tilde{r}^{(2)}]_t, \ldots, [\tilde{r}^{(n-t)}]_t)^{\mathrm{T}} = \boldsymbol{M}([\tilde{u}^{(1)}]_t, [\tilde{u}^{(2)}]_t, \ldots, [\tilde{u}^{(n)}]_t)^{\mathrm{T}}$$

Now all parties get $(n-t)$ random coupled sharings. The amortized communication cost per such sharing is $O(n)$.

We now describe the *refresh phase*. For each sharing $[s]_t$ of a client secret which needs to be refreshed, one random coupled sharing $([r]_t, [\tilde{r}]_t)$ is consumed. Parties in the old committee first select a special party $P_{\texttt{king}}$. To reconstruct $[s]_t + [r]_t$, parties in the old committee locally compute their shares of $[s]_t + [r]_t$, and then send the shares to $P_{\texttt{king}}$. Then, $P_{\texttt{king}}$ uses these shares to reconstruct $s+r$ and publishes the result. Finally, parties in the new committee can compute $[\tilde{s}]_t = (s+r) - [\tilde{r}]_t$.

3.2 Moving to a Fully-Malicious Setting

In a fully-malicious setting, three problems might arise.

- During preparation, a party distributes an inconsistent degree-t sharing or incorrect coupled sharing.
- During refresh, a party provides an incorrect share to $P_{\texttt{king}}$, causing a reconstruction failure.

- $P_{\texttt{king}}$ provides an incorrectly reconstructed value.

We address these problems by checking the correctness of coupled sharings in the preparation phase and relying on polynomial commitments to transform a plain Shamir secret sharing into a verifiable one.

Checking the Correctness of Coupled Sharings. While it is possible to check the correctness of each coupled sharing separately, we can increase efficiency by utilizing the following trick: check if *all* of them are correct by checking their *random linear combination*. It works since any linear combination of coupled sharings is also a valid coupled sharing.

Note that in the process we need to to protect the privacy of every coupled sharing $([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$ generated by a party P_i'. We achieve it by having P_i' generate one additional random coupled sharing as a random mask, which is denoted by $([\mu^{(i)}]_t, [\tilde{\mu}^{(i)}]_t)$.

Consider the following two sharings of polynomials of degree-$(2n-1)$:

$$[F(X)]_t = \sum_{i=1}^{n}([\mu^{(i)}]_t + [u^{(i)}]_t \cdot X)X^{2(i-1)},$$
$$[\tilde{F}(X)]_t = \sum_{i=1}^{n}([\tilde{\mu}^{(i)}]_t + [\tilde{u}^{(i)}]_t \cdot X)X^{2(i-1)}.$$

These two sharings have the following benefitial properties:

1. If *all* coupled sharings are correct, then $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$ is also a coupled sharing for any λ. Otherwise, the number of λ such that $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$ is a coupled sharing is bounded by $2n-1$. Thus, in order to test whether all sharings are correct, it is sufficient to test $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$ at a random evaluation point λ.
2. The coupled sharing $([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$ generated by P_i' is masked by a random coupled sharing $([\mu^{(i)}]_t, [\tilde{\mu}^{(i)}]_t)$ which is also generated by P_i'. Thus, the secrecy of $([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$ is preserved during the check of $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$.

Therefore, we first let all parties generate a random challenge λ. Parties in the old committee compute $[F(\lambda)]_t$ and publish their shares. Parties in the new committee compute $[\tilde{F}(\lambda)]_t$ and publish their shares. Finally, all parties check whether $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$ is a valid coupled sharing.

If the check fails (not all sharings are correct), we need to pinpoint parties who distributed incorrect coupled sharings. Since each coupled sharing $([u^{(i)}]_t, [\tilde{u}^{(i)}]_t)$ is masked by $([\mu^{(i)}]_t, [\tilde{\mu}^{(i)}]_t)$, it is safe to open the whole sharing $([\mu^{(i)}]_t + [u^{(i)}]_t \cdot \lambda, [\tilde{\mu}^{(i)}]_t + [\tilde{u}^{(i)}]_t \cdot \lambda)$ and check whether it is a valid coupled sharing. Since $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t)$ is a linear combination of the coupled sharings $\{([\mu^{(i)}]_t + [u^{(i)}]_t \cdot \lambda, [\tilde{\mu}^{(i)}]_t + [\tilde{u}^{(i)}]_t \cdot \lambda)\}_{i=1}^{n}$, at least one coupled sharing of $\{([\mu^{(i)}]_t + [u^{(i)}]_t \cdot \lambda, [\tilde{\mu}^{(i)}]_t + [\tilde{u}^{(i)}]_t \cdot \lambda)\}_{i=1}^{n}$ is inconsistent. In fact, we can find all inconsistent coupled sharings with overwhelming probability with the help of polynomial commitments introduced later. For each i, parties in the old committee compute $[\mu^{(i)}]_t + [u^{(i)}]_t \cdot \lambda$ and publish their shares, and parties in the new committee compute $[\tilde{\mu}^{(i)}]_t + [\tilde{u}^{(i)}]_t \cdot \lambda$ and publish their shares. This way, we can tell which coupled sharings are inconsistent. This inconsistency in the coupled sharing distributed by some party P_i' (in the following, *dealer*) has two possible causes:

- The dealer P_i' distributed an invalid coupled sharing (either the secrets were not the same or one of the degree-t sharings was invalid).

- Some corrupted party $P_j \in C \cup C'$ provided an incorrect share during the verification of the sharing distributed by the dealer P'_i.

The first case implies that the dealer is a corrupted party. To distinguish the first case from the second, we will rely on *polynomial commitments*, which can be used to transform a plain Shamir secret sharing into a verifiable one so that an incorrect share (e.g., in case 2) can be identified and rejected by all parties.

Relying on Polynomial Commitments. A degree-t Shamir secret sharing corresponds to a degree-t polynomial $f(\cdot)$ such that: **(a)** the secret is $f(0)$, and **(b)** the i-th share is $f(i)$. Thus, each dealer can commit to f by using a polynomial commitment scheme to add verifiability.

A polynomial commitment scheme allows the dealer to open one evaluation of f (which corresponds to one share of the Shamir secret sharing) and the receiver can verify the correctness of this evaluation value. Essentially, whenever a dealer distributes a share it also provides a *witness* which can be used to verify this share. Informally, a polynomial commitment scheme satisfies three properties:

- Polynomial Binding: A commitment cannot be opened to two different polynomials.
- Evaluation Binding: A commitment cannot be opened to two different values at the same evaluation point.
- Hiding: A commitment should not leak any information about the committed polynomial.

We use polynomial commitments as follows: in the beginning, each dealer first commits to the sharings it generated and opens the shares to corresponding parties. To ensure that each party is satisfied with the shares it received, there is a following accusation-and-response phase:

1. Each party publishes $(\texttt{accuse}, P'_i)$ if the share received from P'_i does not pass the verification algorithm.
2. For each accusation made by P_j, P'_i opens the j-th share to all parties, and P_j uses the new share published by P'_i if it passes the verification. Otherwise, P'_i is regarded as a corrupted party by everyone else.

Note that an honest party will never accuse another honest party. Also, if a malicious party accuses an honest party, no more information is revealed to the adversary than what the adversary knew already. Thus, it is safe to reveal the share sent from P'_i to P_j. *After this step, all parties should always be able to provide valid witnesses for their shares.*

Recall that parties need to do various linear operations on the shares. In FaB-DPSS we use the KZG commitment scheme [KZG10], which is linearly homomorphic. Thus, even if a share is a result of a number of linear operations, it is still possible for a party to compute the witness for this share. From now on, each time a party sends or publishes a share, this party also provides *the associated witness* to allow other parties verify the correctness of the share. Since honest parties will always provide shares with valid witnesses and there are at least $n - t \geq t + 1$ honest parties, all parties will only use shares that pass verification. Intuitively, this solves the problem of incorrect shares provided by corrupted parties since corrupted parties cannot provide valid witnesses for those shares. Similarly, it should solve the problem of a malicious $P_{\texttt{king}}$, since he cannot provide a valid witness for the incorrectly reconstructed value. However, due to a subtle limitation of the KZG commitment scheme, we actually need to add an additional minor verification step (see [GKM+20] for details).

See [GKM+20] for a complete description of the hand-off process.

3.3 FaB-DPSS Setup Phase

The setup phase uses a similar approach to the hand-off phase. First, the committee prepares random sharings. As in the hand-off phase, the validity of the distributed shares is verified using the KZG commitment scheme. For each secret s distributed by a client, one random sharing $[r]_t$ is consumed. The client receives the whole sharing $[r]_t$ from the committee and reconstructs the value r. Finally, the client publishes $s+r$. The committee then computes $[s]_t = s+r-[r]_t$. See [GKM+20] for details.

3.4 FaB-DPSS Reconstruction Phase

When a client asks for the reconstruction of some secret $s^\star$, all parties in the current committee simply send their shares of $[s^\star]_t$ and the associated witnesses to the client. The client then reconstructs the secret using the first $t+1$ shares that pass the verification checks. See [GKM+20] for details.

3.5 Security of Our Construction

We give a high-level idea of our proof. The goal is to construct a simulator to simulate the behaviour of honest parties. For each sharing, corrupted parties receive at most t shares, which are independent of the secret. Thus, when an honest party needs to distribute a random sharing, the simulator can send random elements to corrupted parties as their shares without fixing the shares of honest parties. Since we use the perfectly hiding variant of the KZG commitment, the commitment is independent of the secret, and can be generated using the trapdoor of the KZG scheme. Furthermore, we can adaptively open t shares chosen by the adversary after the commitment is generated. This makes our scheme secure against adaptive corruptions. We present the full formal security proof of our scheme in [GKM+20].

4 DPSS Applications – eWEB Primitive

Our next goal is to expand the reach of DPSS. We ask the following question:

> *Is it possible to let users store secrets and specify release conditions for these secrets in a way that allows (other) users to retrieve these secrets later on if and only if they are able to satisfy the release condition?*

Our goal is to achieve this without relying on trusted third parties. Instead, we imagine a distributed storage of secrets which would allow for a high adversarial threshold. We refine our question as follows:

> *Is it possible to let users store secrets with some group of parties and specify release conditions for these secrets in a way that allows (other) users to retrieve these secrets later on if and only if they are able to satisfy the release condition? Furthermore, is it possible to achieve this if the adversary is able to corrupt up to $t < \frac{1}{2}n$ number of parties storing the secrets?*

We are able to answer these questions positively by utilizing DPSS and a dynamic set with honest majority, PKI, and authenticated broadcast. We utilize blockchains as a real-world system which provides the latter three primitives. While we emphasize that technically our solution can be based on any other set of parties with honest majority (supplemented with a PKI and authenticated broadcast), for ease of exposition, in the following we will use "dynamic set with honest majority" and "blockchains" interchangeably.

We now formally introduce the extractable witness encryption on a blockchain (*eWEB*) primitive. We distinguish between users who deposit secrets (depositors), users who request that a secret be released (requesters), and a changing set of blockchain nodes (miners) who are executing these requests.

An eWEB system consists of the following, possibly randomized and interactive, subroutines:

$\mathit{SecretStore}(M, F) \rightarrow (\mathit{id}, \{\mathit{frag}_1, .., \mathit{frag}_n\}, F)$: A depositor stores a secret M which can be released to a requester who knows a witness w s.t. $F(w)$ is true. After interacting with the depositor, each of the n miners obtains a "fragment", frag_i, of the secret that is associated with the secret storage request with the identifier id.

$\mathit{SecretsHandoff}(\{\mathit{frag}_1^1, .., \mathit{frag}_n^1\}, .., \{\mathit{frag}_1^m, .., \mathit{frag}_n^m\}) \rightarrow$ $(\{\widetilde{\mathit{frag}}_1^1, .., \widetilde{\mathit{frag}}_n^1\}, .., \{\widetilde{\mathit{frag}}_1^m, .., \widetilde{\mathit{frag}}_n^m\})$: Miners periodically execute this function to hand over all m stored secrets from the old committee to the new committee. Each miner i of the old committee possesses m fragments (one for each secret) $\mathit{frag}_i^1, .., \mathit{frag}_i^m$ at the start of the hand-off protocol. Each miner i of the new committee possesses m fragments (one for each secret) $\widetilde{\mathit{frag}}_i^1, .., \widetilde{\mathit{frag}}_i^m$ at the end of the protocol.

$\mathit{SecretRelease}(\mathit{id}, w) \rightarrow M$ or $\perp$: A requester uses this function to request the release of the secret with the identifier id. The requester specifies the witness w to the release condition. Miners check whether the requester holds a valid witness and if so, as a result of the interaction with the miners, the requester obtains the secret M. Otherwise the function returns $\perp$ (i.e., attempt failed).

Security Definition We provide a formal game-based secrecy definition in [GKM+20]. Practically, this definition states that if an adversary is able to distinguish between the protocol executed with secret M_0 and the protocol executed with secret M_1, then we can extract a valid witness for the release condition F

using this adversary. Intuitively, this notion is quite similar to the extractable security of witness encryption, which states that if an adversary can distinguish between two ciphertexts, then he can also extract a witness from the corresponding problem instance. For robustness, intuitively we require that it is always possible for an honest requester to reconstruct a secret dealt by an honest depositor.

Remark 1. We also propose a variant of eWEB with a slightly relaxed security notion we dub *Public Witness* security. Here, the secret is made public after a single successful secret release. As we show in Section 6, this notion proves quite useful in a number of applications.

5 Our eWEB Protocol Design

Before we introduce our eWEB construction, we provide an overview of the assumptions that we rely on in our scheme (§5.1).

5.1 Assumptions

Adversary model. We rely on blockchains and assume that eWEB is a core functionality, which allows us to focus on the fundamental construction without worrying about selfish mining or bribery attacks. The adversary is able to control a polynomial number of users and miners, subject to the constraint that the blockchain has $(\frac{n}{2}+1, n)$-chain quality, meaning that for each n or more continuous blocks mined in the system, more than half were mined by honest parties. As noted by [GG17], for proof of work blockchains, where the probability of successful mining is proportional to the amount of computational power, this assumption follows from the assumption that honest miners possess the majority of the computational power in the system. We assume this majority is "significant enough" (to, for example, defeat selfish mining attacks [ES14] that would threaten Bitcoin's security). For proof of stake blockchains, where the probability of successful mining is proportional to the amount of coins possessed by the miner, it follows from the assumption that honest miners possess the majority of stake in the system. In practice, we pick an n that is big enough to provide this property with only a very small error probability. Honest majority assumptions are very common in the blockchain space [GG17, CGJ$^+$17, MZW$^+$19, GHM$^+$17, KKAS$^+$18], especially in permissioned blockchains, which often rely on BFT replication protocols, which in turn usually assume an honest supermajority [ABB$^+$18]. We assume that the blockchain is an append-only log, and it is hard to modify or erase its contents.

We assume that once an adversary corrupts a party it remains corrupted. The adversary cannot adaptively corrupt previously honest parties. When a party is corrupted by the adversary, the adversary fully controls this party's behaviour and internal memory state. We do not distinguish between adversarial and honest parties who behave maliciously unintentionally; e.g., those who have connection issues and cannot access the blockchain to participate.

Infrastructure model. It is common for public keys to be known in blockchains. We require that additionally each party p_i has a *unique identifier*, denoted by pid_i, that is known to all other parties. In practice, this identifier can be the hash of the party's public key. For simplicity, we present the scheme as if there were authenticated channels between all parties in the system. In practice, these channels can be realized using standard techniques such as signatures.

Communication model. Our DPSS scheme assumes secure point-to-point communication channels. In the decentralized blockchain setting of eWEB we prefer not to make such an assumption, since using point-to-point channels could compromise nodes' anonymity and lead to targeted attacks [MZW+19]. Instead, we assume that parties communicate via an existing blockchain. We distinguish between posting a message on the blockchain (expensive) and using the blockchain's peer-to-peer network for broadcast (cheap). Point-to-point channels can be simulated using IND-CCA secure encryption and broadcasting the ciphertexts.

Storage. We assume that, in addition to parties' internal storage, there exists some publicly accessible off-chain storage that is cheaper than on-chain one. Thus, we store data off-chain and save only data *hashes* on-chain. Our system's robustness depends on the robustness of the off-chain storage. Thus, storage systems with a reputation for high availability should be chosen. However malicious off-chain storage does not impact the secrecy properties of our system (see [GKM+20]. Alternatively, at a higher cost, we can use on-chain storage for everything.

Cryptographic assumptions. In addition to the assumptions outlined above, we assume IND-CCA secure encryption, collision-resistant hash functions, simulation extractable non-interactive zero knowledge proofs of knowledge, and (for DPSS) the t-SDH assumption.

Blockchain setting. eWEB can be built atop of any node set with honest majority supplemented by a PKI. We present eWEB in the blockchain setting simply because it is a system which already exists in practice (and because multiple applications which rely on extractable witness encryption and which we try to achieve already rely on blockchains [LJKW18, CGJ+17, GG17]).

We note that miners that behave honestly w.r.t. the blockchain protocol might need further incentivization to behave honestly w.r.t. eWEB; otherwise they might try to disrupt the execution of the eWEB protocol or leak their secret shares. Our DPSS scheme has numerous checks that identify parties disrupting correct protocol execution (see §3.2), which could translate to economic disincentivization. Traitor-tracing secret sharing [GSS21] as well as trusted hardware that can verify correct share deletion could be used as mechanisms that ensure that miners are punished for leaking secrets entrusted to them. We leave exploring these directions for future work.

5.2 Our eWEB Construction

We now describe our eWEB scheme. Its key building block is a DPSS scheme used in a black-box way. The initial committee are miners who mined the most recent n blocks in the underlying blockchains.

Given a secret message M and a release condition F, the depositor stores the release condition F on the blockchain and secret-shares M among the miners using the secret storage (setup) algorithm of the DPSS scheme.

During the protocol's periodically executed hand-off phase, the secrets are passed from the miners of the old committee to the miners of the new committee using the DPSS hand-off algorithm. The new committee consists of the miners who mined the most recent n blocks. This keeps the size of the committee constant and allows all parties to determine the current committee by looking at the blockchain state. It is possible that some committee members receive more information about the secrets than others — roughly, if a party mined m out of the last n blocks, this party receives $\frac{m}{n}$ of all the shares. This reflects the distribution of the computing power (for POW blockchains) or stake (for POS blockchains) in the system [GG17].

To retrieve a stored secret, a requester U needs to prove that they are eligible to do so. This poses a challenge. An insecure solution is to just send a valid witness w ($F(w) = \mathit{true}$) to the miners. One obvious problem with this solution is that a malicious miner can use the provided witness to construct a new secret release request and retrieve the secret himself. To solve this problem, instead of sending the witness in clear, the user *proves that they know a valid witness*. Thus, while the committee members are able to check the validity of the request and privately release the secret to U, the witness remains hidden. In our scheme we rely on non-interactive zero knowledge proofs (NIZKs) [BFM88]. Such proofs allow one party (the prover) to prove validity of some statement to another party (the verifier), such that nothing except for the validity of the statement is revealed. In eWEB we specifically use simulation extractable non-interactive zero knowledge *proofs of knowledge*, which allow the prover convince the verifier that they know a witness to some statement. Note that extractability can be added to any NIZK [ARS20, KZM+15]. We use NIZKs for relation $R = \{(pk, w) \mid F(w) = \mathit{true} \text{ and } pk = pk\}$, where $F(\cdot)$ is the release condition specified by the depositor and pk is the public key of user U and is used to identify the user eligible to receive the secret. After the miners verify the validity of the request, they engage in the DPSS's secret reconstruction with requester U to release the secret to U.

We provide the full secret storage protocol in Figure 3. The hand-off protocol is given in Figure 4. The secret release protocol is in Figure 5. Note that the asymptotics of eWEB match those of our underlying DPSS scheme. Below, we elaborate on additional details of our construction.

Subtleties of Point-to-Point Channels As mentioned in §5.1, while FaB-DPSS assumes secure point-to-point channels, we do not make such an assumption in eWEB. Instead, we rely on authenticated encryption and Protocols 1 and 2, executed whenever a message needs to be securely sent from one party to another. It is used for all messages exchanged in eWEB, including the underlying DPSS protocol. Whenever a party receives an encrypted message, it performs an authentication check to ensure that a ciphertext received from some party was generated by that party. This prevents the following malleability issue - a malicious user desiring to learn a secret with the identifier id could generate a new secret storage request with a function $\tilde{F}$ for which he knows a witness, copy the DPSS messages sent by the user who created the storage request id to the miners and later on prove his knowledge of a witness for $\tilde{F}$ to release the corresponding secret. Without the authentication check, our scheme would be insecure, and our security proof (see [GKM+20]) would not go through.

Protocol 1 MessagePreparation

1. For a message m to be sent by party P_s to party P_r, P_s computes the ciphertext $c \leftarrow Enc_{pk_r}(m|pid_s)$, where pk_r is the public key of P_r and pid_s is the party identifier of P_s.
2. P_s prepends the storage identifier id of his request and sends the tuple (id, c) to P_r.

Protocol 2 AuthenticatedDecryption

1. Upon receiving a tuple (id, c) from party P_s over an authenticated channel, the receiving party P_r decrypts c using its secret key sk to obtain $m \leftarrow Dec_{sk}(c)$.
2. P_r verifies that m is of the form $m'|pid_s$ for some message m', where pid_s is the identifier of party P_s.
3. If the verification check fails, P_r stops processing c and outputs an error message.

Storage Identifiers Each storage request has a unique identifier id. This can be, e.g., the address of this particular transaction in the blockchain. It is used for practical reasons, and is not relevant for the security of our construction.

Handling Large Secrets Since the secret itself might be very large, it is also possible to first encrypt the secret using a symmetric encryption scheme, store the ciphertext publicly off chain and then secret-share the symmetric key instead. Also, we store request parameters (such as release conditions or proofs) off-chain, saving only the hash of the message on-chain.

5.3 Security Proof Intuition

We provide a formal proof of security in [GKM+20], showing that our scheme satisfies the security definition for eWEB given in [GKM+20]. In this proof, we rely on the zero-knowledge and simulation-sound extractability properties

Protocol 3 SecretStore

1. The depositor executes NIZK's KeyGen protocol to obtain a CRS: $\sigma \leftarrow \mathsf{KeyGen}(1^k)$.
2. The depositor computes hash $\mathsf{requestHash} \leftarrow H(F|\sigma)$, and publishes $\mathsf{requestHash}$ on the blockchain. Let id be the storage identifier of the published request.
3. The depositor stores the tuple $(id, F|\sigma)$ offchain.
4. The depositor and the current members of the miner committee engage in the DPSS **Setup Phase**.
5. Each committee member retrieves $\mathsf{requestHash}$ from the blockchain, $F|\sigma$ from the offchain storage, and verifies that $\mathsf{requestHash}$ is indeed the hash of $F|\sigma$:

   $$\mathsf{requestHash} \stackrel{?}{=} H(F|\sigma)$$

   If this is not the case, the committee member aborts.
6. C_i stores $(id, \mathsf{dpss\text{-}data}_i)$ internally, where $\mathsf{dpss\text{-}data}_i$ is the data obtained from the DPSS **Setup Phase**.

Protocol 4 SecretsHandoff

1. For each secret storage identifier id, the miners of the old and the new committee engage in the DPSS **Handoff Phase** for the corresponding secret. Let $\mathsf{dpss\text{-}data}_i^{id}$ denote the resulting DPSS data corresponding to the storage identifier id of party C_i of the new committee after the handoff phase.
2. For each secret storage identifier id, each miner of the new committee stores $(id, \mathsf{dpss\text{-}data}_i^{id})$ internally.

of the NIZK scheme to switch from providing honest proofs to using simulated proofs. Next, we rely on the collision-resistance of the hash function to show that any modification of the data stored offchain will be detected. Then, we rely on the multi-message IND-CCA security of the encryption scheme to change all encrypted messages exchanged between honest parties to encryptions of zero. Finally, we rely on the security of our DPSS scheme to switch from honestly executing the DPSS protocol to using a DPSS simulator. At this point, we can show that either the adversary was able to provide a valid secret release request for the challenge's secret-release function, in which case we are able to extract a witness from the provided NIZK proof (relying on the NIZK's proof-of-knowledge property), or the adversary did not provide a valid secret release request and in this case we are able to "forget" the secret altogether, since it is never used.

Protocol 5 SecretRelease

1. To request the release of a secret with identifier id, the requester retrieves requestHash from the blockchain, $F|\sigma$ from off-chain storage, and verifies that requestHash is indeed the hash of $F|\sigma$:

$$\mathsf{requestHash} \overset{?}{=} H(F|\sigma)$$

   If this is not the case, the requester aborts.
2. The requester computes a NIZK proof of knowledge of the witness for F and his identifier p_{id}:

$$\pi \leftarrow P(\sigma, p_{id}, w),$$

3. The requester computes hash of the storage identifier, his identifier and the proof to obtain $\mathsf{requestHash}^* \leftarrow H(id|p_{id}|\pi)$ and publishes $\mathsf{requestHash}^*$ on blockchain. Let id^* be the identifier of the published request.
4. The requester stores $(id^*, id|p_{id}|\pi)$ offchain.
5. Each committee member retrieves $\mathsf{requestHash}^*$ from the blockchain request with the identifier id^*, $id|p_{id}|\pi$ from the offchain storage, and verifies that:

$$\mathsf{requestHash}^* \overset{?}{=} H(id|p_{id}|\pi)$$

   If not, the committee member aborts.
6. Each committee member retrieves requestHash from the blockchain request with the identifier id, $F|\sigma$ from the offchain storage, and verifies that:

$$\mathsf{requestHash} \overset{?}{=} H(F|\sigma)$$

   If not, the committee member aborts.
7. Each committee member C_i retrieves its share of the secret, $\mathsf{dpss\text{-}data}_i$, from its internal storage.
8. Each committee member C_i checks if π is a valid proof using the NIZK's verification algorithm V:

$$V(\sigma, p_{id}, \pi) \overset{?}{=} true$$

   If so, C_i and party p_{id} engage in the DPSS **Reconstruction** using $\mathsf{dpss\text{-}data}_i$.

6 Application Examples

In this section, we present some motivational application examples and briefly explain the key ideas behind implementing each of them using our construction. **Time-lock Encryption.** Time-lock encryption, related to timed-release encryption introduced by Rivest et al. [RSW96], allows one to encrypt a message such that it can only be decrypted after a certain deadline has passed. Time-lock encryption must satisfy a number of properties [LJKW18], such as the encrypter needs not be available for decryption and trusted parties are not allowed. Time-lock encryption can be easily implemented using the PublicWitness scheme (see [GKM+20]). Using this scheme, the encrypter executes *SecretStore* with a secret release condition F specifying the time t when the data can be released. Once the time has passed, a user who wishes to see the message submits a *SecretRelease* request with the witness "The deadline has passed". Miners check whether the time is indeed past t and if so, release their fragments of the secret. With a slight modification to our scheme, it is also possible to enable automatic decryption - upon receiving a secret storage request with an "automatic" tag, miners would place the identifier in a list and periodically check whether the release condition holds for any request in this list.

Note that we evade the issue that some time-lock schemes [LJKW18] have: even if the adversary becomes computationally more powerful, it does not allow him to receive the secret message earlier. Additionally, we avoid the computational waste of timed-release encryption schemes [RSW96], which often require the decrypter to, say, compute a long series of repeated modular squarings.

Dead-man's Switch. A dead-man's switch is designed to be activated when the human operator becomes incapacitated. Software versions of the dead-man's switch typically trigger a process such as making public (or deleting) some secret data. The triggering event, for centralized software versions, can be a user failing to log in for three days, a GPS-enabled mobile phone that does not move for a period of time, or a user failing to respond to an automated email. A dead-man's switch can be seen as insurance for journalists and whistleblowers.

A dead-man's switch can use our PublicWitness protocol as follows: the user who wishes to setup the switch generates a SecretStore request with the desired release condition. Such condition can be failing to post a signed message on the blockchain for several days or anything publicly verifiable. As in the time-lock example, we can either use the standard scheme where a person (e.g., a relative or a friend) proves to the miners that the release condition has been satisfied or define an "automatic" request where the miners periodically check the release condition.

Fairness. eWEB can be used to support fair exchange, which ensures that two parties receive each other's inputs atomically. Using eWEB, Alice specifies a release condition that requires a signature from her and from Bob, while Bob's release condition requires only a signature from Bob. Once both secrets are posted, Alice verifies Bob's release condition and posts her signature. When Bob posts his signature, the committee releases both their secrets atomically. Fair exchange can be used to build fair MPC [Yao82, GHY87].

Multi-party computation (MPC) is considered fair if it ensures that either all parties receive the output of the protocol, or none. In the standard model, fair MPC was proven to be impossible to achieve for general functions when a majority of the parties are dishonest [Cle86]. However, we can achieve it by simply adapting the construction of Choudhuri et al. [CGJ+17] to use our eWEB protocol, instead of traditional witness encryption. Conveniently, Choudhuri et al.'s scheme relies on a public bulletin board, which is most readily realized in practice via a blockchain-based ledger. Thus, by replacing witness encryption with our blockchain-based scheme, we do not add any extra assumptions to Choudhuri et al.'s construction.

One-time Programs. A one-time program, introduced by Goldwasser et al. [GKR08], is a program that runs only once and then "self-destructs". In the same work they presented a proof of concept construction that relies on tamper-proof hardware. Considerable work on one-time programs followed [GIS+10, BHR12, AIKW15, DDKZ13], but all such schemes relied on tamper-proof hardware. Goyal and Goyal [GG17], however, present the first construction for one-time programs that does not rely on tamper-proof hardware (but does rely on extractable witness encryption). As with fair MPC and Choudhuri et al.'s construction, by replacing the witness encryption scheme with our eWEB protocol in the Goyal and Goyal's one-time program construction with public inputs, we are not adding any extra assumptions since they already rely on blockchains. Since eWEB reveals whether a secret was retrieved, additional mechanisms are needed when the inputs submitted to the one-time program must be kept private.

Non-repudiation/Proof of Receipt. A protocol allows *repudiation* if one of the entities involved can deny participating in all or part of the communication. With eWEB, it is easy to provide a proof that a person received certain data. In this case, the user providing the data stores it using the SecretStore protocol. To satisfy the release condition F, a user with public identifier pid publishes a signed message "User pid requests the message". The miners then securely release a secret to the user pid as specified by SecretRelease. The publicly verifiable signature on the message "User pid requests the message" then serves as a proof that party pid indeed received the data.

6.1 Voting Protocol

As a more detailed example, we show how eWEB can support a "yes-no" voting application. Specifically, using eWEB, each voter can independently and asynchronously cast their vote by secret sharing a -1 for a "no" or a 1 for a "yes" (note that $(0, 1)$ voting can be supported as well). When voting closes, the miners release an aggregate of the votes. The vote of any specific client must be kept private (guaranteed by eWEB's secrecy), and no client should be able to manipulate the result more than with his own vote.

To prevent improper votes, the committee must verify the correctness of the secrets shared by the clients; i.e., that each $s \in \{-1, 1\}$. Our key idea is to let each client first commit to its secret and then prove its correctness to the miners. However, this requires the client to prove that the committed value is the same as the value the client shared to the committee. To avoid this expensive check, committee members instead *compute* the necessary commitment using the secret shares they receive from the client (guaranteeing consistency by construction).

In [GKM+20], we show that the committee members can prepare Pedersen commitments [Ped92] for all of the clients with constant amortized cost. For a client's secret s, the resulting commitment is of the form $c = g^s h^z$, where z is a random value (known to the client) and g, h are publicly known generators with $h = g^\beta$ for some unknown β.

With such a commitment, the user can prove $s \in \{-1, 1\}$ by proving $s^2 = 1$. To prove that $s^2 = 1$, the client (who knows s and z) computes $w = g^{2sz} h^{z^2}$ and publishes w to all parties. To check that $s^2 = 1$, anyone can check that:

$$e(c, c) = e(g, g) \cdot e(h, w).$$

Correctness. To show correctness, note that

$$\begin{aligned} \text{LHS} &= e(g^{s+\beta z}, g^{s+\beta z}) = e(g, g)^{s^2+2\beta sz+\beta^2 z^2} \\ \text{RHS} &= e(g, g) \cdot e(g^\beta, g^{2sz+\beta z^2}) = e(g, g)^{1+2\beta sz+\beta^2 z^2}. \end{aligned}$$

If the equation holds, then $s^2 = 1$ and thus the client's vote is valid.

To compute the voting result the committee computes the sharing of the result relying on the linear homomorphism of KZG commitments and Shamir's secret sharing, and then follows the usual *SecretRelease* procedure.

7 Implementation

We implement both FaB-DPSS and our eWEB scheme in about 2000 lines of Python code. To perform the underlying field and curve operations, we add Python wrappers around the C++ code of the Ate-Pairings library [Shi10]. For networking, we rely on gRPC [gRP], and for hashing, we use SHA256. For our NIZK scheme, we currently use Schnorr's proof of knowledge [Sch90]. We make it non-interactive via the Fiat-Shamir heuristic [FS86], thus simultaneously making it simulation extractable [FKMV12].

Polynomial arithmetic is done over the polynomial ring $\mathbb{F}_p[X]$ for a 254-bit prime p. For the KZG commitment scheme [KZG10], we use an ate pairing over Barreto-Naehrig curves of the form $y^2 = x^3 + b$ for constant b over $\mathbb{F}_p$ with a 254-bit prime p. We implement polynomial interpolation for polynomials of degree n in time $O(n \log^2 n)$ using an algorithm presented by Aho et al. [AHU74].

8 Experimental Evaluation

We evaluate FaB-DPSS and eWEB and show that:

1. FaB-DPSS outperforms the state-of-the-art (§8.1).
2. eWEB's performance is dominated by FaB-DPSS.
3. Our eWEB prototype's performance matches the expected asymptotics with small constants (§8.2), making it practical to integrate with existing blockchains.

We discuss microbenchmarks in [GKM+20].

Setup. We run experiments using CloudLab [DRM+19], an NSF-sponsored testbed that provides compute nodes along with a configurable networking substrate. We run experiments in both a LAN setting (∼0.2 ms ping) to focus on the

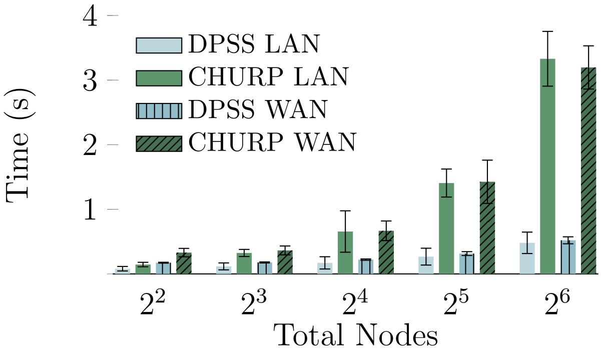

Fig. 2: Handoff Times for our DPSS vs. CHURP. Error bars represent 95% confidence interval.

CPU overhead of our cryptography and a WAN setting (∼40 ms ping) to demonstrate the networking overhead. In the LAN setting we use up to 128 machines each with 8-core 2.00GHz CPUs and 4 GB RAM. In the WAN setting we use up to 128 machines split between Salt Lake City, Utah and Madison, Wisconsin. These machines have 8–10 cores and 2.00–2.4GHz CPUs with 2–4GB RAM.

Since eWEB is compatible with a wide range of blockchains, we abstract away the blockchain and simulate it via a single trusted node. In practice, writes to the blockchain will incur additional blockchain-specific latency.

8.1 DPSS Comparison

As §1.2 discusses, the most efficient prior DPSS schemes are CHURP [MZW$^+$19] and that of Baron et al. [BDLO15]. Since CHURP reports [MZW$^+$19, §6.3] that their performance dominates that of Baron et al., we focus on CHURP.

In our experiment, we measure the time required for each scheme to handoff secrets to a new committee in the optimistic case where parties behave honestly. Both schemes have a fallback path for when malfeasance is detected; it adds an $O(n)$ factor to both schemes.

Figure 2 summarizes the average time for 50 runs. As expected from our asymptotic analysis, FaB-DPSS increasingly out-performs CHURP as the number of nodes increases, to the point where our scheme is ∼7× faster than CHURP with 64 nodes. The absolute difference will increase as committee sizes grow.

Note that the additional networking overhead in the WAN setting (∼40 ms latency) only significantly affects the end-to-end latency for committees with less than 8 members for both FaB-DPSS and CHURP. For larger committees, computation dominates networking costs even with realistic latencies.

8.2 eWEB Performance

We measure the costs of eWEB's top-level operations (*SecretStore*, *SecretsHandoff*, and *SecretRelease*) for the minimal Schnorr identification application over an increasing number of committee members. In particular, given a public key, com-

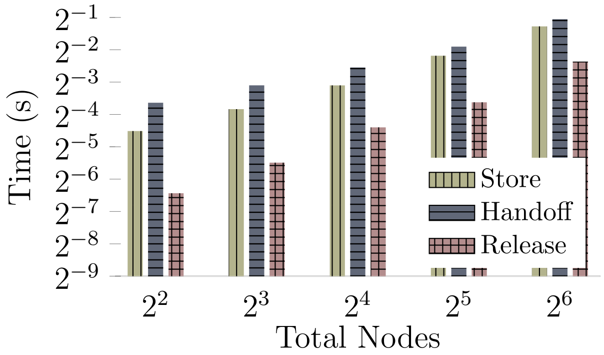

Fig. 3: Time required for high-level eWEB steps on a LAN. Non-DPSS operations are too small to see.

mittee members release the secret if a client proves (in zero-knowledge) that they possess the associated secret key.

Figure 3 summarizes the average time for 150 runs (note the log-log scale). Each bar shows the split between eWEB operations (e.g., preparing the NIZK proof) and the underlying DPSS operations. Note that the time for *SecretsHandoff* includes the amortized cost for the preparation phase that produces coupled sharings of random value used during the refresh phase. Similiarly the time for *SecretStore* includes the amortized cost for the preparation phase that produces sharings for random values used to distribute the initial secret.

The DPSS costs dominate, to the point where the time for eWEB operations cannot be seen. The performance results match our expectation of linear asymptotic growth, and concretely costs ∼7.3 milliseconds/node, ∼10.7 milliseconds/node, and ∼3.0 milliseconds/node for the store, refresh, and release secret operations respectively. This suggests if CloudLab allowed us to scale beyond 64 nodes per committee, we would expect eWEB to store, refresh, release secrets in 7.3 s, 10.7 s, and 3.0 s respectively, even with a 1000-node committee.

8.3 Applications

We implement several applications on top of our eWEB protocol in order to demonstrate practicality and efficiency for common use cases. As a baseline we implement the minimal Schorr identification application: Given a public key, committee members release a secret when provided a (zero-knowledge) proof that a client possess the associated private key. Because the Schnorr identification protocol only requires a few additional group operations for both the client and committee members, this gives us the best view of eWEB's core operational cost.

We additionally implement *time-lock encryption* and *dead-man's switch* as described in §6. In both applications, a claim that the prescribed amount of time has passed is treated as the "witness". In the latter application, we additionally implement an update functionality that allows an operator to extend the secret-release timeout if they provide a valid signature.

We implement the *fair exchange* (§6), where given valid signatures from two clients, the committee releases both their secrets atomically.

	Committee Size				
	4	8	16	32	64
Schnorr Identification	0.15 s	0.22 s	0.37 s	0.67 s	1.22 s
Time Lock Encryption	0.15 s	0.22 s	0.36 s	0.66 s	1.25 s
Dead-man's Switch	0.15 s	0.23 s	0.37 s	0.68 s	1.23 s
Fair Exchange	0.19 s	0.27 s	0.44 s	0.78 s	1.45 s

Table 1: Cost of eWEB applications. End-to-end latency including secret store, a single handoff, and secret release. (50 trials)

9 Conclusion

We have introduced a new and highly efficient batched DPSS protocol – FaB-DPSS. We also proposed eWEB – a new cryptographic primitive which allows the blockchain to store and release secrets. We designed a proof of concept eWEB protocol based on FaB-DPSS and implemented it. Additionally, we implemented and evaluated several applications atop eWEB.

Acknowledgements

We thank Emanuel Jöbstl for helping us with the experimental evaluation of this work.

References

ABB$^+$18. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, 2018.

ABG$^+$13. Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. *IACR ePrint, 2013/689*, 2013.

AHU74. A. Aho, J. Hopcroft, and J. Ulman. *The Design and Analysis of Computer Algorithms.* 1974.

AIKW15. Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate, or how to compress garbled circuit keys. *SIAM Journal on Computing*, 44(2), 2015.

ARS20. Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. Lift-and-shift: Obtaining simulation extractable subversion and updatable SNARKs generically. *IACR ePrint, 2020:62*, 2020.

BDLO15. Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *Applied Cryptography and Network Security*, 2015.

BEDLO14. Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014.

BFM88. Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 1988.

BGG$^+$20. 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 *Theory of Cryptography Conference*, 2020.

BGI$^+$12. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM)*, 59(2), 2012.

BH15. Mihir Bellare and Viet Tung Hoang. Adaptive witness encryption and asymmetric password-based cryptography. In *IACR Workshop on Public Key Cryptography*, 2015.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *Conference on the Theory and Application of Cryptology and Information Security*, 2012.

CGJ$^+$17. Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, 2017.

CKLS02. Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

Cle86. Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, 1986.

DDKZ13. Konrad Durnoga, Stefan Dziembowski, Tomasz Kazana, and Michal Zajac. One-time programs with limited memory. In *Conference on Information Security and Cryptology*, 2013.

DJ97. Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications. Technical report, Citeseer, 1997.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *IACR CRYPTO*, 2007.

DRM$^+$19. Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019.

EOPY18. Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive secure multiparty computation with a dishonest majority. In *International Conference on Security and Cryptography for Networks*, 2018.

ES14. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Conference on Financial Cryptography and Data Security*, 2014.

FKMV12. Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the fiat-shamir transform. In *International Conference on Cryptology in India*, 2012.

FS86. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, 1986.

FY92. Matthew Franklin and Moti Yung. Communication complexity of secure computation. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, 1992.

GG17. Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In *Theory of Cryptography Conference*, 2017.

GGHW14. Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In *IACR CRYPTO*, 2014.

GGSW13. Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013.

GHM$^+$17. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

GHM$^+$20. Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR and applications. Cryptology ePrint Archive, Report 2020/1248, 2020.

GHY87. Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure fault-tolerant protocols and the public-key model. In *Conference on the Theory and Application of Cryptographic Techniques*, 1987.

GIS$^+$10. Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *Theory of Cryptography Conference*, 2010.

GKM$^+$20. Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. https://eprint.iacr.org/2020/504.

GKP$^+$13. Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run Turing machines on encrypted data. In *IACR CRYPTO*, 2013.

GKR08. Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. One-time programs. In *IACR CRYPTO*, 2008.

GMPS21. Vipul Goyal, Elisaweta Masserova, Bryan Parno, and Yifan Song. Blockchains enable non-interactive MPC. In *Theory of Cryptography Conference*, 2021.

gRP. gRPC: A high performance, open-source universal RPC framework. https://grpc.io/.

GSS21. Vipul Goyal, Yifan Song, and Akshayaram Srinivasan. Traceable secret sharing and applications. In *IACR CRYPTO*, 2021.

HJKY95. Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *IACR CRYPTO*, 1995.

KKAS$^+$18. Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Verifiable management of private data under byzantine failures. IACR ePrint, 2018/209, 2018.

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.

KZM$^+$15. Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, H Chan, Charalampos Papamanthou, Rafael Pass, Shelat Abhi, and Elaine Shi. CØCØ: a framework for building composable zero-knowledge proofs. *IACR ePrint, 2015/1093*, 2015.

LJKW18. Jia Liu, Tibor Jager, Saqib A Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86(11), 2018.

MZW$^+$19. Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: Dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM Conference on Computer and Communications Security*, 2019.

OY91. Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the ACM symposium on Principles of distributed computing*, 1991.

Ped92. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *IACR CRYPTO*, 1992.

RSW96. Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

Sch90. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *IACR CRYPTO*, 1990.

Sha79. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

Shi10. Mitsunari Shigeo. High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves. In *International Conference on Pairing-Based Cryptography*, 2010. https://github.com/herumi/ate-pairing.

SLL08. David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, 2008.

WWW02. Theodore M Wong, Chenxi Wang, and Jeannette M Wing. Verifiable secret redistribution for archive systems. In *IEEE Security in Storage Workshop*, 2002.

Yao82. Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science*, 1982.

ZSVR05. Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. In *ACM Transactions on Information and System Security*, 2005.