Benjamin Kuykendall*, Hugo Krawczyk, and Tal Rabin

Cryptography for #MeToo

Abstract: Reporting sexual assault and harassment is an important and difficult problem. Since late 2017, it has received increased attention as the viral #MeToo movement has brought about accusations against high-profile individuals and a wider discussion around the prevalence of sexual violence. Addressing occurrences of sexual assault requires a system to record and process accusations. It is natural to ask what security guarantees are necessary and achievable in such a system. In particular, we focus on detecting repeat offenders: only when a set number of accusations are lodged against the same party will the accusations be revealed to a legal counselor. Previous solutions to this privacy-preserving reporting problem, such as the Callisto Protocol of Rajan et al., have focused on the confidentiality of accusers. This paper proposes a stronger security model that ensures the confidentiality of the accuser and the accused as well as the traceability of false accusations. We propose the WhoToo protocol to achieve this notion of security using suitable cryptographic techniques. The protocol design emphasizes practicality, preferring fast operations that are implemented in existing software libraries. We estimate that an implementation would be suitably performant for real-world deployment.

Keywords: threshold cryptography

DOI 10.2478/popets-2019-0054
Received 2018-11-30; revised 2019-03-15; accepted 2019-03-16.

1 Introduction

All perpetrators of sexual assault should face consequences. But recent events indicate that cases brought against offenders are the most successful if a repeated pattern of abuse can be exhibited. Furthermore, survivors are empowered by their peers: the knowledge that others will testify against the same perpetrator gives them confidence to speak up. However, if all survivors remain silent until they know they are not alone, these accusations will seldom come to light. Similarly to the Callisto Project [30], we propose a system to securely match accusations against the same person. Any such system must balance privacy and utility. Our protocol keeps accusations completely private, at first. But when enough people accuse the same person, the accusations are revealed to the appropriate parties. Matching accusations while keeping them hidden seems contradictory, but using cryptographic tools, we can design protocols that provide both privacy and functionality.

In this paper we present a protocol that enable survivors of abuse to submit an accusation to a special database. The details of the accusations, including the names of the accuser and the accused, will be hidden until the system identifies that there is a *quorum* of complaints against the same person. Here, quorum refers to a number set by the system, e.g., 3. It is illustrative to consider the physical analogy where the accuser sends a sealed envelope with the name of the accused on the outside and his or her own name on the inside. A trusted party will read the name on the outside of the envelopes and sort them into piles. Thus each pile will contain the accusations against a single person. Once there are three envelopes in a pile, the trusted party opens them, reads the accuser names, and acts upon their accusations.

The protocol above places an enormous amount of trust in the central party. Not only will it learn the names of the accused, but it must be trusted not to open the envelopes prematurely, learning the names of the accusers. Cryptography allows us to emulate the trusted central party described above while reducing the trust that one has to place on those running the system. We achieve this by utilizing various cryptographic techniques, including group signatures, multiplicatively homomorphic encryption, privacy-preserving set operations, and threshold cryptography. The protocol design emphasizes practicality, preferring fast operations that are implemented in existing software libraries. We estimate that an implementation of the protocol would be suitably performant for real-world deployment. Concretely, accusers should be be able to submit accusations from a cell phone or personal computer, and the rest of the rest of the system should run on a small number

***Corresponding Author: Benjamin Kuykendall:** Princeton University, E-mail: brk@princeton.edu
Hugo Krawczyk: Algorand Foundation, E-mail: hugokraw@gmail.com
Tal Rabin: Algorand Foundation, E-mail: tal@algorand.foundation
Work done while the authors were at IBM Research.

of reasonably-sized servers. Our cryptographic design strives to guarantee the privacy of each accuser and assure them that their voice will not be heard alone.

In addition to combining existing cryptographic tools, we present new procedures that efficiently solve the problem at hand. In particular, we present a randomized algorithm to check if an element of a multiset exceeds some multiplicity (Section 6.1) and an efficient amortized protocol to perform zero test in private polynomials (Section 6.2). We hope these generic components can be re-used to solve other problems in threshold cryptography.

1.1 Technical Overview

Our goal is to emulate the envelope, piles and trusted party described above. We use encryption and group signatures to emulate an envelope, a privacy-preserving multiset and equality test to store the piles and detect multiple accusations, and threshold cryptography and private computations to create the trusted party. To formalize our security requirements, we present an ideal functionality in Appendix B.1. In summary, the following pseudocode describes an ideal accusation.

Ideal.Accuse(Name)

```
1. Receive input Name from User
2. If User is not valid: halt
3. Accusers[Name] ← Accusers[Name] $\cup$ {User}
4. If # Accusers[Name] < Quorum: halt
5. Contact each User$_i$ $\in$ Accusers[Name]
```

The group signature primitive allows us to emulate Step 2 in the ideal accusation. The user will sign her accusation, and to check if she is a valid user, the authority will verify her signature. Group signatures guarantee privacy during verification: although the authority will learn that some valid user signed the message, they will not learn which. To emulate Step 5 of the functionality, the authority will recover the user identity from the signature using the trace functionality. This separation of verification and tracing is the key feature of group signatures. We shall use the BBS group signature scheme in our instantiation [10]. The BBS group signature scheme relies on a single trusted authority to distribute keys and perform traces. In order to use group signatures with weaker trust assumptions, we build a *threshold BBS* protocol, distributing the BBS authority over n servers in a way that tolerates t corruptions.

To submit an accusation, an accuser encrypts the name of the accused, signs it with her secret group signature key, and sends the accusation to the distributed authority. In order to emulate Step 3 of the ideal accusation, the distributed authority must be able to build a privacy-preserving multiset of accused names using only encryptions of the names. We build this data structure by combining structural properties of polynomials with the homomorphic property of the encryption scheme. In addition to building the data structure, to emulate Step 4 in the ideal functionality, the system must detect when a quorum of accusations is reached. We present a new quorum-finding algorithm based on polynomial-based multiset operations [25].

Finally, once the quorum is reached and detected, Step 5 requires identifying the set of accusers and revealing the identity of the accused. The distributed authority will find this set via a computation utilizing equality plaintext testing on ciphertexts. Once the set of accusers is found, the authority will trace the group signatures and decrypt the name of the accused. Our design will permit arbitrary operations to occur after the quorum is met: e.g. the information could be distributed between the accusers or revealed to an external counselor.

The protocol can be easily extended to support arbitrary attachments. These attachments are encrypted along with the name of the accused and decrypted upon reaching a quorum. They are intended to provide auxiliary information about the accusation, and this method of storage provides confidentiality as will as time stamping. This extension could be important in practice, but for simplicity of presentation, we omit it from our description of the protocol.

1.2 Prior Work

Cryptographic tools. We rely on techniques from threshold cryptography [19, 20] including protocols for sharing in the exponent and the one-time multiplication of secret shares. In a similar vein to our work in Section 5, the authors of [7] use threshold cryptography to build a distributed version of the BBS.Trace functionality. However, their work alone is not sufficient for our uses. In particular, it assumes a trusted setup.

We base our new quorum detection protocol on the polynomial set intersection protocols of [18, 25] augmented to support distributed zero tests. Kissner and Song [25] present element reduction in multisets; that is, given one multiset, they build a new multiset where each item's multiplicity is reduced by Quorum. Their approach hinges on the observation that if $(x-s)^q \mid F$ then the $(q-1)$-th derivative $F^{(q-1)}$ must have a root at s. They use this observation, combined with a careful re-randomization of the polynomial, to perform private element reduction. Our quorum operation protocol follows the spirit of their protocol, but is more efficient.

Media Men list. The first efforts to catalogue accusations of sexual violence did so without cryptographic guarantees. In one highly-publicized example, accusations were published on a collaborative spreadsheet [15]. On this spreadsheet, names of the accused were public and accusers were anonymous. However, since the project was hosted on the Google Drive platform, it may have revealed identifying information to Google. This project made it easy to bring an accusation into the public eye; however, it failed to bring accusers together and failed to adequately protect privacy.

Project Callisto. In a cryptographic setting, Project Callisto, an organization that develops technology to support survivors of sexual assault, implemented the functionality we are interested in, albeit with weaker security guarantees [30]. Their protocol uses an oblivious pseudorandom function (OPRF) in order to hide the names of the accused. In particular, their database stores $\pi \leftarrow \mathsf{H}(f(\text{Name}))$ where H is a hash function, f is an OPRF with a fixed key, and Name is the name of the accused. Oracle access to $f(\cdot)$ is provided to all registered users via two key servers that use threshold primitives. Despite the use of cryptographic primitives to hide some data, their solution does not entirely protect the identities of the accuser or accused; we present three attacks that can be executed by relatively weak adversaries.

First, consider a single malicious accuser. Despite a registration and authentication system, the Callisto solution does not bind the accuser's identity to their accusation. The accuser identity R is simply encrypted and submitted to the system. To make superfluous accusations, a malicious accuser can replace R with a random string. By submitting such accusations, the adversary can force the number of accusations against some accused party Name past the quorum. This prematurely exposes the accusers and accused to the counselor. To strengthen this attack, instead of using a random string as R, the attacker can frame another real user R' by submitting the accusation under the identity of R'.

Second, consider the collaboration between a malicious user and an honest-but-curious database server. They will learn how many accusations against Name_0 are in the system. First, the malicious user can authenticate with the key server to acquire $\pi_0 \leftarrow \mathsf{H}(f(\text{Name}_0))$. Then as H, f are deterministic, the database server can count the number of occurrences of π_0 in the database to learn the number of accusations against Name_0.

Third, consider an honest-but-curious key server. It learns when each accuser makes an accusation. This is a simple consequence of the accuser authenticating to the key server each time it needs to compute the OPRF.

Our protocol prevents each attack: we use the traceability of group signatures to bind accuser identity to the accusation, homomorphic encryption to protect the names of the accused, and the anonymity of group signatures to protect accuser identities, in particular against framing attacks. We believe preventing these attacks warrants the conceptual complexity and computational cost we introduce.

Write-in Elections. The primary functionality of our protocol is to match anonymous authenticated records containing the same encrypted name. Thus, it can be viewed as a secure election protocol for write-in candidates. Although election protocols are well-studied in cryptography, write-ins are less so, and in most election protocols either write-ins are not allowed (e.g. [14]) or write-in candidates are made public (e.g. [24]). This is sufficient in balloted elections when write-ins are an exception to the norm. However, write-ins are always used in nominations: each voter can nominate someone, and persons who achieve at least some quorum of nominations are added to the ballot. A protocol such as ours, combined with a verified mix-net to avoid revealing anything corresponding to the order of the votes, would be appropriate for conducting a secure write-in election.

Distributed encryption. Distributed encryption is a primitive that allows the recipient, or *combiner* in the literature, to decrypt a message if and only if it receives ciphertexts from sufficiently many *encryptors* [22, 26]. It is primarily studied in the context of revocable privacy, but it is connected to our problem as well. Each accuser acts as an encryptor, and an accusation serves the role of an encryption share of the name of the accused. The distributed authority acts as the decryptor. Thus, our protocol implies a q-out-of-N distributed encryption with a thresholdized combiner. This guarantee is weaker than the standard notion of distributed encryption. Usually, the combiner is a single malicious party, but our authority is distributed and partially trusted. Viewed in this light, our protocol trades trust for efficiency, achieving fast amortized recombination at the cost of assuming at most t-out-of-n recombination servers are corrupted.

2 Overview

This section provides the motivation for and informal description of each step of the WhoToo protocol. Recall the basic structure of the system: accusers submit accusations containing the name of the accused party. When the number of distinct accusers providing a given name reaches the quorum, the names of the accusers and the accused are disclosed to a counselor that decides on the appropriate subsequent actions to take. The security of WhoToo ensures full anonymity of both the accuser and the accused until this point.

Identifiers. The letters R and D are used, respectively, for the identities of the accuse$\underline{r}$ and accuse$\underline{d}$. For accusers, the identifier R is a public key distributed during registration. For the accused, the identifier D is simply a name or other real-world identifying string. The accuser, as a participant in multi-party protocols, is denoted by U. There is no such distinction needed for the accused, since it does not participate in any protocols.

Registration. Each user registers with a trusted registration authority. This authority checks the user's identity against some real-world credential (e.g. a government-issued credential, a school or employee ID) to confirm the user is eligible to participate in the system. If these checks pass, the user is appended to the list of valid users.

Each valid user is issued a public key R that is tied to her identity as well as a private key α. To ensure that keys are issued to the authorized parties and that secret keys remain unknown to anyone but the authorized recipient, the issuing authority is distributed among a set of n servers refereed to as the *Distributed Authority* $\mathcal{DA}$. The $\mathcal{DA}$ is built using threshold cryptography, ensuring that as long as at most t out of the n servers are corrupted, all operations will be performed correctly and no one will learn private values such as user signing keys or accused/accuser identities. The $\mathcal{DA}$ is used throughout the protocol to provide these guarantees.

Registration is needed to prevent malicious accusations from anonymous parties. Such accusations present a real security threat: without registration, a malicious user Bob can determine if anyone has lodged an accusation against him. Assume the quorum is set to three. Bob will accuse himself anonymously in two distinct accusations. If there was a previous accusation against Bob, then after his attack, there will be three, reaching the quorum. Bob will learn about this fact when he (or his fake identity) is contacted as an accuser whose accusation reached a quorum.

A variant of this attack is still possible in a system with registration: Instead of lodging anonymous accusations, Bob can ask two registered friends to accuse him. While registration does not fully prevent this attack, the association of users to real-world identities should act as a major deterrent against false accusations. Indeed, the identities of Bob's friends will be revealed during the legal process initiated when an accusation reaches a quorum, making these friends accountable, and possibly punishable, for their fraudulent actions.

For these deterrents to be significant, it must be hard to corrupt registered users or to register fake users. This puts trust on those issuing credentials and registering users. These functions can also be distributed in order to lower the trust on any single entity but such decentralization is beyond the scope of this work. Instead, we model a trusted registration authority by a simple list of valid users. While our simple abstraction assumes this list is fixed, the solution easily supports the dynamic case.

Lodging accusations. Say an accuser U with identifier R wishes to accuse D. The accusation is an encoding of R and D. The encoding must be hiding to guarantee anonymity. But it also must have a special structure so the $\mathcal{DA}$ can manipulate it appropriately. On top of the encoded values, R provides signatures and proofs of correctness, ensuring accountability and allowing the system to discard malformed accusations.

Encryption achieves the necessary hiding property. In particular, we use *threshold* public key encryption where a single public key can be used for encryption, but decryption requires the cooperation of $t + 1$ of the $\mathcal{DA}$ servers. We also require a *multiplicatively homomorphic* encryption scheme where the product of two encryptions is an encryption of the product of the underlying plaintexts. This special structure allows some efficient protocols on encrypted data.

The accuser must also prepare a *sharing* of D into n shares, one share sent to each $\mathcal{DA}$ server. Secret sharing provides privacy by sending only a part of the secret D to each server. We choose a secret sharing scheme that preserves the additive structure of shared values. This allows some efficient protocols on shared values.

After preparing encryptions of R and D and shares of D, the accuser signs the accusation with her private key α. Specifically, we use group signatures which provide two operations: a *verify* procedure that ensures that some valid private key was used for signing, and a *trace* procedure that recovers the public key R of the signer. In our *thresholdized group signature* scheme, a set of at least $t+1$ servers need to agree to trace the accusation to R. They do so only upon reaching the quorum.

Finally, the accuser must provide cryptographic proof, in the form of a string appended to the accusation, that she knows the values that she encrypted. This prevents others from copying her accusation and submitting it as their own. The proofs are zero-knowledge, that is, they reveal nothing about the underlying values.

Aggregating accusations. First, the $\mathcal{DA}$ will discard any malformed accusations, checking the proofs and signatures described above are valid and the secret shares they received are consistent with each other. Next, they check that the accusation is not a duplicate, i.e. no previous accusation was prepared with the same R, D. To perform this check, the $\mathcal{DA}$ servers make use of an *equality test* that checks if two ciphertexts encrypt the same plaintext. This protocol can be implemented efficiently and securely due to the homomorphic structure of the ciphertexts.

After removing malformed and duplicate accusations, the $\mathcal{DA}$ servers proceed to detect quorums. They use a special *private multiset* data structure S that stores in a privacy-preserving way the identities of all persons accused so far, each with a multiplicity equal to the number of accusations against them. This is done in two steps. First, the newly accused identity D is added to the multiset S. Second, the $\mathcal{DA}$ checks whether the multiplicity of D in S equals the quorum. Both of these operations can be implemented efficiently by writing the multiset as an *encrypted* polynomial F_S: each root of the polynomial corresponds to an element of the set, and the multiplicity of the root is the multiplicity of the item. Then, adding an element D corresponds to polynomial multiplication, namely computing $(x-D)F_S$. Checking if the multiplicity meets a quorum q reduces to evaluating the $(q-1)^{\text{th}}$ derivative of F_S at D. These operations are implemented in Section 6 using a secret sharing of D and an encryption of the polynomial coefficients.

If the test reveals that the quorum is met, it remains to collect the list of accusers. This is done by once again utilizing the equality test primitive. The $\mathcal{DA}$ performs a scan of all accusations, and for each accusation against the same party, it performs the group signature trace operation to determine the identity of the accuser.

Finally, once the accusers and accused are identified, control is transferred to a counselor for further processing. This treatment changes according to the scenario in which WhoToo would operate and is non-cryptographic in nature, hence beyond the technical scope of this work. See Section 4.2.

Security. Assume no more than t servers are corrupted. Our solution achieves a very high standard of security, reducing the observable information to the total number of accusations processed by the system. The formal security is defined via an ideal model in Appendix B. Here we state some resulting security properties.

For accusations that have not reached a quorum, there is no leakage on identities, individual accusation counts, or any other private information. A variety of attacks such as copying part or all of an accusation, submitting an accusation anonymously, forging the identity of the author of an accusation are all prevented. In comparison, none of three attacks against [30] state in Section 1.2 are possible against WhoToo. Finally, while submission of fraudulent accusations cannot be prevented by cryptography, our ability to trace false accusations serves as a deterrent to the remaining attack.

3 Building Blocks

This section presents the notation and specification of components used in the protocol. Constructions and proofs are relegated to later sections.

General notation. Let $[N] = \{1, \ldots, N\}$.

Computational assumptions. Our protocol relies on a variety of standard Diffie-Hellman type assumptions. They are stated explicitly in Appendix A.1 and referred to by name when needed.

Random oracle. Let H denote a hash function into $\mathbb{Z}_p$. In some cases we utilize a hash function with a different codomain Y; in this case we write H_Y. We utilize the hash function with different domains in different contexts; in each case, the number and type of arguments should make the domain clear. Treat all hash functions as random oracles for the purposes of security proofs.

Bilinear groups. Let $\mathbb{G}_1, \mathbb{G}_2$ be two cyclic groups of prime order p with generators g_1, g_2 respectively. Let $\phi : \mathbb{G}_2 \to \mathbb{G}_1$ be a computable isomorphism such that $\phi(g_2) = g_1$. Finally let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear map such that $e(g_1, g_2) \neq 1$. We suggest a specific family of groups to utilize in Appendix A.1.

3.1 ElGamal Encryption

Encrypting group elements.

ElGamal.Setup. Choose $h \in_{\mathrm{R}} \mathbb{G}_1 \setminus \{1\}$ and $x \in_{\mathrm{R}} \mathbb{Z}_p$. Let $g = h^{1/x}$. Output keys $\mathrm{pkeg} \leftarrow (g, h)$ and $\mathrm{skeg} \leftarrow x$.

ElGamal.Enc(pkeg, m). Choose $a \in_{\mathrm{R}} \mathbb{Z}_p$ and output $c \leftarrow (c_1 = g^a, c_2 = h^a m)$.

ElGamal.Dec(skeg, c). On input $c = (c_1, c_2)$ output c_2/c_1^x.

Assuming the Decisional Diffie-Hellman problem is hard in $\mathbb{G}_1$, this scheme is semantically secure against chosen-plaintext attacks [16]. Further, the scheme is multiplicatively homomorphic in the sense that decryption is a group homomorphism when ciphertexts are considered as elements of $\mathbb{G}_1 \times \mathbb{G}_1$.

Extension to Strings. We can use ElGamal as a key-encapsulation mechanism to encrypt longer strings efficiently. Let (AuthEnc, AuthDec) be a symmetric key authenticated encryption scheme with keyspace $\mathcal{K}$. Define a new encryption scheme with the same ElGamal keys.

ElGamal.EncString(pkeg, m). Choose $a \in_{\mathrm{R}} \mathbb{Z}_p$ and output $C \leftarrow (g^a, \mathsf{AuthEnc}(\mathsf{H}_{\mathcal{K}}(h^a), m))$.

ElGamal.DecString(skeg, c). On input $c = (c_1, c_2)$ output $\mathsf{AuthDec}(\mathsf{H}_{\mathcal{K}}(c_1^x), c_2)$.

Assume AuthEnc is CCA-secure. Under the Strong Diffie-Hellman Assumption, this scheme is CCA-secure when $\mathsf{H}_{\mathcal{K}}$ is a random oracle [2].

Proof of Plaintext Knowledge. We will also need a proof of plaintext knowledge to prevent attackers from submitting encryptions of values that they do not know. We give this proof explicitly in Appendix A.3. Assume c is an ElGamal encryption of m with randomness a. Then let ElGamal.Prove(c, a, ρ) be the non-interactive zero-knowledge proof of knowledge proving knowledge of m such that $c =$ ElGamal.Enc(m) where the random oracle challenge is derived from a given value ρ. Let ElGamal.Verify be the verification procedure.

Additive encoding. We will encode a value $s \in \mathbb{Z}_p$ via an ElGamal encryption of the plaintext g^s for fixed public g. Let $e_s \leftarrow$ ElGamal.Enc(g^s). Considered as a function of s this transforms the multiplicatively homomorphic ElGamal into an additively homomorphic encoding: indeed $e_{s_1+s_2} = e_{s_1} \cdot e_{s_2}$. Note that this encoding is not efficiently invertible even with the knowledge of the ElGamal decryption key. However, it is amenable to zero tests: decrypting the encoding e_s yields g^s, and $g^s = \mathrm{id}$ if and only if $s = 0$. The zero testing will be sufficient for our needs in the following equality test and the privacy-preserving polynomials.

Equality testing. Given the additively homomorphic properties of the encoding there is a fast protocol to test equality of plaintexts using only their encodings [23]. In particular Equal(sk, e_1, e_2) will return True if e_1 and e_2 are encodings of the same message and with high probability will return False when they are not the same. Further, in the threshold setting this protocol will reveal nothing else about the underlying messages. For the specification of this protocol see Appendix A.2.

3.2 Threshold Operations

We decentralize several of the operations used by the group signature authority using standard techniques from the area of threshold cryptography. We rely on multiple operations on shared secrets that we define next and for which we assume a broadcast channel between the participants (the servers of the distributed authority in our application). Below, we assume a group $\mathbb{G}$ of prime order p with generators g, h. Except if said otherwise, secrets and shares will be taken from $\mathbb{Z}_p$ and verification information from G.

Supported operations. In order to share $x \in \mathbb{Z}_p$, we follow the basic outline of Shamir secret sharing. Take the $x_i = P(i)$ for some polynomial P of degree t such that $P(0) = x$. Our discrete log based schemes and applications generally allow to publish the values $\{g^{x_i}\}$ and g^x for verification purposes. This allows for operations analogous to standard Lagrange interpolation on shares but computed in the exponent; e.g., we can compute $g^x = \prod (g^{x_i})^{\lambda_i}$ where λ_i are the appropriate Lagrange coefficients.

Let t be the threshold parameter. In the following we denote the shares of a (t, n) sharing of a secret x by $\{x_i\}_{t,n}$. When we write $\{x_i\}_{t,n}$ as an input or output to a distributed protocol, this means that party C_i has input or output x_i.

SecShare.Reconstruct(x). Reconstruct a secret x from its shares.

SecShare.Gen(g). A primitive which we use repeatedly in the set-up of the system is a distributed generation of a pair x, g^x [20]. The protocol takes as input a generator g of the group $\mathbb{G}$. It results in each party holding the share x_i of $\{x_i\}_{t,n}$ for a secret x uniformly distributed over $\mathbb{Z}_p$ and a public value g^x.

SecShare.GenInv(g). We will also need a distributed generation of a pair $x, g^{1/x}$ [19]. The protocol takes as input the generator g of the group $\mathbb{G}$. It results in each party holding the share x_i of $\{x_i\}_{t,n}$ for a random x and the public value $g^{1/x}$.

$\mathsf{SecShare.Verify}(s)$. Often, when shares of a given secret are shared by a dealer, the receiving parties need to verify that the shares they receive interpolate to a unique secret (called a *verifiable secret sharing (VSS)* protocol [13]). $\mathsf{SecShare.Verify}(s)$ implements such protocol for a secret s.

$\mathsf{SecShare.Add}(x, y)$. On input two shared secrets x and y with shares $\{x_i\}_{t,n}$ and $\{y_i\}_{t,n}$ respectively, the protocol outputs a sharing $\{z_i\}_{t,n}$ of a secret z such that $z = x+y$. For subtraction, define $\mathsf{SecShare.Sub}(x, y)$ analogously.

$\mathsf{SecShare.Mult}(x, y)$. On input two shared secrets x and y with shares $\{x_i\}_{t,n}$ and $\{y_i\}_{t,n}$ respectively, the protocol outputs a sharing $\{z_i\}_{t,n}$ of a secret z, such that $z = x \cdot y$ [6, 21].

$\mathsf{SecShare.Invert}(x)$. On input a shared secret x, output a sharing of x^{-1} [4].

$\mathsf{SecShare.Exp}(b, x)$. On input $b \in \mathbb{G}$ and a shared secret x with shares $\{x_i\}_{t,n}$, generate shares $\{b_i\}_{t,n}$ of $z = b^x$ where $b_i = b^{x_i}$. Given values g^{x_i} a party can prove that b^{x_i} and g^{x_i} have the same exponent x_i using ZK proof of equality of discrete log.

Security. Assume a malicious adversary who can corrupt t parties for $n \geq 2t + 1$. We require correctness and secrecy of the secret sharing scheme. *Correctness* states that recombination does in fact recover the shared value. *Secrecy* states that the information held by corrupted parties reveals nothing about the shared value. Further, we require that each $\mathsf{SecShare.Operation}$ is a secure multi-party computation of the operation described above. Equivalently, this means that the outputs are distributed correctly and the view of the parties during the operation is simulatable from public outputs alone.

3.3 VSS and ElGamal

The previous section presented secret sharing operations in the abstract: we made no stipulations on what values were published for verification purposes. In this section, we choose a concrete verification protocol and show how it connects to the ElGamal-based additive encodings defined in Section 3.1.

To motivate this move, consider both sharing and additively encoding a value s. The shares $\{s\}$ will be generated via a secret sharing scheme. The encoding e_s will be generated by encrypting g^s with a multiplicatively homomorphic encryption scheme. To perform this action in an untrusted environment requires a means to verify the sharing and the encoding are consistent, i.e.

$$\mathsf{Dec}(e_s) = \mathsf{SecShare.Exp}(g, \{s\}).$$

For concreteness and efficiency, we will use the Pedersen verifiable secret sharing scheme [29] and the ElGamal encryption scheme. A structural similarity between the two schemes allows the holders of the shares to easily verify consistency if the sharing and encoding were generated in a coordinated fashion.

Pedersen VSS. A *Pedersen commitment* to s is in the form $g^s h^r$ for random r. The Pedersen verifiable secret sharing scheme uses Pedersen commitments to facilitate a verification procedure.

To generate a share of s, choose a random r. Then generate shares of s and r as in Shamir secret sharing: pick coefficients a_j, b_j for polynomials f_s, f_r of degree t such that $f_s(0) = s$, $f_r(0) = r$. Evaluate the polynomials to obtain $s_i = f_s(i)$ and $r_i = f_r(i)$. Finally, commit to the coefficients by $v_j = g^{a_j} h^{b_j}$. Publish the verification values $v = (v_0, \ldots, v_t)$ and send the i^{th} party the share (s_i, r_i). The pseudocode for this $\mathsf{SecShare.GenPedersen(s)}$ procedure is given in Fig. 1. The verification protocol is not given here, but is presented in full in [29].

Connection to the Additive Encoding. Recall that an additive encoding of s is basically an ElGamal encryption of g^s. For v prepared as above we have

$$v_0 = g^{a_0} h^{b_0} = g^s h^r.$$

Thus, if the public key for the ElGamal encryption is (g, h) then

$$e_s = (g^r, v_0) = (g^r, g^s h^r)$$

is an ElGamal encryption of g^s, i.e. the additive encoding of s. The additional value $e_0 = g^r$ is output by the $\mathsf{SecShare.ShareEncode}(s)$ procedure in Fig. 1. The randomness r is also an output, as it is required for the ElGamal proof of plaintext knowledge.

To check the consistency of the encoding with the sharing, the parties holding the shares will verify

$$e_s = (\mathsf{SecShare.Exp}(g, \{r\}), v_0).$$

This check will be called $\mathsf{SecShare.CheckConsistent}(\omega, v, e_0)$.

3.4 Exponentiation of Ciphertexts

Our privacy-preserving polynomial entails raising an element of the group to a shared secret value. This is in

```
SecShare.GenPedersen($s$)
    1. $r \in_R Z_p$
    2. $a_0 \leftarrow s$, $b_0 \leftarrow r$
    3. $a_j, b_j \in_R Z_p$ for $j \in [t]$
    4. $s_i \leftarrow \sum_{j=0}^{t} a_j i^j$ for $i \in [n]$
    5. $r_i \leftarrow \sum_{j=0}^{t} b_j i^j$ for $i \in [n]$
    6. $v_j \leftarrow g^{a_j} h^{b_j}$ for $j \in [0..t]$
    7. $v \leftarrow (v_0, \ldots, v_t)$
    8. $\omega \leftarrow ((s_1, r_1), \ldots, (s_n, r_n))$
    9. Output $\omega, v$

SecShare.ShareEncode($s$)
    1. Take $r, v, \omega$ as in SecShare.GenPedersen($s$)
    2. $e_0 \leftarrow g^r$
    3. Output $\omega, v, e_0, r$

SecShare.CheckConsistent($\omega, v, e_0$)
    1. $\{s\}, \{r\} \leftarrow \omega$
    2. Return True iff $e_0 =$ SecShare.Exp($g, \{r\}$)
```

Fig. 1. Pedersen sharing and coordination with encoding

fact exactly the operation carried out by SecShare.Exp. However, we also need this exponentiation operation to be semantically secure, i.e. to preserve the secrecy of the exponent and its shares. Thus, we need to extend the SecShare.Exp to incorporate a re-randomization step. As our privacy-preserving polynomials operate over encodings of values we will use the specifics of this in our description of new function SecShare.ExpRR.

SecShare.ExpRR($e = (e_0, e_1), \{s\}$). To exponentiate the encoding corresponding to (e_0, e_1), each party will pick a random encoding of 0 by choosing $r_i \in_R Z_p$ and defining (g^{r_i}, h^{r_i}). Then they will compute $e_i \leftarrow (g^{r_i} e_0^{s_i}, h^{r_i} e_1^{s_i})$, publish e_i, and prove knowledge of r_i and prove equality of discrete log as in SecShare.Exp. Once the proofs are verified they proceed with the usual exponentiation computing $y = \prod e_i^{\lambda_i}$ for appropriate Lagrange coefficients λ_i and output y.

To show correctness, consider $e_0 = g^a$, $e_1 = h^a g^m$. As $\sum \lambda_i s_i = s$ we have

$$\begin{aligned} y &= \prod e_i^{\lambda_i} \\ &= \prod (g^{r_i}(g^a)^{s_i}, h^{r_i}(h^a g^m)^{s_i})^{\lambda_i} \\ &= (g^{as + \sum r_i \lambda_i}, h^{as + \sum r_i \lambda_i} g^{ms}). \end{aligned}$$

Thus conclude y is a random encoding of g^{ms} as desired. Security follows as in SecShare.Exp. The fact that s is hidden follows from the randomness of the r_i and the discrete log assumption.

3.5 BBS Group Signatures

From a high level, a BBS group signature is an encryption c of the user identity together with a "signature of knowledge" σ of a valid user key. Together, these serve to sign the message. To verify the signature, anyone can check σ using the public key. To trace the identity of the signer, the holder of the secret key decrypts c.

We present the full BBS specification in Appendix A.4. Below we highlight the elements that are directly relevant to our use of BBS and its implementation as a distributed system as presented in Section 5.

The original presentation of BBS uses linear encryption rather than ElGamal which provides security under weaker computational assumptions. However, since the stronger assumptions needed to use ElGamal (described in Section 8.1 of [10]) are now rather standard in many pairing-friendly elliptic curve groups, we are comfortable using the weaker variant of the scheme. This variant comes with the benefit of faster operations, smaller signatures, and a simpler explication. It is possible to substitute the original linear encryption scheme in our application if the weaker assumptions are desired.

BBS.Setup. Compute $(\mathrm{pkeg}, \mathrm{skeg}) \leftarrow$ ElGamal.Setup(). Sample a secret $\gamma \in_R \mathbb{Z}_p$ and publish $w = g_2^{\gamma}$ for the zero-knowledge proof. The public key is $\mathrm{pk} = (\mathrm{pkeg}, g_1, g_2, w)$ and the secret key is $\mathrm{msk} = (\mathrm{skeg}, \gamma)$.

BBS.UserKeyIssue(msk). To issue a key to U, choose $\alpha \in_R \mathbb{Z}_p$ and compute $R \leftarrow g_1^{1/(\alpha+\gamma)}$. The identity associated to U is R and her secret key is $\mathrm{sk}_U = (R, \alpha)$.

BBS.Sign($\mathrm{pk}, \mathrm{sk}_U, m$). Compute $c \leftarrow$ ElGamal.Enc(pkeg, R) and the remaining values σ as defined in Appendix A.4. Output (c, σ).

BBS.Verify($\mathrm{pk}, m, c, \sigma$). The signature verification procedure is defined in Appendix A.4.

BBS.Trace($\mathrm{msk}, \mathrm{pk}, m, c, \sigma$). Verify the signature and output ElGamal.Dec(skeg, c).

In the random oracle model, and under the DDH and sDH$_k$ assumptions in $\mathbb{G}_1$, the BBS scheme is secure per the definition given in [5]. In particular, the scheme is correct, unforgeable, anonymous without knowing the msk, and traceable given the msk.

In Section 5 we present thresholdized versions of the above functions. In particular, we specify the protocols DistBBS.Setup, DistBBS.UserKeyIssue, and DistBBS.Trace as distributed protocols performed by the *distributed authority* $\mathcal{DA}$. The BBS.Sign and BBS.Verify procedures do not change in the distributed setting; they remain the regular single-party BBS operations.

3.6 Privacy-Preserving Multisets

The following operations allow the distributed authority to maintain a multiset of accuser names. These operations are performed in a distributed manner such that no adversary learns anything about the content of the set except what is obtained by the quorum operation. Let $\{s\}$ be a secret sharing of the input s, where each party i receives the i^{th} share as in Section 3.2. The implementation of these operations are given in Section 6. The interface is as follows:

$\mathsf{Set.Init}() \rightarrow S$ representing $\varnothing$.

$\mathsf{Set.Add}(S, s) \rightarrow S'$ such that S' representing $S \cup \{s\}$.

$\mathsf{Set.Quorum}(S, s) \rightarrow \mathsf{True}$ iff the multiplicity of s in S is at least q.

4 WhoToo Protocol

Following the outline in Section 2 and using the notation from Section 3 we specify the WhoToo protocol in Section 4.1 and discuss deployment in Section 4.2.

Two operations used in the protocol are highly context-specific and involve real-world information; thus they are not specified formally. $\mathsf{Investigate}(D, S)$ denotes the operation taken once the quorum has been reached, where D is the name of the accused and S the set of accusers. GetUsers denotes the process by which the list of valid accusers is obtained. We re-visit the issue of specifying Investigate and GetUsers from a less formal standpoint in Section 4.2.

4.1 Protocol Description

We proceed to describe the protocol. We summarize it in words below and algorithmically in Fig. 2.

Initialization. The WhoToo.Initialize protocol describes the steps taken by the $\mathcal{DA}$ to initialize cryptographic primitives and distribute keys. First, the $\mathcal{DA}$ obtains the user list. It initializes the threshold BBS scheme by calling DistBBS.Setup as described in Section 5. For each user U, the $\mathcal{DA}$ performs the distributed BBS key generation protocol DistBBS.UserKeyIssue. Thus each user receives the BBS keys (R, sk_U). Further the $\mathcal{DA}$ will associate each public key to the corresponding user by setting $\mathrm{IdentityMap}[R] = U$. Finally, the $\mathcal{DA}$ initializes the sets relating to the accusations, i.e. an empty privacy-preserving multiset as specified in Section 6 and an empty list of accusations.

Lodging an accusation. The WhoToo.Accuse protocol breaks down the accusation process into a number of smaller steps. First, the user prepares an accusation in the WhoToo.PrepareAcc function. The accusation itself is n different messages $\mathrm{acc}_1, \dots, \mathrm{acc}_n$. The user sends acc_i to the i^{th} server of the $\mathcal{DA}$. In the second step the $\mathcal{DA}$ verifies the accusation as detailed in WhoToo.VerifyAcc, halting if it fails to verify. Otherwise, WhoToo.Accuse continues to process the accusation by storing it, i.e. the shared value s is used to update the multiset representing the accusations. The final step is to check for a quorum, using the distributed multiset protocols of Section 6. If the quorum is reached, the $\mathcal{DA}$ proceeds to the WhoToo.OpenAccusations protocol.

Note that the accusation preparation is non-interactive with respect to the user: she need only prepare her messages and send them to the $\mathcal{DA}$ servers.

Preparing an accusation. The $\mathsf{WhoToo.PrepareAcc}(D)$ function specifies how a user U with keys (R, sk_U) builds an accusation against D. Remember, the accusation has n parts, one for each server.

The user starts by processing D in three different ways. The first is a simple encryption c_D of the string D. This allows the $\mathcal{DA}$ to eventually recover D. We use the ElGamal.EncString encryption scheme for simplicity. The next two preparations are more involved, but are required for the $\mathcal{DA}$ to perform privacy-preserving operations on D.

The user will hash D, yielding a value $s \in Z_p$. This values should be thought of analogously to D; it will stand in for D in algorithms that take place in the exponent of $\mathbb{G}_1$. This step is unnecessary if we take the names of the accused to be elements of Z_p to begin with. However, in the more realistic setting where D is a string of arbitrary length, using a hashed value is required.

Next, the user will share and additively encode s as described in Section 3.3. This generates values ω the actual shares of s and r, v a vector of verification values, e_0 the first element of the encoding, and r itself. The values ω_i, v, and e_0 must be sent to the i^{th} server.

The remainder of the function serves to “package” the message sent to server i. Call this message m_i, consisting of c_D along with the values from the share and encoding of s. The inner layer of packaging is the BBS group signature given in Appendix A.4, yielding a signature $c_R||\sigma$. The outer layer of packaging is the proofs π_0, π_1 of plaintext knowledge for encryptions e_s, c_D respectively. These proofs contain the group signature as auxiliary information, assuring that they will only vali-

```
WhoToo.Initialize
    1. ValidAccusers ← GetUsers()
    2. $(\mathrm{pk}, \mathrm{msk})$ ← DistBBS.Setup()
    3. IdentityMap ← $\varnothing$
    4. For each $U \in$ ValidAccusers:
        (a) $R$ ← DistBBS.UserKeyIssue$_U$(msk)
            over authenticated confidential channel
        (b) IdentityMap[$R$] ← $U$
    5. $S$ ← Set.Init()
    6. Accusations ← $\varnothing$

WhoToo.Accuse
U:  1. $(\mathrm{acc}_i)_{i \in [n]}$ ← WhoToo.PrepareAcc(D)
    2. Send $\mathrm{acc}_i$ to server $i$ of the $\mathcal{DA}$
       over anonymous confidential channel
$\mathcal{DA}_i$:3. Halt if not WhoToo.VerifyAcc($\mathrm{acc}_i$)
    4. $S$ ← Set.Union($S, \omega$)
    5. Accusations ← Accusations $\cup \{(c_R, e_s, c_D)\}$
    6. If Set.Quorum($S, \omega$):
           Run WhoToo.OpenAccusations($e_s, \omega, c_D$)

WhoToo.OpenAccusations($e_s, \omega, c_D$)
    1. Accusers ← $\varnothing$
    2. $s$ ← SecShare.Reconstruct($\omega$)
    3. $D$ ← Null
    4. For each $(c_{R'}, e_{s'}, c_{D'}) \in$ Accusations:
           If Equal(skeg, $e_s, e_{s'}$):
               $R'$ ← ElGamal.DistDec(skeg, $c_{R'}$)
               $U$ ← IdentityMap[$R'$]
               Accusers ← Accusers $\cup \{U\}$
               $D'$ ← ElGamal.DistDecString(skeg, $c'_D$)
               If H($D'$) $= s$: $D \leftarrow D'$
    5. If $D$ = Null: halt
    6. Run Investigate($D$, Accusers)
```

```
WhoToo.PrepareAcc($D$)
    1. $c_D$ ← ElGamal.EncString(pkeg, $D$),
       encrypted with randomness $r_D$
    2. $s$ ← H($D$)
    3. $\omega, v, e_0, r_D$ ← SecShare.ShareEncode($s$)
    4. $e_s \leftarrow (e_0, v_0)$
    5. For each $i \in [n]$:
        (a) $m_i \leftarrow c_D || \omega_i || v || e_0$
        (b) $(c_R, \sigma)$ ← BBS.Sign($m_i, \mathrm{sk}_U$)
        (c) $\pi_0$ ← ElGamal.Prove($e_s, r_D, c_R || \sigma$)
        (d) $\pi_1$ ← ElGamal.Prove($c_D, r_D, c_R || \sigma$))
        (e) $\mathrm{acc}_i \leftarrow (c_R, c_D, \omega_i, v, e_0, \sigma, \pi_0, \pi_1)$
    6. Return $(\mathrm{acc}_i)_{i \in [n]}$

WhoToo.VerifyAcc($c_R, c_D, \omega_i, v, e_0, \sigma, \pi_0, \pi_1$)
    1. $e_s \leftarrow (e_0, v_0)$
    2. $m_i \leftarrow c_D || \omega_i || v || e_0$
    3. If any of the following fail, return False:
           ElGamal.Verify(pkeg, $\pi_0, e_s, c_R || \sigma$)
           ElGamal.VerifyString(pkeg, $\pi_1, c_D, c_R || \sigma$)
           BBS.Verify(pk, $m_i, c_R, \sigma$)
           SecShare.Verify($\omega, v$)
           SecShare.CheckConsistent($\omega, v, e_0$)
    4. For each $(c_{R'}, e_{s'}, c_{D'}) \in$ Accusations:
           If Equal(skeg, $(c_R, e_s), (c_{R'}, e_{s'})$):
               Return False
    5. Return True
```

Fig. 2. The #WhoToo protocol: Initialize, Accuse; and Accuse subroutines PrepareAcc, VerifyAcc, and OpenAccusations.

date when attached to this message. Details of the proof system are given in Appendix A.3.

Finally the user assembles the accusation $(c_R, c_D, \omega_i, v, e_0, \sigma, \pi_0, \pi_1)$ for server i.

Verifying an Accusation. The WhoToo.VerifyAcc procedure serves to remove malformed accusations. First, the $\mathcal{DA}$ will re-assemble the encoding e_s and the message m_i from its inputs. This step is completely syntactic; it is a re-labeling of input values in order to utilize them more concisely in the remainder of the verification.

Next, each server of the $\mathcal{DA}$ verifies the proof of plaintext knowledge by running ElGamal.Verify and ElGamal.VerifyString, verifies the group signature by running BBS.Verify, and verifies the secret sharing by running SecShare.Verify. If all check pass, it proceeds.

Then the $\mathcal{DA}$ check to see if this accusation is a duplicate. To do so, it iterates through each prior accusation. It uses the Equal protocol detailed in Appendix A.2 to check if current and previous accusations were generated using the same values of R and s; this is equivalent to checking if they were made by the same user against the same party. If the new accusation is not a duplicate in this sense, the protocol returns successfully.

Opening Accusations. After a quorum occurs, the WhoToo.OpenAccusations procedure is called. The $\mathcal{DA}$ performs a linear scan to find all of the accusations against D. It again utilizes Equal to check if $s = s'$ without decrypting these values. It also recovers the string D by decrypting the value $c_{D'}$ attached to each matching accusation. However, since $c_{D'}$ may not be an encryption of the correct value, the servers check that $\mathsf{H}(D') = s$ as desired. The servers also decrypt the public keys R' of each matching party and uses them to recover the users themselves. It uses all of this data to run Investigate.

4.2 Operational Considerations

This section describes operational issues in deploying WhoToo. We present each issue and give an example of how it could be addressed in a specific context. In other contexts, especially as the size of the system scales, these issues require different solutions.

$\mathcal{DA}$ **Servers.** Servers consituting the distributed authority should not be co-located and should have separate administrations. This should ensure that compromising one server does not make it easy to compromise others. The servers can communicate with eachother over private authenticated channels; typically instantiated with TLS.

Key distribution channel. Each $\mathcal{DA}$ server needs a private authenticated channel to each user; typically instantiated with TLS.

Submission accusation channel. Users need an anonymous channel to each $\mathcal{DA}$ server; can be instantiated with Tor service [1].

User list. Choose implementation of GetUsers. The list of users must be known to each server of the $\mathcal{DA}$.

Names of accused. Our description assumes that accused parties have unique identifiers (e.g., a name, email address, etc.). In practice, the same person can be known by different identities. In this case, a user should submit a separate accusation for every identity by which the accused may be known (as matching is done on the basis of such identities). A more sophisticated "fuzzy matching" of names could be accomplished by replacing the Equal predicate (Appendix A.2) with a more complicated predicate; instantiating this securely and efficiently is outside the scope of this work.

Choice of action. Once a quorum is identified, the WhoToo protocol begins the Investigate process using the name of accused and accusers. This procedure is a place holder for the actions taken by a counselor, legal team, etc., on the basis of this information; it represents steps that are outside the scope of cryptography and of the WhoToo protocol.

Case study. Deploy WhoToo within a single educational institution. At this scale, it is easy to identify and communicate with any member of the institution. Keys can be delivered to students via sealed envelopes placed in student mailboxes. The anonymous channel can be a number of mailboxes placed in common areas. Student governing boards, school services, and external advocacy groups will maintain servers comprising the $\mathcal{DA}$. The school administration will provide a list of enrolled students. In order to accuse parties inside the school, the official school email addresses will serve as accused identities. Rules on how to process accusations that reached a quorum can be integrated into existing school judicial processes.

5 Distributed BBS

In order to provide privacy for the accusers until a sufficient number have accused the same party, we substitute the secret key generations by a distributed process. Though a fully threshold BBS does not appear in the literature, we can describe it as a straight-forward composition of existing threshold primitives. For the specification of each primitive see Section 3.2. Our threshold decryption algorithm is identical to that of [7]; however, our setup protocol is new.

As promised in Section 3.5, we transform DistBBS.Setup, DistBBS.UserKeyIssue, and DistBBS.Trace into distributed protocols performed by the $\mathcal{DA}$. The algorithms are first described in words, and then specified algorithmically in Fig. 3.

DistBBS.Setup(g_1, g_2): To set up the scheme, the parties first generate an ElGamal keypair. They share the private key x and publish the public key (g, h) where $h = g^x$. Next they generate a BBS keypair. They share the private key γ and publish the public key $w = g_2^\gamma$.

DistBBS.UserKeyIssue$_U(\gamma)$: To issue a key to a user U, the parties share the private key α and derive the public key $R = g_1^{1/(\alpha+\gamma)}$ in the clear. Each server sends its share α to the user U. The user recombines the shares to obtain α.

ElGamal.DistDec: To decrypt an ElGamal encryption (c_1, c_2) where $c_1 = g^r$ and $c_2 = mh^s$, the parties use the shared key x to obtain $d = c_1^x = g^{rx} = h^r$. They divide c_2 by d to obtain the message.

Note the ElGamal.DistDec protocol also serves as the distributed BBS trace functionality. To trace the signature (c_R, σ) the parties simply decrypt c_R.

ElGamal.DistDecString: To decrypt an ElGamal string encryption (c_1, c_2) where $c_1 = g^r$ and $c_2 = \mathsf{AuthEnc}(\mathsf{H}(d), m)$, the parties obtain d as before and perform AuthDec with key $\mathsf{H}(d)$ to obtain the message.

We claim that the security of the threshold operations immediately gives the security of the distributed BBS scheme in the sense discussed in Section 3.5.

Lemma 5.1. *Fix any implementation of SecShare.Gen, SecShare.Mult, SecShare.GenInv, SecShare.Sub, and SecShare.Exp, and fix an adversary the secret sharing primitives are secure against. Then, making the same assumptions required for the security of the BBS group signature, the distributed signature scheme is secure, that is, correct, fully anonymous, and fully traceable.*

Proof. Fix a coalition of $\mathcal{DA}$ servers comprising an adversary against which the secret sharing primitives are secure, along with a set of corrupted users C. First observe that the distribution of outputs from DistBBS.Setup, DistBBS.UserKeyIssue, and DistBBS.Trace are indistinguishable from the outputs of BBS.Setup, BBS.UserKeyIssue, and BBS.Trace respectively, run on the same inputs. This follows from the correctness property of the secret sharing scheme and associated operations. Further we argue that the view of the adversary can be simulated from the public parameters of the BBS system and the values revealed to the corrupted users: namely g_1, g_2, g, h, w, the set of all public keys $\{R_U\}$, and the set of private keys $\{\alpha_U\}_{U \in C}$. This follows from the secrecy property of the secret sharing scheme.

This suffices to translate any attack against the distributed system into an attack on the original BBS system. Replace the distributed protocols by the original protocols and the true adversary by a simulated one. This translated adversary has output indistinguishable from the output of the true adversary. Further, this adversary will be a valid adversary against the original BBS scheme, as it consists only of users that see only public parameters and their own private keys. □

```
DistBBS.Setup()
    1. $g \in_R \mathbb{G}_1$
    2. $(\{x_i\}_{t,n}, h = g^x) \leftarrow \mathsf{SecShare.Gen}(g)$
    3. $\mathrm{pkeg} \leftarrow (g, h)$
    4. $(\{\gamma_i\}_{t,n}, w = g_2^\gamma) \leftarrow \mathsf{SecShare.Gen}(g_2)$
    5. Publish $\mathrm{pk} \leftarrow (\mathrm{pkeg}, w)$

DistBBS.UserKeyIssue$_U(\{\gamma_i\}_{t,n})$
    1. $(\{\gamma'_i\}_{t,n}, R) \leftarrow \mathsf{SecShare.GenInv}(g_1)$
    2. $\{\alpha_i\}_{t,n} \leftarrow \mathsf{SecShare.Sub}(\gamma', \gamma)$
    3. Party $\mathcal{DA}_i$ sends to $U$ its share $\alpha_i$
    4. $U$ interpolates the value $\alpha$ from the $\alpha_i$

ElGamal.DistDec$(c_1, c_2, \{x_i\}_{t,n})$
    1. $\{d_i\}_{t,n} \leftarrow \mathsf{SecShare.Exp}(c_1, x)$
    2. Recover $d$ from the $\{d_i\}_{t,n}$
    3. Output $m \leftarrow c_2/d$

ElGamal.DistDecString$(c_1, c_2, \{x_i\}_{t,n})$
    1. $\{d_i\}_{t,n} \leftarrow \mathsf{SecShare.Exp}(c_1, x)$
    2. Recover $d$ from the $\{d_i\}_{t,n}$
    3. Output $m \leftarrow \mathsf{AuthDec}(\mathsf{H}(d), c_2)$
```

Fig. 3. Distributed BBS and ElGamal Operations

6 Private Multiset Operations

First, we introduce a polynomial representation of multisets, specifying multiset operations in terms of polynomial operations. Then, we provide privacy-preserving polynomials using our additive encoding and show how to carry out the needed computations under encryption. The construction follows the basic construction of [25], but our quorum operation and distributed evaluation are new.

6.1 Multisets via Polynomials

Represent a finite multiset S of elements from $\mathbb{F}_p$ by

$$F = \prod_{s_i \in S} (x - s_i).$$

In this language, the empty set corresponds to the identity polynomial. Adding an element s corresponds to the multiplication of F by $(x - s)$. Performing a quorum test, i.e. determining if s appears q times in the set S, corresponds to checking if $(x - s)^q$ divides F. We propose an alternate test: check if the $(q - 1)^{\text{th}}$-derivative of F evaluated at s is zero. It is this test that we shall implement privately.

Note that the quorum test is complete, but not sound: as a concrete example consider the multiset $S = \{1, 1, 2\}$. This set is encoded by the polynomial

$$F(x) = (x-1)^2(x-2).$$

To check for quorums of size 2, take the first derivative:

$$F^{(1)}(x) = (x-1)(3x-5).$$

The derivative is 0 at $x = 1$ because 1 has multiplicity 2 in S. However, it is also 0 at $x = 5/3$ yet this value has multiplicity 0 in S.

Instead of failing on some fixed inputs, we will randomize the test to succeed with high probability on every input. The randomized version of the test will sample R a random polynomial of degree $q-1$, and then take the derivative of RF. We will show that for any input, this randomized test has soundness $1 - 1/p$.

Lemma 6.1. *PolySet.Quorum has perfect correctness.*

Proof. By construction s occurs q times in $\mathrm{Set}(F)$ if and only if $(x-s)^q \mid F$. Writing $F = (x-s)^q G$, the correctness of PolySet.Quorum follows using the product rule for $(q-1)^{\text{th}}$ derivatives:

$$(R(x-s)^q G)^{(q-1)}[s] = \sum_{k=0}^{q-1} \binom{q-1}{k} (RG)^{(q-1-k)}[s]((x-s)^q)^{(k)}[s].$$

Observe that $((x-s)^q)^{(k)}[s] = 0$ for $k < q$. □

Lemma 6.2. *If R is uniformly random in $\mathbb{F}_p^{q-1}[x]$ then $v = \left(R^{(k)}[s]\right)_{k=0}^{q-1}$ is uniformly random in $\mathbb{F}_p^q$.*

Proof. From a calculus lens: the value of v suffices to give the degree q Taylor expansion of R at s; then since R is a degree q polynomial this expansion is exact. Thus there is a bijection from choices of R to choices of v.

A linear algebra argument gives this bijection explicitly: the coefficients of R and the derivatives evaluated at s are related by the matrix below. It is invertible since the determinant is $(q-1)!(q-2)!\dots 1$.

$$\begin{bmatrix} (q{-}1)! & 0 & \dots & 0 & 0 \\ (q{-}1)!s & (q{-}2)! & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (q{-}1)s^{q-2} & (q{-}2)s^{q-3} & \dots & 1 & 0 \\ s^{q-1} & s^{q-2} & \dots & s & 1 \end{bmatrix} \begin{bmatrix} R_{q-1} \\ R_{q-2} \\ \vdots \\ R_1 \\ R_0 \end{bmatrix} = \begin{bmatrix} R^{(q-1)}[s] \\ R^{(q-2)}[s] \\ \vdots \\ R^{(1)}[s] \\ R[s] \end{bmatrix}$$

□

Lemma 6.3. *Fix any $F \in \mathbb{F}_p[x]$ and $s \in \mathbb{F}_p$. Let $R \in_R \mathbb{F}_p^{(q-1)}[x]$ and define $G = RF$. Then either*

1. *$(x-s)^q \mid F$ or*
2. *$G^{(q-1)}[s] = 0$ with probability $1/p$.*

Proof. Again invoke the product rule. Observe s is a root of $G^{(q-1)}$ if and only if

$$0 = \sum_{k=0}^{q-1} \left(\binom{q-1}{k} F^{(k)}[s] \right) R^{(q-1-k)}[s].$$

Now apply Lemma 6.2 to the case where the $\{F^{(k)}[s]\}_{k=0}^{q-1}$ are not all zero; this immediately gives that $G^{(q-1)}[s] \in_{\mathrm{R}} \mathbb{F}_p$ giving property 2. Otherwise, $F^{(k)}[s] = 0$ for all $k \in \{0, \dots, q-1\}$, so s is a root of F with degree $n+1$ giving property 1. □

Lemmas 6.1 and 6.3 give that PolySet.Quorum has perfect completeness and soundness $1 - 1/p$. However, we need stronger guarantees to use this algorithm in a setting where both the elements of the multiset and the location to perform the quorum test are chosen adversarially. This will again follow from the lemma. We write this guarantee in the following security game:

Lemma 6.4. *For any adversary A define*

$$\epsilon(A) = \Pr[A \to (F, s) : \textit{PolySet.Quorum}(F, s) = 0]$$

If $(x-s)^q \nmid F$ then $\epsilon(A) \leq 1/p$.

Proof. R was chosen uniformly, independently of the choice of F and s. By Lemma 6.3, for all F and s, we have $(RF)^{(q-1)}[s] = 0$ with probability $1/p$. □

6.2 Privacy-preserving Polynomials

As the polynomial in our application represents the multiset of accused parties, we need to keep the polynomial secret. We will do this by representing the polynomial as encodings of its coefficients. Furthermore, we will show how to both extend the polynomial when a new element is added to the multiset and how to compute the derivative and its evaluation at a given point.

The encoding of a polynomial $F = \sum_{i=0}^{d} F_i x^i$ will be set to the encoding of each of its coefficient. Recall that the encoding of an element s denoted e_s is the ElGamal encryption of g^s. Thus, the encoding of F, denoted e_F is defined to be $\{e_{F_0}, \dots, e_{F_d}\}$.

```
Inputs let $F, G \in \mathbb{F}_p^d[x]$, $R \in \mathbb{F}_p^q[x]$, $s \in \mathbb{F}_p$, $n \in \mathbb{N}$

PrivatePoly.Subtract($e_F, e_G$)
    1. For $i \in [0, \ldots, d]$:
           $e_{H_i} \leftarrow e_{F_i}(e_{G_i})^{-1}$
    2. Output $(e_{H_0}, \ldots, e_{H_d})$

PrivatePoly.Differentiate($e_F, n$)
    1. For $i \in [n, \ldots, d]$:
       (a) $c \leftarrow i!/(i-n)!$.
       (b) $e_{H_{i-n}} \leftarrow e_{F_i}^c$
    2. Output $(e_{H_0}, \ldots, e_{H_{d-n}})$

PrivatePoly.Multiply($e_F, R$)
    1. $e_G \leftarrow (e_{g^0}, \ldots, e_{g^0})$ with $d + q$ terms.
    2. For $i, j \in [d]$:
           $e_{G_{i+j}} \leftarrow e_{G_{i+j}}(e_{G_i})^{R_j}$
    3. Output $C_G$.

PrivatePoly.MultiplyLinear($e_F, \{s\}$)
    1. $e_G \leftarrow (e_{g^0}, e_{F_0}, \ldots, e_{F_d})$
    2. For $i \in [1, d]$:
           $e_{H_i} \leftarrow$ SecShare.ExpRR($e_{F_i}, s$)
    3. Output PrivatePoly.Subtract($e_G, e_H$)

PrivatePoly.ZeroTest($e_F, \{s\}$)
    1. $\{r_1\} \leftarrow$ SecShare.Gen()
    2. $\{t\} \leftarrow$ SecShare.Invert($\{r_1\}$).
    3. For $i \in [2, \ldots, t]$:
           $\{r_i\} \leftarrow$ SecShare.Mult($\{r_{i-1}\}, \{r_1\}$)
    4. Output $R \leftarrow (\{t\}, \{r_1\}, \ldots, \{r_t\})$
    5. $\{x\} \leftarrow$ SecShare.Mult($\{t\}, \{s\}$)
    6. $x \leftarrow$ SecShare.Reconstruct($\{x\}$)
    7. $c_y \leftarrow e_{F_0}$.
    8. For $i \in [1, \ldots, t]$:
       (a) $\{s_i\} \leftarrow x^i\{r_i\}$
       (b) $c_{y_i} \leftarrow$ SecShare.ExpRR($e_{F_i}, \{s_i\}$)
       (c) $c_y \leftarrow c_y c_{y_i}$
    9. Output True iff ElGamal.DistDec($y$) $= g^0$
```

Fig. 4. Operations on encoded polynomials.

Operations on encoded polynomials. In order to execute the set operations using encoded polynomials, we first need to implement a number of primitive operations. We describe the supported operations below. All operations are given in pseudo-code in Fig. 4. The design will be aided by the fact that the encoding is additive. In particular we implement the following.

PrivatePoly.Subract(e_F, e_G): output e_H such that $H = F - G$.

PrivatePoly.Differentiate(e_F, n): output e_H such that $H = F^{(n)}$, i.e. the n^{th} derivative of F.

PrivatePoly.Multiply(e_F, R): output e_H such that $H = FR$. In this case the polynomial R is in the clear, i.e. its coefficients are known to all parties.

We need two more operations that are a bit more complicated to compute because they entail multiplication of an encoding of a coefficient, i.e. e_y, by a secret value s. To achieve this multiplication, we assume that s has been secret shared among the parties and we raise the group element to the shared power using SecShare.ExpRR (Section 3.2). These combined operations will be utilized to compute a multiplication and will enable our remaining two operations.

Before proceeding we clarify a point about our notation. Our encoding of y is in fact two group elements, i.e. $e_y = (e_0, e_1)$, and we slightly abuse notation by calling SecShare.ExpRR($e_y, \{s\}$) where it is intended to mean:

$$(\text{SecShare.ExpRR}(e_0, \{s\}), \text{SecShare.ExpRR}(e_1, \{s\})).$$

PrivatePoly.MultiplyLinear(e_F, s): output e_H such that $H = (x - s)F$. We first use the above secret exponentiation procedure to compute the coefficients of sF. Then shifting the coefficients of F gives an encryption of xF and subtracting gives $xF - sF = (x - s)F$.

PrivatePoly.ZeroTest($e_F, \{s\}$): output True iff $F[s] = 0$. This protocol will compute an encoding of $F[s] = \sum_{i=0}^{d} F_i s^i$. Again the above procedure suffices: given sharings of s^i for each $1 \leq i \leq d$, it is easy to compute $e_{F[s]} = \Pi_{i=0}^{d}(e_{F_i})^{s^i}$. It remains to generate these sharings of the s^i.

The parties first generate sharings of a random value r and all its powers up to some constant $\ell \geq d$, say $r, r^2, ..., r^\ell$. This is done by starting with a sharing of r and multiplying it repeatedly to create the ℓ sharings. Then they compute the inverse of r using [4] yielding a sharing of the value r^{-1}.

Now given the sharing of the evaluation point s we proceed as follows. Multiply the sharing of s with the sharing of r^{-1} and reconstruct this value in the clear, exposing $x = s/r$. Computing in the clear each party can compute $x^2, \ldots, x^\ell$. Taking x^i and multiplying it by the sharing of r^i obtain a sharing of $x^i r^i = (s/r)^i r^i = s^i$.

Lines 1 through 4 of ZeroTest, in which the value R is derived, constitute a pre-processing step. Note this step is independent of F and s. Thus it can be run in advance. By running $\Omega(d)$ copies of the pre-processing step in parallel, we note that the algorithm has amortized constant round complexity.

Application to Multisets. Given these operations on polynomials, we define Set.Init, Set.Add, Set.Quorum by taking the multi-set operations defined using polynomials and replacing each polynomial operation with the privacy-preserving variant defined in this section. This is shown explicitly in Fig. 5.

```
Set.Init:  1. Output $e_{g^0}$

Set.Add($e_F, \{s\}$):
    1. Output PrivatePoly.MultiplyLinear($e_F, \{s\}$)

Set.Quorum($e_F, \{s\}$):
    1. For $i \in [q]$:
           $\{R_i\}, e_{R_i} \leftarrow$ SecShare.Gen()
    2. $e_R \leftarrow (e_{R_0}, \ldots, e_{R_{q-1}})$
    3. $e_G \leftarrow$ PrivatePoly.Multiply($e_F, e_R$)
    4. $e_H \leftarrow$ PrivatePoly.Differentiate($e_G, q-1$)
    5. Output PrivatePoly.ZeroTest($e_H, \{s\}$)
```

Fig. 5. Privacy-preserving multiset operations.

Lemma 6.5. *Let $\mathcal{A}$ be an attacker controlling at most t servers of the $\mathcal{DA}$ and assume that the encoding of s is secure as defined in Section 3.1. Then the above encoding of a polynomial representing the multiset S and the Set.Add operation on the encoded polynomial hide the elements of the set (revealing only the size of the set). Furthermore, the correctness of Set.Add, PrivatePoly.Differentiate, PrivatePoly.ZeroTest correlates to the equivalent operations carried out in the clear on the set as defined in Section 6.1.*

7 Efficiency

We provide some general runtime properties of the WhoToo protocol. These properties suggest the protocol could be implemented efficiently. Throughout our analysis, let N be the total number of accusations in the system, n the number of servers in the $\mathcal{DA}$, and q the quorum. We will consider all group operations as constant time. The choice of group is an important factor in runtime, as many exponentiations and multiplications are performed. The cost of the pairing operation is less significant, as only one pairing is performed per accusation. Further, note that all threshold operations take a constant number of rounds and a run time that scales polynomially with the threshold. Again, the exact performance characteristics of the threshold operations is an important factor in the performance of the system.

User. The protocol is designed for users with very limited resources. To create an accusation, the user will perform $O(n)$ secret sharings, BBS signatures, and ElGamal operations, each of which consists of a small constant number of exponentiations and multiplications. They perform no pairings. Further note that the protocol is non-interactive: after the user submits an accusation, they can go off-line. Online presence for a given user is only needed for and during the submission of an accusation by that user.

Server. In order to validate an accusation, each server performs a constant number of group operations. This includes a single pairing used to verify the BBS signature. In order to check for duplicates, the equality testing protocol must be run N times. Each run requires a constant number of group operations and rounds of communication; by running the equality checks in parallel, this yields a constant number of rounds of communication. The servers must also run the set add and quorum operations which for a quorum of q require $O(qN)$ group operations and amortized constant rounds of communication. Note that accusations can be submitted concurrently by different users but the processing by the servers needs to be sequential as the addition of elements to the set of identities is done one by one.

We do not have measurements of concrete runtime. A proof-of-concept implementation would provide further evidence that the protocol is feasible in practice. Thus such an implementation would be valuable future work.

Acknowledgments.

We thank the reviewers of PETS 2019, particularly the amazing work by our shepherd Wouter Lueks. The WhoToo name for our protocol was suggested by Sharon Rabin-Margaliot. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] Tor project, www.torproject.org

[2] Abdalla, M., Bellare, M., Rogaway, P.: The oracle Diffie-Hellman assumptions and an analysis of DHIES. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 143–158. Springer, Heidelberg (Apr 2001)

[3] Aranha, D.: Pairings are not dead, just resting. In: Workshop on Elliptic Curve Cryptography (2017)

[4] Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: Rudnicki, P. (ed.) 8th ACM PODC. pp. 201–209. ACM (Aug 1989)

[5] Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 614–629. Springer, Heidelberg (May 2003)

[6] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th ACM STOC. pp. 1–10. ACM Press (May 1988)

[7] Blömer, J., Juhnke, J., Löken, N.: Short group signatures with distributed traceability. In: Kotsireas, I.S., Rump, S.M., Yap, C.K. (eds.) MACIS. pp. 166–180. Springer, Cham (2016)

[8] Boneh, D.: personal communications (March 2019), see https://cryptobook.us/

[9] Boneh, D., Boyen, X.: Short signatures without random oracles. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 56–73. Springer, Heidelberg (May 2004)

[10] Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 41–55. Springer, Heidelberg (Aug 2004)

[11] Bowe, S.: Bls12-381: New zk-snark elliptic curve construction (October 2018), https://z.cash/blog/new-snark-curve

[12] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)

[13] Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In: 26th FOCS. pp. 383–395. IEEE Computer Society Press (Oct 1985)

[14] Cohen, J.D., Fischer, M.J.: A robust and verifiable cryptographically secure election scheme (extended abstract). In: 26th FOCS. pp. 372–382. IEEE Computer Society Press (Oct 1985)

[15] Donegan, M.: I started the media men list. The Cut (Jan 2018), https://www.thecut.com/2018/01/moira-donegan-i-started-the-media-men-list.html

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

[17] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO'86. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (Aug 1987)

[18] Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (May 2004)

[19] Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Robust threshold DSS signatures. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 354–371. Springer, Heidelberg (May 1996)

[20] Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. Journal of Cryptology 20(1), 51–83 (Jan 2007)

[21] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In: Coan, B.A., Afek, Y. (eds.) 17th ACM PODC. pp. 101–111. ACM (Jun / Jul 1998)

[22] Hoepman, J.H., Galindo, D.: Non-interactive distributed encryption: A new primitive for revocable privacy. In: Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society. pp. 81–92. WPES '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046556.2046567

[23] Jakobsson, M., Juels, A.: Mix and match: Secure function evaluation via ciphertexts. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 162–177. Springer, Heidelberg (Dec 2000)

[24] Kiayias, A., Yung, M.: The vector-ballot e-voting approach. In: Juels, A. (ed.) FC 2004. LNCS, vol. 3110, pp. 72–89. Springer, Heidelberg (Feb 2004)

[25] Kissner, L., Song, D.X.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (Aug 2005)

[26] Lueks, W., Hoepman, J.H., Kursawe, K.: Forward-secure distributed encryption. In: De Cristofaro, E., Murdoch, S.J. (eds.) Privacy Enhancing Technologies. pp. 123–142. Springer International Publishing, Cham (2014)

[27] Maurer, U.M.: Unifying zero-knowledge proofs of knowledge. In: Preneel, B. (ed.) AFRICACRYPT 09. LNCS, vol. 5580, pp. 272–286. Springer, Heidelberg (Jun 2009)

[28] Menezes, A., Sarkar, P., Singh, S.: Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography. In: Mycrypt (2016)

[29] Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (Aug 1992)

[30] Rajan, A., Qin, L., Archer, D.W., Boneh, D., Lepoint, T., Varia, M.: Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In: Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies. pp. 49:1–49:4. COMPASS '18, ACM, New York, NY, USA (2018)

[31] Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology 4(3), 161–174 (1991)

A Appendix

A.1 Computational assumptions

Decisional Diffie-Hellman (DDH). Define

$$\mathcal{D}_0 := (g, h, g^a, h^a), \quad \mathcal{D}_1 := (g, h, g^a, \eta)$$

for $a \in_{\mathrm{R}} \mathbb{Z}_p$ and $g, h, \eta \in_{\mathrm{R}} \mathbb{G}_1$. The DDH assumption states that no polynomial time algorithm can distinguish the two distributions with non-negligible advantage. This assumption, when made of the group $\mathbb{G}_1$ with an appropriate pairing $e : G_1 \times G_2 \to G_T$ is known as the *external Diffie-Hellman assumption* (XDH).

Strong computational Diffie-Hellman (sCDH). Define

$$\mathrm{DDH}_{g,h} := \{(g^a, h^a) \; : \; a \in \mathbb{Z}_p\}.$$

For any oracle algorithm $\mathcal{A}^{(\cdot)}$ define

$$\epsilon_{\mathrm{SCLDH}}(\mathcal{A}) := \Pr\bigl[\mathcal{A}^{\mathrm{DDH}_{g,h}}(g, h, g^a) = h^a\bigr]$$

for $a \in_{\mathrm{R}} \mathbb{Z}_p$, $g, h \in_{\mathrm{R}} \mathbb{G}_1$. The sCDH assumption states that $\epsilon_{\mathrm{SCLHD}}(\mathcal{A})$ is negligible for any polynomial time oracle algorithm. This captures the idea that computing h^a given g^a is hard, even with an oracle for the decisional variant. This assumption was first made by Abdalla, Bellare, and Rogaway, and is unrelated to the similarly named assumption in the paragraph below [2].

***k*-strong Diffie-Hellman (sDH$_k$).** Fix some $k \in \mathbb{Z}$. For any algorithm $\mathcal{A}$, define

$$\epsilon_{\mathsf{SDH}}(\mathcal{A}) := \Pr\left[\exists x : \mathcal{A}\left(g_1, g_2, g_2^{\gamma}, \ldots, g_2^{\gamma^k}\right) = \left(g_1^{\frac{1}{\gamma+x}}, x\right)\right]$$

for $g_2 \in_{\mathrm{R}} \mathbb{G}_2$ (with $g_1 \leftarrow \psi(g_2)$), $\gamma \in \mathbb{Z}_p^*$. The sDH$_k$ assumption states that $\epsilon_{\mathsf{SDH}}(\mathcal{A})$ is negligible for any polynomial time algorithm. This assumption is used by Boneh and Boyen in their short signature scheme, where they prove it holds in generic groups [9].

Concrete choice of curves. We propose the pairing-friendly family of curves BLS12 to instantiate our bilinear group. The XDH assumptions are believed to hold for this group. Concretely, a choice of prime p of bit-length 384 is recommended for 128-bit security [28]. Such curves are efficient to implement at a variety of security levels, as demonstrated by the use of BLS12-381 by Zcash [11]. Specific runtimes of operations in various BLS12 curves using the RELIC software library have been computed [3].

The BLS12 curves are not known to have an efficiently computable isomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$. However, our protocol uses such an isomorphism to select $g_1 = \phi(g_2)$. Without the isomorphism, the servers can choose g_1 in $\mathbb{G}_1$, g_2 in $\mathbb{G}_2$ independently and proceed as usual. Because the proof of correctness does not use ϕ, correctness of the scheme is unaffected by this change. The security of the modified scheme follows from slightly stronger assumptions. Concretely, we augment the DDH and k-strong DH assumptions to accept oracle adversaries. We assume the adversary has a negligible advantage even when the oracle is instantiated with the (unique) isomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$ such that $g_1 = \phi(g_2)$. These stronger assumptions are conjectured to hold in the BLS-type groups [8].

A.2 Equality Testing

Once we have established that there are more than q accusations for a given $s = \mathsf{H}(D)$ we need to find the other accusation against s. Recall that the accusations are encoded and as we want to preserve the privacy of the accusation we need to carry out this computation over the encodings. This must be done without leaking any other information about the accused. Given that the encoding scheme is additively homomorphic, a naïve method would be to divide the two ciphers, decrypt the quotient, and check if the result is 1. This would indicate that the two original messages were identical. However, if they were not the same, this procedure would reveal the encoding of the difference of the two messages. Thus, we need to add a randomization step; this amounts to raising the quotient to a random power before decryption.

The description above works for arbitrary encoding schemes. For concreteness and simplicity, we present the Equal protocol using the encoding which we have chosen which is based on ElGamal encryption in Fig. 6. Assume g, h are the ElGamal public key, and that the servers hold a distributed private key as in Section 5. Furthermore, they hold e_1, e_2 that are encodings of s_1, s_2. The following protocol is from [23]. The proof of knowledge is instantiated in the language of [27] and uses the Fiat-Shamir heuristic [17] for non-interactivity. We provide the details of $\mathsf{Equal}(\mathrm{sk}, e_1, e_2)$ in our distributed setting in Fig. 6.

```
$\mathsf{Equal}(\mathrm{sk}, e_1, e_2)$
    1. Output True if $\mathsf{MsgRand}(e_1 e_2^{-1}) = 1$
    2. False otherwise.

$\mathsf{MsgRand}(c = (c_1, c_2))$
    1. Each server chooses $r_i \in_{\mathrm{R}} \mathbb{Z}_p$ and publishes
       $a^i = (c_1^{r_i}, c_2^{r_i})$, and $\pi_i \leftarrow \mathsf{Prove}_{c_1,c_2}(a^i, r_i)$.
    2. Each server runs, for each $i$:
           If $\mathsf{Verify}_{c_1,c_2}(\pi_i, a^i)$ rejects: $a^i \leftarrow (1, 1)$.
    3. $c' \leftarrow (\prod_{i=1}^{n} a_1^i, \prod_{i=1}^{n} a_2^i)$.
    4. Output $m' \leftarrow \mathsf{ElGamal.DistDec}(c')$.

$\mathsf{Prove}_{c_1,c_2}(a = (a_1, a_2), r)$
    1. $k \in_{\mathrm{R}} \mathbb{Z}_p$
    2. $b_i \leftarrow c_i^k$
    3. $\alpha \leftarrow \mathsf{H}(a_1, a_2, b_1, b_2, c_1, c_2)$
    4. $\beta \leftarrow \alpha k + r$
    5. Output $\pi \leftarrow (\beta, b_1, b_2)$

$\mathsf{Verify}_{c_1,c_2}(\pi_i, a^i)$
    1. $(\beta, b_1, b_2) \leftarrow \pi_i$
    2. $\alpha \leftarrow \mathsf{H}(a_1, a_2, b_1, b_2, c_1, c_2)$
    3. Accept iff $c_i^{\beta} = a_i b_i^{\alpha}$
```

Fig. 6. Equality testing using message randomization; from [23]

We start by defining a randomization functionality MsgRand as follows:

$$\mathsf{MsgRand}(e) = \begin{cases} 1 & \text{if } e = 1 \\ e^r \in_{\mathrm{R}} \mathbb{G} & \text{otherwise.} \end{cases}$$

Now if we define our equality testing to return True if $\mathsf{MsgRand}(e_1 e_2^{-1})$ returns 1 and False otherwise we will have our desired equality testing, i.e. it will always output True when its inputs are the same and with probability $1 - 1/p$ it will output False if the encodings are of different messages. Further, it leaks no additional information. The following claim adapted from [23] gives the security of the MsgRand.

Claim A.1. *MsgRand is a secure multi-party computation of F when the threshold for secret sharing is met.*

A.3 Proof of plaintext knowledge

We specify the zero-knowledge proof of plaintext knowledge for the ElGamal encryption functions ElGamal.Enc and ElGamal.Dec from Section 3.1. Given a ciphertext $c = (c_1, c_2)$, the proof of knowledge of m such that c is the encryption of m reduces to proving knowledge of $\log_g(c_1)$. The standard proof of knowledge is known as Schnorr's protocol [31]. We make this proof non-interactive by applying the Fiat-Shamir heuristic [17] into the usually scheme, incorporating a value ρ into the hash input in Step 2. This serves to authenticate the proof: given c, m, ρ and $\pi \leftarrow \mathsf{PPK}(c, a, b, \rho)$ it is hard to find $\rho' \neq \rho$ and π' such that $\mathsf{VerifyPPK}(\pi', c, \rho')$ accepts. The non-interactive proof is specified in Fig. 7.

```
Parameters: $g, h, c = (g^a, h^a m)$

$\mathsf{PPK}(c, a, \rho)$
    1. $r \in_{\mathrm{R}} \mathbb{Z}_p$
    2. $t \leftarrow g^r$
    3. $z \leftarrow \mathsf{H}(c, t, \rho)$
    4. $s \leftarrow r + za$
    5. Output $\pi \leftarrow (t, s)$

$\mathsf{VerifyPPK}(\pi, c, \rho)$
    1. $z \leftarrow \mathsf{H}(c, t, \rho)$
    2. Accept iff $g^s = t c_1^z$
```

Fig. 7. ElGamal proofs of plaintext knowledge.

A.4 BBS signature specification

Here we give the full specification of the BBS.Sign and BBS.Verify procedures of the BBS signature scheme, adapted for use with ElGamal encryption. The notation is slightly changed from the original in order to make the role of the encryption of A clear. After fixing u, v, g_2, w, c_1, c_2 define the homomorphism $\phi : \mathbb{Z}_p^5 \to \mathbb{G}_1 \times \mathbb{G}_T \times \mathbb{G}_1$ given by $\phi : (t_1, t_2, t_3) \mapsto T = (T_1, T_2, T_3)$ where

$$T_1 = u^{t_1}, \quad T_2 = c_1^{t_2} u^{-t_3},$$
$$T_3 = e(c_2, g_2)^{t_2} e(v, w)^{-t_1} e(v, g_2)^{-t_3}.$$

Given this definition of ϕ the BBS.Sign and BBS.Verify operations given in Fig. 8, together with ElGamal decryption to trace identities, constitute the BBS group signature scheme.

```
Public key: $((u, v), g_1, g_2, w)$

BBS.Sign($\mathrm{sk}_U, m$)
    $(A, \alpha) \leftarrow \mathrm{sk}_U$
    $a \in_{\mathrm{R}} \mathbb{Z}_p$
    $c_A \leftarrow (u^a, v^a A)$
    $r \in_{\mathrm{R}} \mathbb{Z}_p^3$
    $R \leftarrow \phi(r)$
    $z \leftarrow \mathsf{H}(m, c_A, R)$
    $t \leftarrow z(a, \alpha, a\alpha)$
    $s \leftarrow r + t$
    $\sigma \leftarrow (z, s)$
    Output $(c_A, \sigma)$

BBS.Verify($m, c_A, \sigma$)
    $(c_1, c_2) \leftarrow c_A$
    $(z, s) \leftarrow \sigma$
    $S \leftarrow \phi(s)$
    $T \leftarrow (c_1^z, 1, (e(g_1, g_2)/e(c_2, w))^z)$
    $\widetilde{R} \leftarrow S/T$
    Accept iff $z = \mathsf{H}(m, c_A, \widetilde{R})$
```

Fig. 8. BBS Sign and Verify Procedures

B Security

We informally outline a security proof for the WhoToo protocol defined in previous sections assuming the security of of its component blocks in a composable security model. To make this outline into a formal proof, one would frame the ideal functionalities and simulation arguments below in the UC model [12] or similar.

B.1 Ideal functionality

The ideal functionality $\mathcal{F}_{\mathsf{WhoToo}}$ in Fig. 9, when executed by a trusted party, can be seen as a formalization of the envelope scheme presented in the introduction and as a basis for defining the required security from the WhoToo protocol. That is, one is to show that any real-world adversary against the WhoToo protocol can be simulated by the ideal adversary acting against $\mathcal{F}_{\mathsf{WhoToo}}$.

Note that Investigate takes as input the name of the accusers and accused in the clear, hence we can think of these values as given to the ideal adversary. This means that the simulation needs to simulate the view of the adversary given these "leaked" values.

```
Ideal.Initialize
    $\mathrm{ValidAccusers} \leftarrow \mathsf{GetUsers}()$
    $\mathrm{Accusers} \leftarrow \varnothing$

Ideal.Accuse
    Receive input $D$ from a user $U$
    If $U \notin \mathrm{ValidAccusers}$: halt
    $\mathrm{Accusers}[D] \leftarrow \mathrm{Accusers}[D] \cup \{U\}$
    If $\#\,\mathrm{Accusers}[D] \geq q$:
        Run $\mathsf{Investigate}(D, \mathrm{Accusers}[D])$
```

Fig. 9. Ideal functionality $\mathcal{F}_{\mathsf{WhoToo}}$

B.2 Threat model

Terminology. Let "servers" refer to the parties that constitute the distributed authority. Let "user" refer to any party which receives a key during initialization or sends a (correctly or incorrectly formed) accusation to the distributed authority.

Network model. Stipulate that DistBBS.UserKeyIssue occurs over an *authenticated* and *confidential* channel. This assures that the servers send keys to the right users and no one else observes them. It also assures that users know they are receiving keys from the intended servers. Similarly, stipulate that accusations are submitted over *anonymous* and *confidential* channels to the respective servers. Anonymity assures no information is leaked due to metadata; i.e. the adversary will not know who sent a message simply by observing communication over the network. Confidentiality assures that each server sees only its share of the secrets shared in accusation.

User and server corruption. Assume the adversary can control any subset of corrupted users and up to t servers in the distributed authority. Consider a static model where the corrupted parties are chosen before initialization of the system and fixed throughout.

Excluded attacks. There are attacks inherent to any reporting system as ours that cannot be prevented by cryptographic means or made part of the formal security model, e.g., the submission of false accusations by malicious users in order to influence the accusation count against someone. Our solution, that allows binding accusations to real-world identities, facilitates accountability and disincentivizes such behavior hence limiting potential abuses of the system. See Section 2.

B.3 Security arguments

We sketch our security argument for the WhoToo protocol assuming the security of several of its components. Assume composable notions of security (and corresponding ideal functionalities) for the distributed BBS signatures, the privacy-preserving set operations from Section 6, the Equal functionality of Appendix A.2, and for threshold ElGamal encryption, including its related subroutines and extractable proofs of plaintext knowledge.

Theorem B.1. *Let $\mathcal{A}$ be an attacker controlling at most t servers of the $\mathcal{DA}$ and any number of corrupted users, and assume composable security notions for the above components. Then, the interaction between $\mathcal{A}$ and the WhoToo protocol can be simulated in the ideal model given $(D, \text{Accusers}[D])$ as output by Investigate.*

We sketch the proof. Fix a real attacker $\mathcal{A}_0$ against the WhoToo protocol. Our argument proceeds in steps. In each step we take an attacker $\mathcal{A}_i$ in some game and change its behavior to define $\mathcal{A}_{i+1}$ in another game. We argue that the final attacker is in fact against the ideal game. Arguing the view of $\mathcal{A}_i$ is simulatable given the view of $\mathcal{A}_{i+1}$ concludes the proof. The steps are:

1. Model subprotocols with trusted party.
2. Replace corrupted users with leaked BBS keys.
3. Begin induction on accusations.
4. Remove accusations that fail verification.
5. Remove accusations with mismatched encodings.
6. Simulate honest accusations.
7. Simulate dishonest accusations.
8. Replace leaked BBS keys with corrupted users.

We give details of each step.

1. Let WhoToo* be an augmentation of WhoToo protocol, replacing any SecShare.Operation, Set.Operation, and Equal calls with their corresponding ideal operations, executed by a trusted third party. If $\mathcal{A}$ can distinguish WhoToo from $\mathcal{F}_{\mathsf{WhoToo}}$ but cannot distinguish WhoToo* from $\mathcal{F}_{\mathsf{WhoToo}}$, then it can distinguish WhoToo from WhoToo*. This contradicts the security of the subprotocols. We have thus reduced to security against $\mathcal{A}_1$, an adversary against the WhoToo* game.

2. Let $U_1, \ldots, U_N$ be the users corrupted by $\mathcal{A}$. By the privacy guarantees of the DistBBS.UserKeyIssue protocol, even if some U_i does not follow the protocol, his view after completing the key issuing protocol is simulatable from the key (R_i, α_i) alone. Thus, $\mathcal{A}_1$ can be simulated by $\mathcal{A}_2$, where $\mathcal{A}_2$ is an adversary where $U_1, \ldots, U_N$ follow the DistBBS.UserKeyIssue protocol properly obtaining (R_i, α_i) but submit no accusations. Then, a single malicious party $\mathcal{A}_2$ is given inputs $\{(R_i, \alpha_i)\}_{i \in [N]}$ and proceeds to interact with the system by submitting accusations $\text{acc}_{i_1}, \ldots, \text{acc}_{i_m}$.

3. From here on, we will show that, assuming all prior accusations are simulatable in the ideal model, the next accusation is. This is again an appeal to composability.

4. Say acc is *ill-formed* if WhoToo.VerifyAcc(acc) → False. We say that an ill-formed accusation is of type I if it fails one of the initial verification checks of ElGamal.Verify, ElGamal.VerifyString, BBS.Verify, SecShare.Verify, and SecShare.CheckConsistent. By inspecting WhoToo.Accuse, it is clear that any such accusation does not change the state of the system. Since ElGamal.Verify, ElGamal.VerifyString, BBS.Verify and SecShare.Verify can be executed using only public parameters, they reveal nothing. Further by the idealization of the secret sharing scheme in Step 1 we know that SecShare.CheckConsistent reveals nothing as well. Thus, if any of these tests fail, the adversary can achieve the same view by running these tests locally instead of submitting a type I accusation.

The remaining case of ill-formed accusations, to which we refer to as *type II*, is when the operation $\mathsf{Equal}(\text{skeg}, (c_R, e_s), (c_{R'}, e_{s'}))$ outputs True. In the ideal execution of Equal, the adversary learns the single bit $b = 1 \iff (R, g^s) = (R', g^{s'})$. Let acc$'$ be the one that contains the pair $(R', g^{s'})$. We have two cases:

Case 1 : acc$'$ is an accusation submitted by the adversary. Since ElGamal.Verify passed, we know with all but negligible probability the PPK was valid. Similarly, we know that the BBS signature serves as a proof of knowledge of R. From this we conclude that either (a) the adversary generated acc$'$ hence the values $(R', g^{s'})$ can be extracted, or (b) that $c_{R'}, e_{s'}$ were copied from a previous accusation acc$''$ submitted by some user in which case $b = 1$ if and only if acc$'$ = acc$''$. In both cases, the adversary's action can be simulated.

Case 2 : acc$'$ was submitted by some other user. By the inductive hypothesis we know that either: (a) the adversary knows nothing about (R', s') or (b) the quorum for s was reached. In the former case, it is infeasible for the attacker to produce the required BBS signature. In the latter, $\mathcal{F}_{\mathsf{WhoToo}}$ reveals R', D'. In either case the actions are simulatable.

Thus we have reduced to an attacker $\mathcal{A}_4$ which submits no invalid accusations.

5. Let acc be a valid encoding. Put

$$s = \mathsf{SecShare.Reconstruct}(\omega),$$
$$D = \mathsf{HElGamal.StringDecrypt}(c_D).$$

We say acc has *matched accused encodings* when $s = \mathsf{H}(D)$. Otherwise, say it has mismatched accused encodings. We argue any adversary that submits accusations with mismatched accused encodings can be simulated by an adversary with matched ones. Fix a mismatched accusation. Two cases:

Case 1: the adversary called H with input x to obtain $s = \mathsf{H}(D')$. Since acc is mismatched, note that $D' \neq D$. Since the value c_D is never used until $\mathsf{WhoToo.OpenAccusations}$, observe that acc will be treated exactly as an accusation against D' until accusations are opened. Two further cases:

Case 1a: no accusation with $s' = s$ is matched. Thus $\mathsf{WhoToo.OpenAccusations}$ will halt and never call $\mathsf{Investigate}$. Since honest parties never generate mismatched accusations, we see that all of the accusations were generated by the adversary. The adversary could obtain the same view by submitting none of the accusations.

Case 1b: some accusation with $s' = s$ is matched. In this case, $\mathsf{WhoToo.OpenAccusations}$ will perform exactly as if all the accusations were matched. Thus the adversary could obtain the same view by submitting all matched accusations with $c_D = \mathsf{ElGamal.EncString}(D')$.

Case 2: the adversary never made a call to H that returned s. With overwhelming probability, $s \neq \mathsf{H}(D)$ for all D corresponding to the identities accused by honest accusers. Thus, if the adversary instead sampled a random D' and replaced all mismatched accusations containing a sharing of s with matched accusations containing a sharing of $s' = \mathsf{H}(D')$, the resulting view would be indistinguishable.

Thus we have reduced to an attacker $\mathcal{A}_5$ which only submits accusations with matched accused encodings.

6. Let acc be an accusation made by an honest user: a tuple $(c_R, e_s, c_D, \pi_0, \pi_1, \sigma, s_i)$. We know honestly generated accusations are processed identically in the real and ideal models by correctness of the set operations. Further, we know that the ideal set operations reveal nothing about the elements of the set (Lemma 6.5). It remains to show that the accusation itself is simulatable. By the security of ElGamal encryption, we know the encoding of s and encryption of D are indistinguishable from encryptions of random values. By the security of secret sharing we know that the view of the adversary's shares is simulatable and so is the pair (c_R, σ) by the anonymity property of group signatures,

7. Let acc be an accusation made by the dishonest party. Since by now, we have excluded accusations that fail verification, we know acc is valid and its group signature verifies. By the traceability property of group signatures, we have that $\mathsf{ElGamal.DistDec}(\mathrm{skeg}, c_R) \to R$ where R is the public key of some corrupted user. Further, since π_1 is a PPK for e_s, and we know that e_s was not copied from a previous accusation (we excluded this with type II ill-formed accusations), we can extract g^s. Further, by the validity of the sharing of s, the adversary in fact knows s. Similarly, extracting the PPK for c_D yields a string D. Since we excluded mismatched accusations, we know that $s = \mathsf{H}(D)$. Thus, in the ideal world, the adversary can submit the pair (R, D) to the authority.

8. The last step is to reverse the transformation made in Step 2. This is simple: each corrupted user can send his keys to the adversary, and the adversary can forward accusations through any corrupted user. Let $\mathcal{A}_8$ have this behavior.

Conclusion. Conclude that the view of $\mathcal{A}_8$ interacting with $\mathcal{F}_{\mathsf{WhoToo}}$ is indistinguishable from the view of $\mathcal{A}$ acting against the real-world WhoToo protocol.