PACIFIC: Privacy-preserving automated contact tracing featuring integrity against cloning

Griffy, Scott
Brown University
scott_griffy@brown.edu

Lysyanskaya, Anna
Brown University
anna_lysyanskaya@brown.edu

March 26, 2023

Abstract

To be useful and widely accepted, automated contact tracing / exposure notification schemes need to solve two problems at the same time: they need to protect the privacy of users while also protecting the users' from the behavior of a malicious adversary who may potentially cause a false alarm. In this paper, we provide, for the first time, an exposure notification construction that guarantees the same levels of privacy as existing schemes (notably, the same as CleverParrot of [CKL+20]), which also provides the following integrity guarantees: no malicious user can cause exposure warnings in two locations at the same time; and any uploaded exposure notifications must be recent, and not previously used. We provide these integrity guarantees while staying efficient by only requiring a single broadcast message to complete multiple contacts. Also, a user's upload remains linear in the number of contacts, similar to other schemes. Linear upload complexity is achieved with a new primitive: zero knowledge subset proofs over commitments. Our integrity guarantees are achieved with a new primitive as well: set commitments on equivalence classes. Both of which are of independent interest.

Contents

1 Introduction **3**
- 1.1 Automated contact tracing . . . 4
- 1.2 Attacks and related work . . . 5
  - 1.2.1 Integrity attacks . . . 5
  - 1.2.2 Privacy attacks . . . 7
- 1.3 Threat model and assumptions . . . 8
- 1.4 Our contribution . . . 10

2 Notation and preliminaries **12**
- 2.1 Notation . . . 12
- 2.2 Preliminaries . . . 12
  - 2.2.1 Non-interactive zero knowledge proofs of knowledge (NIZKs) . . . 13
  - 2.2.2 Mercurial Signatures . . . 13
  - 2.2.3 Verifiable random function . . . 14

3 Definitions **15**
- 3.1 PACIFIC : Privacy-preserving automated contact tracing featuring integrity against cloning . . . 15
- 3.2 PACIFIC security definitions . . . 16
  - 3.2.1 Clone integrity . . . 19
  - 3.2.2 Privacy definitions . . . 20
- 3.3 Set Commitments on equivalence classes (CoECs) . . . 22
- 3.4 Zero knowledge subset proofs over commitments (zk-SPoCs) . . . 23

4 Construction of ProvenParrot **25**
- 4.1 Description . . . 25
- 4.2 ProvenParrot . . . 27
- 4.3 Proofs of security . . . 30

Appendices **35**

A Additional preliminaries **35**
- A.1 Verifiable random functions (VRFs) . . . 35
- A.2 Mercurial signatures . . . 36
- A.3 FHS19 Commitments . . . 39

B Additional constructions **40**
- B.1 Set commitments on equivalence classes (CoEC) construction . . . 40
- B.2 Zero knowledge subset proofs over commitments (zk-SPoC) construction . . . 41
- B.3 Verifiable random functions for mercurial signatures . . . 42

C Formal definition of PACIFIC **43**
C.1 Clone integrity 45
C.2 Privacy definitions 46

D Security proofs **48**
D.1 ProvenParrot proofs 48
D.1.1 Proof of Clone integrity 48
D.1.2 Proof of chirp privacy 53
D.1.3 Proof of upload privacy 54
D.2 Proofs for CoECs 56
D.3 Proofs for zk-SPoCs 59

E Details **61**
E.1 Splitting uploads across batches 61
E.2 Necessity of VRF in chirp 61
E.3 Preventing de-anonymization with nonces 61

1 Introduction

In 2020, the COVID-19 virus spread across the world and as of this writing, the virus has claimed over 6 million lives [Org22]. Countries have responded with a number of measures such as: distributing masks and vaccines, requiring testing, and restricting travel. One of these measures is contact tracing. Contact tracing is the process of discovering who a person interacted with while they were contagious with a virus. This can help inform others to isolate to stop the spread of the virus. Contact tracing can be performed by interviewing patients who have been exposed and notifying those who the patient claims to have come into contact with. Automated contact tracing (also known as automated exposure notification) skips the need for an in-person interview, instead, relying on devices to track and notify users. This makes the contact tracing process more efficient. But ensuring the privacy and integrity of automated contact tracing simultaneously is not trivial. A contact tracing system that lacks privacy could deter people from using the system, reducing its effectiveness [ACK+20]. An authoritarian regime could also use the guise of public health to track civilians. These issues make privacy critical to ensuring an automated contact tracing system is effective and safe. But making contact tracing completely anonymous could have potential issues as well as unchecked users could wreak havoc on the system by spoofing outbreaks [RG20].

In this work, we introduce a new construction, "ProvenParrot," which goes further to ensure that it is difficult to create fake outbreaks. We also introduce a definition (PACIFIC) that, when met, ensures that no adversary can violate the privacy or integrity of the contact tracing scheme. We then prove our construction meets this definition.

1.1 Automated contact tracing

To give context to our discussion in the rest of this introduction, we first describe a simple automated contact tracing scheme. This simple automated exposure notification protocol uses a person's phone to broadcast random messages over bluetooth. Then, if a user tests positive for COVID-19, they upload all of the random messages they broadcasted to a public database where others can check to see if they were exposed. This naive scheme suffers from a multitude of problems, but gives the basic blueprint for a contact tracing scheme:

Scheme 1 (Example automated contact tracing scheme)

1. ***Chirp.*** *Users in close proximity exchange bluetooth messages (commonly referred to as "chirps"). These messages could be randomly generated bit strings.*

2. ***Listen.*** *Users listen and record any chirps nearby.*

3. ***Upload.*** *When they determine they're infected (through a test by a health authority), users upload some set of ciphertexts to a database (we refer to this upload as a "batch" of notifications). This set of ciphertexts could simply be the random bit strings heard.*

4. ***Check.*** *Users regularly check if the database suggests that they should quarantine (e.g. if some batch includes a chirp they made, indicating that they were in contact).*

The Chirp and Listen steps can occur many times before an upload occurs, allowing an uploader to notify many other users.

We can inspect this scheme to see that it will correctly notify users who were in contact with others, thus providing contact tracing functionality. But a malicious user could upload any chirps they heard, or inspect the database to link the chirps to messages they heard. If these malicious users took note of the people around them when they heard these messages, they could learn who was infected. A malicious user could then look at other chirps in the batch to determine other locations where that same user had been.

Upload-what-you-heard This naive scheme follows the "upload-what-you-heard" model used by [CKL$^+$20], where users upload chirps that they heard. Some existing schemes instead use a "upload-what-you-sent" model where instead, users upload chirps that they sent and users instead check the database for chirps they heard. This model is used by DP3T, Google and Apple's contact tracing scheme (sometimes referred to as "GAPPLE"), and ReBabbler [TPH$^+$20, GA20, CKL$^+$20] . Upload-what-you-sent models can be more efficient, but generally sacrifice privacy at upload time as users can link the upload to chirps they heard, so we'll be focusing on the upload-what-you-heard model for this paper.

1.2 Attacks and related work

In this section, we'll discuss a few schemes and attacks that are related to our scheme. There is a much larger body of work on contact tracing which we reference here [RAC+20, CFG+20, BRS20, PAJ20, BDH+20, DDL+20, Tan20, PR21, ABIV20, Vau20b, RBS21, DSM+22, NMD+22, WL20, CBB+20a, GhP+20, JBQ20, RBS20, CBB+20b, TSS+20, Tra20, Daw20]. To keep the discussion concise, we'll only highlight a few schemes in this section.

We'll now explain existing attacks in the literature, as well as describe our own attacks that affect existing schemes. We categorize these attacks into two categories: attacks on privacy and attacks on integrity.

1.2.1 Integrity attacks

Vaudenay [Vau20a] introduced the idea that an automated contact tracing scheme could be used to launch a sort of "terrorist attack" where an entire city could be falsely notified of being in contact with an infected person and thus need to quarantine. He also constructs a solution to this problem which requires an interactive chirp [Vau20a].

Gennaro et al [RG20] explains how a more targeted attack could create fake outbreaks in right or left leaning neighborhoods during the two-week period before an election to sway the outcome. By targeting a specific neighborhood known to vote a certain way, a malicious attacker could exploit a weakness in a contact tracing scheme in order to suggest those voters should quarantine during the election, thus affecting the outcome of the election.

We paraphrase the terrorist attack below, accounting for the politically targeted variant.

Attack 1 (Terrorist attack)

1. *A malicious user enrolls many devices in the contact tracing scheme.*
2. *The malicious user places the devices all over an area such as a city or neighborhood.*
3. *The malicious collects chirps in the area for a period of time on the order of weeks.*
4. *The malicious user uploads all chirps they heard, indicating that everyone in the area should isolate.*

We can see that our naive scheme in Scheme 1 is susceptible to this attack as the server does not do any checking to ensure that uploaders are honest. Thus, a malicious user can listen in areas across a city simultaneously and upload all chirps they heard. The mitigation to any integrity attack involves registering users so that the server can verify that the user is acting honestly during upload (e.g. imposing "upload limits," ensuring no user uploads too many times per week). In this paper, we'll refer to the authority registering

users as the "registration party." Many schemes [CKL+20, BCK+20] leverage a registration party in order to provide privacy guarantees.

Even if we register users, a terrorist attack can be facilitated by replaying and relaying chirps to other honest users. This is known as the "Chirp replay/relay Attack" [CKL+20]. A replay/relay attacker can listen to a chirp, then replay it at another time/place fooling another honest user into believing that the replayed chirp was honestly created and including it in their upload, thus falsely exposing an honest user.

Auerbach et al. [ACK+20] discovered another weakness in existing schemes called the "inverse-Sybil attack." This attack works by having many devices controlled by one adversary who listens in many different areas at the same time and then combines all chirps heard into one upload. Auerbach et al. provide a solution to the inverse-Sybil attack which relies on uploaders computing hash chains and assuming that the adversary cannot have their devices communicate with each other quickly.

We motivate the integrity definitions of [ACK+20] as well as our own integrity definition by posing a new attack: the *event attack*. This attack works similar to the voter variant of the terrorist attack [RG20] but takes the possible damage further by noting that it could be more effective to instead upload data that would indicate that everyone at a single event should quarantine. Examples include: a rally or a political convention which takes place in a shorter time (hours) than the period where an honest user would normally upload data for (multiple days). This attack would be more appealing for an adversary to pull off compared to the terrorist attack as the event attacker would only need to attend a single event where they know all attendees will be voting for one candidate, rather than patrolling a neighborhood for weeks to find enough contacts to upload.

We now describe a new attack: the event attack:

Attack 2 (Event attack)

1. *A malicious user enrolls many devices in the contact tracing scheme. This could be done by cloning the keys from a single device onto many devices.*
2. *The malicious user places the devices across the area of the event.*
3. *During the event (perhaps a few hours) the malicious user has each device listen for chirps in separate areas.*
4. *After the event, the malicious user combines all chirps heard and uploads all of the chirps in a single batch, thus falsely exposing everyone at the event.*

In order to pull off an event attack in a scheme with user registration, the attacker would need to "clone" their device, i.e., recover the key to copy to multiple devices such that they could use the scheme in different areas at the same time. We call the defence against this attack "clone protection" which explains the name of this paper. Applying upload limits to the event attack doesn't solve it as an adversary could allocate all of their permitted notifications

to a single event. An honest user would instead upload data that would be spread out over more time (maybe a week). *This concentration of notifications into a few hours is one weakness we address in our definition and construction.*

We also solve another problem not in the literature, the "upload replay/delay attack." This differs from the replay/relay attacks in the literature in that it focuses on replaying uploads rather than the chirps passed between devices. In this attack, a malicious user listens to a chirp, then uploads the chirp multiple times, thus increasing the severity of the contact. An extreme case of this attack could allow adversaries to continue to falsely expose an honest user of the scheme, resulting in the system showing that they should isolation indefinitely. Obviously, this false alarm would not be followed if it the recommendation were to isolate indefinitely, but this could be used to mask real exposure notifications by overloading the system with false ones. Further, if an adversary were careful about overdoing the number of false exposures, they could cause some false isolations for an honest user months after the actual contact. This attack works as follows:

Attack 3 (Upload replay/delay attack)

1. *A malicious user has a contact with the honest user, receiving a chirp from that user.*
2. *The malicious user then uploads that chirp to the database. If they can upload the same chirp multiple times to falsely increase the severity of their contact, then the scheme does not have upload replay protection.*
3. *The honest user then checks the database and learns that they had a contact with an infected user, potentially isolating if the contact was long enough.*
4. *The malicious user then waits for a few months and reuploads that same chirp without actually ever coming in contact with the honest user again.*
5. *The honest user then falsely believes that they were exposed again, and potentially isolates again. If this occurs, the malicious user has violated the upload delay integrity of the scheme.*

This problem does not arise in schemes where upload privacy is not a concern, since users checking the database can associate an upload with the time it happened, thus learning if the contact is old or a replay. But, while constructing a scheme which achieves upload privacy, it is possible to open the scheme up to this attack and an existing scheme suffers this problem [CKL+20].

1.2.2 Privacy attacks

Vaudenay lists a few attacks on the privacy of D3PT [Vau20a] which apply to any contact tracing scheme. We've included attacks from Avitabile et al. [ABIV20] and Canetti et al. [CKL+20] as well. In this paper, we'll focus on the strongest attacks from these papers, described below.

- The "Paparazzi" attack is where a malicious user listens to chirps from devices and then, uses these chirps to track users, such as linking chirps to uploads or linking multiple chirps together to track where a user has been.
- The "Matrix Attack" involves collusion with the government in various ways to de-anonymize users of the scheme by using all the information available to any authorities in the scheme and actively creates malicious chirps to try to de-anonymize users.
- The "Brutus" attack involves the authority leaking the mapping of a registration to users.
- The "Sybil" attack is where a user changes their identity for each interaction, thus tying any exposure to a specific interaction.

We can see that our naive scheme is susceptible to the paparazzi attack by a malicious user who remembers the time and location they were when they chirped each bit string. This attacker can then link these times and locations to notifications from the public database. We could potentially restrict users from downloading the database during the check function, and instead have users upload their sent chirps and have the server determine if they should quarantine. This solution suffers from the matrix attack as in this attack, the attacker can use the information on the server. Even if we prevent a single user from linking a notification to a specific chirp, an attacker could generate a new identity for each chirp (the "Sybil" attack), effectively acting as a different user for each chirp. Through correctness of the scheme, the attacker would be able to know which identity each notification was related to and thus which chirp. To solve this issue, we can add a registration step to the scheme where users' identities are first verified. This solution opens the door to the "Brutus attack" as the server now knows which registered users were infected.

In our definition, we combine the paparazzi, matrix, and Brutus attacks into one property: "upload privacy" and prove our construction secure in the presence of a semi-honest authority colluding with malicious devices. We also solve a variant of the paparazzi attack (where honest users do not upload) in the presence of a fully malicious authority. We call this second property: "chirp privacy" as it only holds for users who do not upload. This is a similar division of properties to [CKL+20].

1.3 Threat model and assumptions

Some schemes separate the health authority from the database owner and assume non-collusion in order to attain privacy properties [ABIV20]. In our construction, we assume that all central servers are pooling information to de-anonymize users, but run the server code faithfully. We pair this "semi-honest" central authority with fully malicious users who can chirp and upload any ciphertext they want. By "semi-honest" we mean the adversary knows all the information contained in the central database and can create and register new

devices which are fully malicious, but, *the registration secret key is kept secret and only used honestly.* Registration is pointless if the adversary has unrestricted access to the registration secret key, so this is a necessary assumption for any level of security. This is a useful definition since a real-world adversary would have an easier time misbehaving on their own device where they can run any code they want, but would likely have a harder time running a malicious central server. The central server could be made to be semi-honest through some technology such as multi-party computation or replacing the server with a blockchain. Running multi-party computation or blockchain across all the devices would be prohibitively expensive, but running this across a few central servers and assuming one of them is honest is better in terms of the strength of the assumption as well as efficiency. It is also easier to audit a central server, potentially discovering traces of misbehavior by admins of the system.

Time and location. We rely on location data (known as "measurements") to generate a notion of location for users [CKL+20]. Like in [CKL+20], our measurements include things like GPS data, background noise, and possibly information from wifi signals. Differing from [CKL+20], we separate time out from the list of measurements, treating it as a separate attribute. These measurements are used instead of simple GPS data since GPS isn't always reliable and using unpredictable data in the area also provides some integrity against users who might try to broadcast chirps to an area without being present there [CKL+20].

Tying attacks to resources. It's impossible to prevent a user from truly infecting themselves and going to an event to truly expose many people there. It is also a separate problem to verify that users have covid and it is possible that adversaries could get unverified uploads into the database or pay those with covid to misbehave [AFV20]. Thus, instead of stopping these attacks, we need to instead limit the amount of damage that cheating adversaries can do by tying the effectiveness of their attacks to real-world costs such as paying for a phone or bribing "covid mules" to upload maliciously crafted batches. To do this, we need to operate in a semi-honest model for the registration party in order to ensure that they don't create a separate identity for each interaction to trivially violate any privacy and integrity guarantees. Assuming a semi-honest registration party allows us to count the number of registrations the adversaries make (in our definition) and link the number of fake exposures and leakage to this number. While for upload privacy and integrity, we require a semi-honest authority, we do get chirp privacy for users who do not upload in the presence of a malicious authority. [1]

[1] We also allow our adversary to chirp in many areas at the same time with a single device. This might open our construction up to an attack, but is not as big of an attack as the paparazzi, matrix, or Brutus attacks as the adversary essentially "dilutes" the information they learn by chirping in multiple areas. This could be solved with techniques from [CHK+06] if this attack were deemed important.

Like many other contact tracing schemes [ABIV20, CKL+20], we do not consider privacy attacks on the BLE protocol itself, e.g. using power analysis or linking a bluetooth id. Solving these issues is an independent problem.

1.4 Our contribution

New contact tracing definition In this work, we create a new type of scheme with associated security games and definitions which we call privacy-preserving automated contact tracing featuring integrity against cloning (PACIFIC). This game models a real-world adversary interacting with honest users in a contact tracing scheme. We introduce integrity and privacy definitions which captures the paparazzi, matrix, Brutus, and event attacks described in Section 1.2. We'll address the privacy attacks simulating chirps and batch uploads in the game, restricting the simulator to only information that an honest user would gain. To address the event attack, we'll introduce the notion of clone protection. This will prevent an upload from exposing honest users in different locations at the same time.

In schemes without clone protection, an event attacker can use their entire upload limit to focus on an event, covering the entire event and falsely alerting everyone, making it seem like a super spreader event. With clone protection, each device the adversary has can only act like a single person at the event, as they cannot be in multiple places at the same time. Thus, to cover the whole event, they would need to purchase multiple devices. The number of devices depends on how closely the location services on a phone can pinpoint users. If we can pinpoint locations exactly and only accept chirps within 6 feet, then a conference room of about 5000 square feet would take about 140 devices to fill. Enforcing this resource requirements makes this attack more costly. Also, without clone protection, multiple attacks at different events can be carried out at the same time.

Compared to the constructions in the inverse-Sybil paper [ACK+20], our definition ensures that we do not leak the ordering of our contacts (which both of their protocols do). Comparing to their first protocol, our construction has a lower round-complexity as it does not require an interaction to establish a contact. Their second protocol assumes that a user's location can be measured alongside their assumption that adversarial devices cannot communicate, making our scheme have fewer assumptions than their second protocol (we only assume location is measured).

New construction and primitives In this work, we construct ProvenParrot and prove that it satisfies the security definitions of PACIFIC. Our construction keeps chirping complexity to a single broadcast and uploads are constructed in time linear to the number of chirps heard. This is the same complexity as many existing schemes [CKL+20, ACK+20, ABIV20, TPH+20, GA20] while achieving equal privacy guarantees and stronger integrity guarantees. We will use mercurial signatures [CL18] to sign chirpers' and uploaders' keys to verify them while remaining anonymous. We use VRFs [DY05] in a similar way to

[CHK+06] in order to prevent replay/delay attacks as well as prevent event attacks. We also introduce two new primitives: Set commitments on equivalence classes (CoEC) and zero knowledge subset proofs over commitments (zk-SPoC) in order to keep our scheme efficient.

CoECs have a stronger binding guarantee than traditional commitment schemes. The binding game for CoECs allows the adversary to use two commitments when attempting a double-opening as long as those two commitments are in the same equivalence class. We'll see that this definition is perfect to pair with mercurial signatures as we can reduce a forged signing on a set of attributes to a violation of the binding of the commitment scheme when the equivalence class is the same, or a violation of the unforgeability of the mercurial signature scheme if the equivalence class is not the same. In our scheme, users sign a commitment to the time and location when chirping. An uploader can then prove that their batch does not include commitments that violate clone protection while another user can check the database to see if it contains any of their signatures. A naive way of providing clone protection in uploads is to prove that each pair of notifications is either at the same location or at a different time. This would have a quadratic blowup in complexity. Our zk-SPoC construction allows us to construct a linear complexity proof that a user did not violate clone protection. Both of these new definitions and constructions, set commitments on equivalence classes and zero knowledge subset proofs over commitments are of independent interest.

While we use expensive public key operations and NIZKs, we stress that our construction does not use any interactive protocols at any stage beyond registration (which is a necessary two-round protocol). Crucially, chirping is a simple broadcast. We present a strong definition of contact tracing, requiring that batches be simulated and the scheme could possibly become more efficient by relaxing this definition and using tricks to reduce the number of group elements, computations, and NIZKs. Also, our NIZKs are very efficient, requiring only a handful of black-box proofs of knowledge of discrete logs for any given relation.

Paper roadmap We review preliminaries in Section 2. We then define our scheme, PACIFIC, in Section 3, defining the high-level protocol in Section 3.1 and our definition of integrity and privacy including security games in Section 3.2. When then introduce definitions of our two new primitives: set commitments on equivalence classes in Section 3.3 and zero knowledge subset proofs over commitments in Section 3.4. We then construct "ProvenParrot" in Section 4 which meets our security definitions from Section 3.2. A proof of security for ProvenParrot is given in Appendix D.1. We construct set commitments on equivalence classes in Appendix B.1 and zero knowledge subset proofs over commitments in Appendix B.2.

2 Notation and preliminaries

2.1 Notation

By $(m, *) \in S$ we mean there is a tuple in set S such that the first element of the tuple is m and the second element is another value which could be anything. $\{(m, *) \in S : A\}$ is the set of all tuples in S with m as their first element meeting condition A. Applying a function, f, onto a set of inputs results in a set of outputs. I.e. if $f : X \to Y$, then for $S \subseteq X$, $f(S) = \{f(s) : s \in S\}$. Adding two vectors, $A = \{a_0, a_1, ..., a_{\ell-1}\}, B = \{b_0, b_1, ..., b_{\ell-1}\}$, each of size ℓ, results in $C = A + B = \{a_0 + b_0, a_1 + b_1, ..., a_{\ell-1} + b_{\ell-1}\}$. If a set is given to an adversary, it gives the adversary no information about the ordering of the elements in the set (you can imagine a set is a vector that is shuffled whenever a challenger gives it to an adversary). We use $r \xleftarrow{\$} S$ or $r \leftarrow_\$ S$ to denote a random choice from a set. By PPT $\mathcal{A}$ we mean that $\mathcal{A}$ is a probabilistic polynomial-time adversary.

2.2 Preliminaries

Decisional Diffie-Hellman (DDH) The DDH assumption states that: given a group generated by an element, $\mathbb{G} = \langle P \rangle$, of prime order p, no PPT adversary can distinguish distribution: $\{(P, P^a, P^b, P^{ab}) : a, b \leftarrow_\$ \mathbb{Z}_p\}$ from $\{(P, P^a, P^b, P^c) : a, b, c \leftarrow_\$ \mathbb{Z}_p\}$. The symmetric external DDH (SXDH) states that the DDH holds in $\mathbb{G}_1$ and $\mathbb{G}_2$ for the two groups of a bilinear pairing, and there is no efficient isomorphism between the groups.

Cryptographic bilinear pairings A bilinear pairing [GPS06] is a set of groups, $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, along with a "pairing function," e: $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where $\mathbb{G}_T$ is the "target group" and $\mathbb{G}_1$ and $\mathbb{G}_2$ are the "source groups." In this work, we instantiate type III pairings, which means that there is no efficient, non-trivial homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. The pairing function is efficiently computable and has a bilinearity property such that if $\langle P \rangle = \mathbb{G}_1$ and $\langle \hat{P} \rangle = \mathbb{G}_2$, then $e(P^a, b\hat{P}) = e(P, \hat{P})^{ab}$. We instantiate bilinear pairings using elliptic curves and some pairing similar to the Tate pairing which allows $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = p$ for some prime where all 3 groups are cyclic. In the bilinear pairing we use, the decisional Diffie-Hellman assumption holds in the related groups, such that: $(P^a, P^b, P^{ab}) \sim (P^a, P^b, P^c)$ and $(\hat{P}^a, \hat{P}^b, \hat{P}^{ab}) \sim (\hat{P}^a, \hat{P}^b, \hat{P}^c)$ and given (P^a, P^b) it is difficult to compute P^{ab}. We give a more formal definition of the hardness assumptions of cryptographic bilinear pairings in Appendix A.

Commitments A commitment is a hiding and binding ciphertext that is "committed" to some value. Hiding implies that no adversary can distinguish between commitments to different values and binding means that no adversary can open a commitment to two different values. One common commitment is the Pedersen commitment [Ped92] which uses two generators of a prime order (p) group: $\langle P \rangle = \langle H \rangle = \mathbb{G}$ to create commitment of the form: $C = P^v H^r$

where v is the committed value and $r \leftarrow_{\$} \mathbb{Z}_p$ is the opening. One then opens the commitment by revealing v, r which allows a verifier to then recompute C. Opening C to multiple values allows one to recover the discrete log $\mathsf{dlog}_P(H)$ and the commitments are perfectly hiding since multiplying by a random value in $\mathbb{Z}_p$ is bijective.

2.2.1 Non-interactive zero knowledge proofs of knowledge (NIZKs)

We use NIZKs to prove knowledge of witness that satisfies a verifiable relationship, $\mathcal{R}$. By the definition of NIZKs, a witness can be extracted from any proof in the programmable random oracle model with black-box access to the proof creator (ability to rewind). Further, in the programmable random oracle model, there exists a simulator which can construct a verifying NIZK without knowledge of the witness.

NIZKs can be constructed for general circuits, but this is inefficient. NIZKs are far more efficient for proving relationships of algebraic structures such as the discrete logs between group elements and the equivalence of discrete log between two pairs of group elements..

We will use the notation from [CS97] for proofs $\pi = NIZK$ [known values : condition to prove] where the known values are secret to the prover. This allows a verifier to use π and some public values in the condition to verify that the condition holds and that the creator of the NIZK must know some witness for the relation.

2.2.2 Mercurial Signatures

A mercurial signature scheme (introduced in [CL18]) consists of the following functions (which we prefix with MS).

Part of the functions in mercurial signatures form a standard signature scheme: $\mathsf{MS.PPGen}$ outputs public parameters for the scheme, $\mathsf{MS.KeyGen}(pp_{\mathsf{MS}}, \ell)$: Outputs a public (verification) key and a secret (signing) key: (PK, sk), $\mathsf{MS.Sign}(sk, M)$ outputs a signature (σ) on the message M, $\mathsf{MS.Verify}(PK, M, \sigma)$ verifies a signature given a public key and message.

Where mercurial signatures differ from normal signatures is that public keys, messages, and signatures can be randomized so that they are unlinkable from their original forms while still verifying. This involves the use of "equivalence classes" which are the set of allowed randomizations for each element of the scheme. In the construction for mercurial signatures [CL18], the equivalence classes are defined over tuples of group elements such that the discrete log between them is constant. To define the equivalence classes, we first define a relation for messages ($\mathcal{R}_M$), and keys ($\mathcal{R}_{sk}$,$\mathcal{R}_{PK}$):

$\mathcal{R}_M = \{(M, M') \in (\mathbb{G}_1^*)^\ell \times (\mathbb{G}_1^*)^\ell | \exists r \in \mathbb{Z}_p^* \text{ such that } M = M^r\}$

$\mathcal{R}_{sk} = \{(sk, sk') \in (\mathbb{Z}_p^*)^\ell \times (\mathbb{Z}_p^*)^\ell | \exists r \in \mathbb{Z}_p^* \text{ such that } sk = r * sk\}$

$\mathcal{R}_{PK} = \{(PK, PK') \in (\mathbb{G}_2^*)^\ell \times (\mathbb{G}_2^*)^\ell | \exists r \in \mathbb{Z}_p^* \text{ such that } PK = PK^r\}$

These relations define equivalence classes which are represented as $[M]_{\mathcal{R}_M}$ so

that: $[M]_{\mathcal{R}_M} = [M']_{\mathcal{R}_M}$ if $(M, M') \in \mathcal{R}_M$. For our schemes, we use $\ell = 2$ as this is enough to sign our commitments.

Public keys are randomized with the function: $\mathsf{MS.ConvertPK}(PK, \rho) \rightarrow PK'$, where ρ is a "key converter." The new PK' can be thought of as a *pseudonym* for the original PK. Signatures can be randomized with $\mathsf{MS.ConvertSig}(PK, M, \sigma, \rho) \rightarrow \sigma'$ such that if the same ρ is used, then $\mathsf{Verify}(PK', M, \sigma') = 1$. Messages and signatures can be randomized jointly with $\mathsf{MS.ChangeRep}(PK, M, \sigma, \mu) \rightarrow (M^*, \sigma^*)$ where μ is a blinding factor (also known as a "key/message converter"). The correctness of these functions ensures that, given a public key, message, and signature, then, after calling these functions in the order they were presented (using the converted PK' and σ' as input for $\mathsf{MS.ChangeRep}$) we retrieve a new public key, message, and signature: PK', M^*, σ^* which is unlinkable from the starting public key, message, and signature, yet still verifies. Knowing how these elements are made unlinkable should give the reader good intuition for understanding our construction.

If two mercurial schemes are initialized together, they can be created such that the message space of one is the key space of another [CL18]. This is done by using $\mathbb{G}_2$ as the message space in the second scheme (where $\mathbb{G}_2$ is the key space in the first scheme). Because messages and keys are randomized in the same way, this allows a user to use the first scheme to sign a key from the second scheme and allow that key to be randomized while still verifying and being able to sign more messages. We use this property in our construction. Formal definitions of mercurial signatures are shown in Appendix A.

We also need a special function, $\mathsf{MS.Recognize}(sk, PK)$, which operates on public keys from the mercurial signature scheme, determining if they are in the corresponding equivalence class for a given secret key. This function does not exist in the literature.

2.2.3 Verifiable random function

Let $\mathbb{G}$ be a group of prime order, p, generated by an element, $P = \langle \mathbb{G} \rangle$. Let $f_s^{\mathbb{G},P}(x)$ be the function $P^{1/(s+x)}$. Dodis and Yampolskiy proved that function acts like a pseudo random function under the y-DDHI (y-decisional Diffie-Hellman inversion) assumption [DY05]. Further, given $Y = f_s^{\mathbb{G},P}(x)$, x, P, H, $PK = P^s H^r$, r, s, it can be proven in zero knowledge that PK is a commitment to the s used in the VRF calculation (without revealing s). This x can be hidden inside a commitment as well, which our construction makes use of. The introduction of VRFs saw security with a restriction on the input space. A later work [CHK+06] proved the security of VRF under inputs from $\mathbb{Z}_p^*$ in the GGM.

3 Definitions

3.1 PACIFIC : Privacy-preserving automated contact tracing featuring integrity against cloning

A privacy-preserving automated contact tracing scheme featuring integrity against cloning is described in Definition 1.

Usage In an example deployment of this protocol in the case of a viral outbreak, personal devices like phones or smartwatches could be used to complete an interaction for users in close proximity. For a PACIFIC scheme, to record an interaction, one user would chirp (via the Chirp function) and another would listen (via the Listen function). Interactions are meant to take place repeatedly at a set interval on the order of seconds or minutes. In order to ensure that users cannot violate integrity, users must register their devices to receive a certificate which they use in the chirp function. Some semi-trusted party such as a government agency acts as the registration party. This party creates a registration key pair using RegPartyKeyGen and users trust the public part to verify certificates. Users generate a key pair via UserKeyGen and give the public part to the registration party for registration. Users who test positive for the virus would be verified by a health authority. The infected user would then proceed to compute a "batch" of "notifications" generated by computing Notify on the chirps that they heard within the infection window. They then upload this batch to a public database that other users can check if they were exposed using: VerifyBatch[2] and CountExposures. Users are only meant to upload at most once per "epoch." Uploading notifications in batches like this models a real-world system and allows our construction to be more efficient. [3] This epoch is the length of time that users are infectious before they test positive. This epoch is meant to be much larger than the interval between chirps. An epoch is measured on the order of days or weeks while a chirp interval is measured in seconds or minutes. In our scheme, a database, labeled DB, is simply a set of batches. A database authority receives batches and appends them to the database after running VerifyBatch.

Definition 1 *A privacy-preserving automated contact tracing featuring integrity against cloning scheme consists of a set of algorithms:* ParamGen, RegPartyKeyGen, UserKeyGen, RegisterUser, Chirp, Listen, Notify, VerifyBatch, *and* CountExposures.

- $\mathsf{ParamGen}(1^\lambda, e) \to (pp)$*: The parameter generation function takes as input: a complexity parameter,* 1^λ*, and an epoch length,* e*, and outputs public parameters:* pp *which includes an epoch function,* Epoch*, which is* $t \mapsto \lfloor t/e \rfloor$.

[2] VerifyBatch can be run once by the database owner and since we assume they are semi-honest, integrity still holds.

[3] We address problems where batches are split across an epoch in Appendix E.

- $\mathsf{RegPartyKeyGen}(pp) \to (sk_{rp}, PK_{rp})$*: The registration party key generation function takes the public parameters as input and outputs a public/private key for the registration party that users trust to ensure integrity of the scheme. The registration party will sign users' public keys.*
- $\mathsf{UserKeyGen}(pp, PK_{rp}) \to (sk_{\mathcal{U}}, PK_{\mathcal{U}})$*: The user key generation function generates a user's public/private keys.*
- $\mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}}) \to (cert)$*: Register a user's public key resulting in a certificate.*
- $\mathsf{Chirp}(pp, sk_{\mathcal{U}}, PK_{rp}, cert, t, l) \to (\mathsf{c})$*: The chirp function outputs a chirp using a user's secret key and certificate. This chirp corresponds to the given time and location:* (t, l)*.*
- $\mathsf{Listen}(pp, sk_{\hat{\mathcal{U}}}, PK_{rp}, t, l, \mathsf{c}) \to (0 \text{ or } 1)$*: The listen function verifies that a chirp is fresh (matches the given* t, l *and is not a replay) and valid (signed by the registration party* PK_{rp}*) and then remembers the chirp (and metadata) for a potential upload later.*
- $\mathsf{Notify}(pp, sk_{\mathcal{U}}, C, d) \to (B)$*: The* Notify *The notify function creates a batch of notifications which indicate that other users should quarantine. The function accepts a set of chirps,* C*. These notifications are related to chirps that were heard within the epoch,* d*.*
- $\mathsf{VerifyBatch}(pp, PK_{rp}, DB, B, d) \to (0 \text{ or } 1)$*: This ensures that the batch of notifications includes interactions in a single epoch,* d*, and was uploaded by a registered user that hasn't already uploaded for epoch* d*. This function also ensures that clone protection was not violated.*
- $\mathsf{CountExposures}(pp, sk_{\mathcal{U}}, PK_{rp}, DB, d) \to (\perp \text{ or } \lambda)$*: The count exposures function allows any user to check a database to determine how many contacts they've had with any infected user.*

3.2 PACIFIC security definitions

In this section, we define a number of oracles that make up games that allow us to define correctness, integrity, and privacy. These oracles share some global state initialized by SetupGame. The adversary's interaction with the oracles can be thought of as an interaction with a challenger, but this oracle-style definition helps us define the games, with some games sharing oracle definitions. We then define a number of games for integrity, chirp privacy (where the registration party is malicious), and upload privacy (where the registration party is semi-honest).

Summary of integrity game. In the integrity game (Game 6), we allow the adversary to create honest and corrupted users, send their own chirps to honest users, listen to chirps from honest users, have honest users upload batches,

and upload malicious batches. After the adversary exits, we are left with a resulting global state that records all the chirps sent and batches uploaded. We can then use this global state to create a set of possible interactions (Equation 8) that we expect to be in the database due to correctness. We then ensure that the interactions extracted from the database are a subset of those possible interactions (in Equation 11). Each extracted interaction is a tuple consisting of the identities of two users, a time, and location. We then require that users must count no more exposures from the database than what the extracted set indicates (Equation 10). And we ensure that no tuple in the extracted set constitutes a clone protection violation (Equation 12). Using the extractors in this way ensures that *the truth (that defines what the users draw their counted exposures from) maps to some subset of the possible interactions that doesn't violate clone protection.* Thus, there exists an adversary that could produce this exact set of counted exposures simply by acting as a number of honest users equal to the number of registrations the adversary makes. This ensures that no user can do more damage than simply purchasing devices or bribing others and using those devices like an honest user would. This is a strong guarantee and gives us upload replay/delay protection as well as clone protection as both of these require users to act differently than an honest users. An honest user never replays/delays uploads and never violates clone protection. In our definition, we'll extract a set from the database: the "extracted interactions," labeled EI. This can be thought of as a map, with Equation 1 ensuring that it is well defined on the multiset of counted notifications and Equations 2 and 3 ensuring that the interactions during the game explain the map.

To show an attack that our definition prevents, let's say that an adversary obtains two devices and reverse engineers their keys, learning sk_1 and sk_2. They now place beacons in various areas with both of these keys. Now, let's say that at the start of a week (time $t = 0$) two honest users (u_1 and u_2) are at two different locations. We'll label these as locations 1 and 2 ($l = 1$, $l = 2$). After the adversary uploads to the database, we find that both users are exposed twice. If this scenario is valid, we should be able to extract a map explaining the adversary's movements, but we'll see that we cannot. This map can be thought of as a set of tuples (one for each notification) each of which containing a corrupted identity, an honest user, and a time and location. So we need to extract four valid tuples. We can see that the only way to extract these four tuples across the adversary's keys is: $\{(sk_1, u_1, t = 0, l = 1), (sk_2, u_1, t = 0, l = 1), (sk_1, u_2, t = 0, l = 2), (sk_2, u_2, t = 0, l = 2)\}$ since we need these tuples to be valid with the reality of the game (that u_1 was at location $l = 1$ and u_2 was at location $l = 2$). We can see that these tuples violate clone protection as we have a tuple indicating that the corrupted user sk_1 was at locations $l = 1$ at $t = 0$ and another tuple indicating they were at location $l = 2$ at the same time. Thus, no valid map can be extracted, and if an adversary can create these exposures with only sk_1 and sk_2, they must have violated clone protection.

Summary of privacy games. Our chirp privacy game (where honest users only chirp) is private even in the presence of a malicious registration party, while our upload privacy game (where honest users can be directed to upload batches) only defines security in the presence of a semi-honest registration server with fully malicious devices.

In our chirp privacy game, we allow the chirp simulator to know the time and location, but critically not the identity of the honest user. We allow the time and location to leak since this chirp is broadcasted locally, so any nearby adversary would know this information. Our upload privacy game builds on the chirp privacy game, by allowing for honest users to upload. We generate the simulated batches with random times and locations (as well as random user keys), thus enforcing that batches are unlinkable to any given time and location.

The adversary can gain a lot of information by computing $\mathsf{CountExposures}$ on honest batches from honest users who interacted with different corrupted users at different locations. This is because of correctness: the batch must reveal the number of exposures corresponding to each of the corrupted users. But, critically, *the amount of information that the adversary gets through the simulated honest upload oracle scales with the number of corrupted users that the adversary creates.* To simulate a batch, we extract an identity from each chirp heard by the honest user. We then ensure they map to registrations in the game. If a chirp is from an honest user, we create a new chirp with a fresh identity and attributes. If the chirp is from a corrupted registration, we create a chirp on random attributes that are only linkable to the extracted corrupted identity. Thus, the adversary only learns the number of contacts that each of their corrupted users had with each uploader as well as the number of honest interactions that uploader had and nothing more.

We provide the following informal definition and defer the formal definition to Appendix C.

Below, we describe the set of oracles that can be called by the adversary in our games. We will use different sets of oracles for different games. Each oracle shares global state and is run by a challenger. We define values in the global state alongside the oracles in this informal definition.

- $\mathsf{RegisterHonest}(1^\lambda) \to (i, PK_{\mathcal{U}})$: **Generate and register an honest user.** Generate a user key pair $(sk_{\mathcal{U}}, PK_{\mathcal{U}})$ using $\mathsf{UserKeyGen}(pp, PK_{rp})$. Generate a certificate for this user: $cert \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}})$. Return the handle for this honest user, i and the public key, $PK_{\mathcal{U}}$.

- $\mathsf{RegisterCorrupt}(PK_{\mathcal{U}}) \to (cert)$: **Register a corrupted user.** Return a certificate on $PK_{\mathcal{U}}$: $cert \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}})$.

- $\mathsf{RecvChirp}(i, l) \to (\mathsf{c})$: **Receive a chirp from an honest user.** Takes a user handle, i (received by the adversary from $\mathsf{RegisterHonest}$), and a location, l. If this user already chirped at this time, abort. Otherwise, compute the chirp: $\mathsf{c} \leftarrow \mathsf{Chirp}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, \mathsf{Hcert}(i), t_{\mathsf{now}}, l)$ where t_{now} is the current time (initially set to 0), $\mathsf{HU}_{sk}(i)$ is honest user i's secret key, and $\mathsf{Hcert}(i)$ is the honset user i's certificate. Return c.

- $\mathsf{SendChirp}(i, l, \mathsf{c}) \to (\perp)$: **Send a chirp to an honest user.** If this user already listened to a chirp at a different location at this same time, abort. Compute: $\mathsf{Listen}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, t_{\mathsf{now}}, l, \mathsf{c})$ and update the user's set of heard chirps.
- $\mathsf{HonestInteraction}(i, j, l) \to (\perp)$: **Have two honest users interact.** Run $\mathsf{c} \leftarrow \mathsf{RecvChirp}(i, l)$ and $\mathsf{SendChirp}(j, l, \mathsf{c})$.
- $\mathsf{IncrementTime}(1^\lambda) \to (\perp)$: **Increment the current time.** Set $t_{\mathsf{now}} = t_{\mathsf{now}} + 1$. If, after the increment, we enter a new epoch, reset the database by forgetting all previous batches.
- $\mathsf{HonestUpload}(i) \to (B)$: **Have an honest user upload to the server.** Return a batch computed by the honest user: $B = \mathsf{Notify}(pp, \mathsf{HU}_{sk}(i), C)$ where C is the set of all chirps they heard and verified during the game.
- $\mathsf{CorruptedUpload}(B) \to (\perp)$: **A corrupted user uploads to the database.** Run $\mathsf{VerifyBatch}(pp, PK_{rp}, DB, B, \mathsf{Epoch}(t_{\mathsf{now}}))$ where DB is the current set of batches. If this outputs 1, add B to the database, DB.

Game 1 (Correctness game) *Run* $\mathcal{A}^{O_{\mathsf{corr}}}(pp, PK_{rp}, 1^k)$, *where* $O_{\mathsf{corr}} = \{\mathsf{RegisterHonest}, \mathsf{HonestInteraction}, \mathsf{IncrementTime}, \mathsf{HonestUpload}\}$. *After* $\mathcal{A}$ *exits, ensure that the counted exposures matches the interactions from the game that were uploaded. This equation is shown below where* $\mathsf{HU}_{sk}(i)$ *is the secret key for the honest user with handle* i *and* $\mathsf{HI}_{sk,sk}$ *is a set of interactions where the first entry is the secret key of the chirper and the second entry is the secret key of the listener.*

$$\begin{aligned}&\forall i \in \mathsf{HU},\\ &\quad \mathsf{CountExposures}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, DB, t_{\mathsf{now}})\\ &\quad = \#\{(\mathsf{HU}_{sk}(i), \mathsf{HU}_{sk}(j)) \in \mathsf{HI}_{sk,sk} : \textit{honest user } j \textit{ uploaded}\}\end{aligned}$$

Where HU *is the set of honest users. If this check fails, output* 1 *indicating that the adversary won, otherwise, output 0.*

Definition 2 (Correctness) *An automated exposure notification scheme,* Π, *is correct if no PPT adversary can win Game 1 with probability greater than negligible for all valid epoch values.*

3.2.1 Clone integrity

Game 2 (Clone integrity game) *Run* $(sk_{rp}, PK_{rp}) \leftarrow \mathsf{RegPartyKeyGen}(1^\lambda)$. *Run* $\mathcal{A}^{O_{\mathsf{integrity}}}(pp, PK_{rp}, 1^\lambda)$, *where* $O_{\mathsf{integrity}} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{HonestInteraction}, \mathsf{IncrementTime}, \mathsf{HonestUpload}, \mathsf{CorruptedUpload}\}$. *When* $\mathcal{A}$ *exits, the challenger uses the resulting global state to determine if the adversary won.*

- *The challenger computes the set of* ***possible interactions*** *(*PI*) which contains each chirp emitted by honest users paired with each corrupted secret key registered as well as any interactions between honest users. Thus, the size of this set is:* (*number of honestly emitted chirps*) $\times$ (*number of corrupted users*) + (*number of honest interactions*)*. The tuples in this set include an honest chirper secret key, a listener secret key, a time, and location (*$\mathsf{PI} \subset \mathcal{SK} \times \mathcal{SK} \times \mathcal{T} \times \mathcal{L}$*).*
- *The challenger computes the set of* ***extracted interactions*** *(*EI*) from the database which would notify an honest user using the extractor* $\mathcal{E}_{DB}$*. These tuples are of the same form as* PI*.*

We are now ready to check conditions and output 1 *if any fail, indicating that the adversary wins. Otherwise, output* 0*.*

1. ***Correct exposure count.*** *Ensure the extracted interactions match the exposures counted by honest users:*
   $\forall i \in \mathsf{HU}$,
   $$\mathsf{CountExposures}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, DB, t_{\mathsf{now}}) \quad (1)$$
   $$= \#\{(sk_{\mathcal{U}}, sk_{\mathcal{U}}{}', t, l) \in \mathsf{EI}_{sk} : sk_{\mathcal{U}} = \mathsf{HU}_{sk}(i)\}$$
   Where HU *is the set of honest users and* $\mathsf{HU}_{sk}(i)$ *is the secret key of the honest user,* i*.*

2. ***Database contains a subset of possible interactions.*** *Ensure these extractions are within the set of chirps sent by honest users during the game:*
   $$\mathsf{EI} \subseteq \mathsf{PI} \quad (2)$$

3. ***Clone protection.*** *Ensure that no corrupted user was in two locations at the same time:*
   $$\nexists \left((*, sk_{\mathcal{U}}, t, l), (*, sk_{\mathcal{U}}{}', t', l')\right) \in \mathsf{EI} \quad s.t.\ sk_{\mathcal{U}} = sk_{\mathcal{U}}{}' \wedge t = t' \wedge l \neq l' \quad (3)$$

Definition 3 (Clone integrity) *An automated exposure notification scheme,* Π*, has clone integrity if there exists a set of extractors,* $\mathcal{E}$*, such that no PPT adversary can win Game 2 with probability greater than negligible for all valid epoch values.*

3.2.2 Privacy definitions

To define chirp privacy, we create a simulator for chirps in the ideal game:

- $\mathsf{RecvChirp}^{sim}(\perp, l) \rightarrow (\mathsf{c})$: Functions exactly like the real version, $\mathsf{RecvChirp}$, except this simulated version computes the chirp with a new user. I.e.: to compute the chirp, first generate a new keypair: $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{UserKeyGen}(PK_{rp})$. Then register this user: $cert \leftarrow \mathsf{RegisterUser}(sk_{rp}, PK_{\mathcal{U}})$. Compute a chirp for this user on the given time and location, $\mathsf{c} \leftarrow \mathsf{Chirp}(sk_{\mathcal{U}}, PK_{rp}, cert, t_{\mathsf{now}}, l)$. Output this chirp, c.

In the chirp privacy game, we allow the adversary to create a corrupted registration party. The registration party secret key is extracted to be used in the $\mathsf{RecvChirp}^{sim}$ oracle. We define a function that allows the adversary to create their own certificates for honest users.

- $\mathsf{RegisterHonest}^{mal}(1^\lambda) \rightarrow (i, PK_\mathcal{U})$: Generate a key pair, $sk_\mathcal{U}, PK_\mathcal{U} \leftarrow \mathsf{UserKeyGen}(pp, PK_{rp})$. Call the adversary with $PK_\mathcal{U}$ to receive *cert*. Return the handle for this honest user, i and the public key, $PK_\mathcal{U}$.

Game 3 (Chirp privacy game) *Run* $(PK_{rp}, st) \leftarrow \mathcal{A}(pp, 1^\lambda)$ *and use the* PK_{rp} *for oracles in the game. Extract* $sk_{rp} = \mathcal{E}_{PK_{rp}}(PK_{rp})$ *for registering new users in the simulator. Flip a random bit,* $b \leftarrow_\$ \{0, 1\}$*. If* $b = 0$*, run* $b' \leftarrow \mathcal{A}^{O_{real}}(pp, 1^\lambda, st)$*, otherwise, if* $b = 1$*, run* $b' \leftarrow \mathcal{A}^{O_{sim}}(pp, 1^\lambda, st)$*, where:* $O_{real} = \{\mathsf{RegisterHonest}^{mal}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{IncrementTime}\}$ *and* $O_{sim} = \{\mathsf{RegisterHonest}^{mal}, \mathsf{RecvChirp}^{sim}, \mathsf{SendChirp}, \mathsf{IncrementTime}\}$*. The adversary wins if* $b = b'$*.*

Definition 4 (Chirp privacy) *An automated exposure notification scheme,* Π*, is privacy-preserving with respect to chirps if there exists a set of extractors,* $\mathcal{E}$*, such that no PPT adversary has greater than* $\frac{1}{2} + negl(\lambda)$ *advantage in Game 3 for all valid epoch values.*

To define upload privacy, we need another simulator for batches[4]

- $\mathsf{HonestUpload}^{sim}(i) \rightarrow (B)$: First, compute a new uploader keypair, $sk_{\hat{\mathcal{U}}}, PK_{\hat{\mathcal{U}}} \leftarrow \mathsf{UserKeyGen}(pp)$ and register this user: $cert_{\hat{\mathcal{U}}} \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\hat{\mathcal{U}}})$. Next, reconstruct a simulated set of chirps, C, to be used by the Notify function. For each tuple in the chirps received by user i during the game, extract the secret key of the sender from the chirp, labeling the multiset of these extracted secret keys: RC^i. Then count the number of honest senders in this set: $K_{\mathsf{HU}} = \#\{sk \in \mathsf{RC}^i : sk \in \mathsf{HU}_{sk}$. Next, we need to extract all secret keys of corrupted users with $\mathcal{E}_{sk}(PK_\mathcal{U})$ using the public keys provided during registration. For each adversarial user, count the number of interactions with that user: $K_{sk\mathcal{A}} = \#\{sk \in \mathsf{RC}^i : sk = sk\mathcal{A}\}$. Then compute a total of K_{HU} new chirps for new random users, times and locations, using $sk_\mathcal{U}, PK_\mathcal{U} \leftarrow \mathsf{UserKeyGen}(pp, PK_{rp})$, $cert \leftarrow \mathsf{RegisterUser}(sk_{rp}, PK_\mathcal{U})$. Then, for each adversarial secret key, compute $K_{sk_\mathcal{U}}$ new chirps for the extracted adversarial secret key, with random new times and locations, doing the same as was done for honest chirps, but skipping the key generation and using the corrupted key instead. Label the set of all these new simulated chirps from honest and corrupted users as C. Now compute a batch using the new uploader and simulated chirps: $B \leftarrow \mathsf{Notify}(sk_{\hat{\mathcal{U}}}, PK_{rp}, C, t_{\mathsf{now}})$. Output this batch, B.

We are now ready to define the game:

[4]Note that while a user handle is given to $\mathsf{HonestUpload}^{sim}$, it is only used to compute leakage such as batch size.

Game 4 (Upload-privacy game) *Run* $(sk_{rp}, PK_{rp}) \leftarrow \mathsf{RegPartyKeyGen}(1^\lambda)$. *Flip a random bit,* $b \xleftarrow{\$} \{0,1\}$. *If* $b = 0$, *run* $st \leftarrow \mathcal{A}^{O_{real}}(pp, PK_{rp}, 1^\lambda)$, *otherwise, if* $b = 1$, *run* $st \leftarrow \mathcal{A}^{O_{sim}}(pp, PK_{rp}, 1^\lambda)$, *where:* $O_{real} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{IncrementTime}, \mathsf{HonestUpload}\}$ *and* $O_{sim} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}^{sim}, \mathsf{SendChirp}, \mathsf{IncrementTime}, \mathsf{HonestUpload}^{sim}\}$. *Next, run* $b' \leftarrow \mathcal{A}(sk_{rp}, st)$. *The adversary wins if* $b = b'$.

Definition 5 (Upload privacy) *An automated contact tracing scheme,* Π, *is privacy-preserving if there exists a set of extractors,* $\mathcal{E}$, *such that no PPT adversary has greater than* $\frac{1}{2} + negl(\lambda)$ *advantage in Game 4 for all valid epoch values.*

3.3 Set Commitments on equivalence classes (CoECs)

In this Section, we describe CoECs, commitments that are binding across equivalence classes. Traditional commitments ensure that an adversary cannot produce a commitment with openings to distinct messages. Commitments on equivalence classes assume a slightly weaker adversary in that the adversary is allowed to output two commitments with openings to distinct sets of messages as long as the commitments are in the same equivalence class. This allows CoECs to compose well with mercurial signatures, which ensure the equivalence class of messages are unforgeable. We also include a property: "class-hiding" which ensures that even if an adversary creates the commitment, it is still indistinguishable after it has been randomized. This function takes a group description, $\mathbb{G}$, of prime size p and outputs commitments in an equivalence class:

$$\mathcal{R}_C = \{(C, C') \in (\mathbb{G}_1^*)^\ell \times (\mathbb{G}_1^*)^\ell | \exists r \in \mathbb{Z}_p^* \text{ s.t. } C' = C^r\}$$

Here we define a CoEC scheme for $\ell = 2$ in Definition 6. Later, in Appendix B.1, we construct a CoEC scheme in Definition 27.

Definition 6 (CoECs scheme)

- $\mathsf{InitializeCoEC}(1^k, s, \mathbb{G}) \rightarrow pp$*: Initialize the commitment scheme for a number of attributes,* s, *outputting public parameters,* pp.
- $\mathsf{Commit}(pp, M = \{m_0, m_1, ...\}) \rightarrow (C, O)$*: Commit to a set of attributes,* M, *where* $|M| = s$. *Output the commitment,* C, *and opening information* O. $|C| = 2$ *and possible outputs of this function are defined on an equivalence class:* $[C]_{\mathcal{R}_C} = \{C' : \mathsf{dlog}_{C_0'}(C_1') = \mathsf{dlog}_{C_0}(C_1)\}$.
- $\mathsf{RandomizeCom}(pp, C, O) \rightarrow (C', O')$*: Randomizes the commitment so that* C *and* C' *are unlinkable but still a commitment to the same set.*
- $\mathsf{Open}(pp, C, M, O) \rightarrow (0 \text{ or } 1)$*: Take opening information and output whether* C *is a commitment to* M.

Definition 7 (CoEC correctness) *A commitment scheme is correct if* $\forall$ s, k, *and attributes* M *where* $|M| = s$, *then given* $pp \leftarrow \mathsf{InitializeCoEC}(1^k, s)$, *the following holds:*
$\Pr[(C, O) \leftarrow \mathsf{Commit}(pp, M); \mathsf{Open}(pp, C, M, O) = 1] = 1$
And:
$\forall C, M, O$ *such that* $\mathsf{Open}(pp, C, M, O) = 1] = 1$,
$\Pr[(pp, M); (C', O') = \mathsf{Randomize}(C, O); \mathsf{Open}(pp, C', M, O') = 1] = 1$

Definition 8 (CoEC binding) *A commitment scheme is binding if for all* s, k, *and for all PPT adversary,* $\mathcal{A}$, *then given* $pp \leftarrow \mathsf{InitializeCoEC}(1^k, s)$, *the following holds: then the following probabilities are negligible:*
$\Pr[(C, C', M, M', O, O') \leftarrow \mathcal{A}(pp)$
$\mathsf{Open}(pp, C, M, O) = 1$
$\wedge\, \mathsf{Open}(pp, C', M', O') = 1 \wedge M \neq M'$
$\wedge\, [C]_{\mathcal{R}_C} = [C']_{\mathcal{R}_C}] \leq negl(k)$

Definition 9 (CoEC hiding) *A commitment scheme is hiding if for all* s, k, *and for all PPT adversary,* $\mathcal{A}$, *then given* $pp \leftarrow \mathsf{InitializeCoEC}(1^k, s)$, *the following holds:*

$$\Pr[b \xleftarrow{\$} \{0,1\}; (M_0, M_1, st) \leftarrow \mathcal{A}(pp); (C, O) \leftarrow \mathsf{Commit}(pp, M_b); b^* \leftarrow \mathcal{A}(st, C) : b = b^*] \leq \frac{1}{2} + negl(k) \quad (4)$$

Definition 10 (CoEC class-hiding) *A commitment scheme is class-hiding if for all* s, k, *PPT adversary,* $\mathcal{A}$, $pp \leftarrow \mathsf{InitializeCoEC}(1^k, s, \mathbb{G})$, $C, O, M, C', O', M' \leftarrow \mathcal{A}(pp, 1^k)$ *(where* $\mathsf{Open}(pp, C, M, O) = \mathsf{Open}(pp, C', M', O') = 1$*),* $C_0 \leftarrow \mathsf{Randomize}(C, O)$ *is indistinguishable from* $C_1 \leftarrow \mathsf{Randomize}(C', O')$.

We construct CoECs in Appendix B.1 and prove our construction secure in Appendix D.2.

Theorem 1 *The CoEC scheme in Definition 27 is correct, hiding, origin-hiding, and binding as defined in Definitions 7, 8, 10, and 9 as long as the SXDH assumption (from Section 2) holds and mercurial signatures are message class-hiding (in Appendix A).*

3.4 Zero knowledge subset proofs over commitments (zk-SPoCs)

Overview As a wordy summary: zk-SPoCs prove that the set of sets of attributes committed to by one set of commitments is a subset of another set of sets of attributes committed to by another set of commitments. Let's clarify with notation: let's say you have two sets of commitments, each of size m

and n: $A = \{a_0, a_1, ..., a_m\}$ and $B = \{b_0, b_1, ..., b_n\}$. Each of these commitments are committed to a set of attributes. We'll label the set of all these sets of attributes as: $\mathcal{M} = \{m_0, m_1, ..., m_m\}$ and $\mathcal{L} = \{l_0, l_1, ..., l_n\}$ i.e. each a_i opens to the set m_i and each b_i opens to the set l_i. A zk-SPoC proves that all openings are known and that $\mathcal{M} \subseteq \mathcal{L}$ without revealing $\mathcal{M}$ or $\mathcal{L}$. Note that we do not consider duplicates in this subset i.e. if there are two commitments to the same set in A ($\exists m_i = m_j, i \neq j$) but B only has one copy of this set ($\exists l_k = m_i, m_i \neq l_q \forall q \neq k$), we still say $\mathcal{M} \subseteq \mathcal{L}$. In other words, we consider $\mathcal{M}$ and $\mathcal{L}$ sets, not vectors or multisets. This is easily accomplished in n^2 time using a ZKP OR proof. We create definitions so that we can construct proofs in time linear to n while retaining security properties. The high level protocol has an initialization function, a prove function and a verify function:

- $\mathsf{InitZKSPoC}(1^k)$: Initialize the commitment scheme, $pp_{\mathsf{SC}} \leftarrow \mathsf{Setup}(1^k)$. Output $pp = pp_{\mathsf{SC}}$ along with a description of the given commitment scheme.
- $\mathsf{ProveSubset}(pp, \{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]}) \rightarrow (W)$: Takes in two sets of commitments and outputs a proof that $\{L_i\}_{i\in[m]} \subseteq \{M_i\}_{i\in[n]}$ and the commitments open to these sets.
- $\mathsf{VerifSubset}(pp, \{A_i\}_{i\in[m]}, \{B_i\}_{i\in[n]}, W) \rightarrow (1 \text{ or } 0)$: Takes in the witness and outputs 1 if A is a commitment to a set, $\{L_i\}_{i\in[m]}$, which is a subset of $\{M_i\}_{i\in[n]}$, which are the message sets that $\{B_i\}_{i\in[n]}$ are commitments to.

These subset functions have a correctness property and two security properties:

Definition 11 (Correctness)
For all sets of commitments and openings, $\{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]}$, such that $\mathsf{Open}(A_i, L_i, O_i) = \mathsf{Open}(B_i, M_i, P_i) = 1$ and $\{L_i\}_{i\in[m]} \subseteq \{M_i\}_{i\in[n]}$, then given $W \leftarrow \mathsf{ProveSubset}(pp, \{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]})$, the following holds: $\mathsf{VerifSubset}(pp, \{A_i\}_{i\in[m]}, \{B_i\}_{i\in[n]}, W) = 1$

Definition 12 (Soundness)
For all PPT adversaries, $\mathcal{A}$:
$\Pr[(\{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]}, W) \leftarrow \mathcal{A}(pp, 1^k);$
$\mathsf{VerifSubset}(\{A_i\}_{i\in[m]}, \{B_i\}_{i\in[n]}, W) = 1$
$\wedge \forall i \in [m], \mathsf{Open}(A_i, O_i, L_i) = 1$
$\wedge \forall i \in [n], \mathsf{Open}(B_i, P_i, M_i) = 1$
$\wedge \{L_i\}_{i\in[m]} \not\subseteq \{M_i\}_{i\in[n]}] \leq negl(k)$

Definition 13 (Zero knowledge)
A zk-SPoC scheme is zero knowledge if there exists a simulator ($\mathcal{S}^{\mathsf{ProveSubset}}$) that, when given only the commitments (no messages or openings), can output a witness that is indistinguishable from a witness computed with valid information of the sets and openings. Formally, a zk-SPoC scheme is zero knowledge if the

probability below holds for all PPT $\mathcal{A}$ and all $pp \leftarrow \mathsf{InitZKSPoC}(1^k)$*:*
$\Pr[(st, \{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]}) \leftarrow \mathcal{A}(pp);$
$W^0 \leftarrow \mathsf{ProveSubset}(pp, \{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]});$
$W^1 \leftarrow \mathcal{S}^{\mathsf{ProveSubset}}(\{A_i\}_{i\in[m]}, \{B_i\}_{i\in[n]});$
$b \leftarrow \{0, 1\};$
$b' \leftarrow \mathcal{A}(pp, st, W^b);$
$\mathsf{Open}(A_i, O_i, L_i) = \mathsf{Open}(B_i, P_i, M_i) = 1 \wedge b = b'] \leq \frac{1}{2} + negl(k)$

We construct CoECs in Appendix B.2 and prove our construction secure in Appendix D.3.

Theorem 2 *The zk-SPoC scheme described in Definition 28 meets Definition 11.*

Theorem 3 *The zk-SPoC scheme described in Definition 28 meets Definition 13 with the simulator defined in Definition 34 given a subset-sound FHS19 commitment scheme described in Appendix .*

Theorem 4 *The zk-SPoC scheme described in Definition 28 meets Definition 12 given a binding FHS19 commitment scheme described in Appendix .*

4 Construction of ProvenParrot

4.1 Description

Anonymous registration, chirps, and uploads. In order to register users without opening ourselves up to the Brutus attack, i.e. linking registrations to uploads (described in Section 1.2), we will use mercurial signatures. We reviewed mercurial signatures in Section 2. Mercurial signatures allow users to randomize their pseudonym after registration so that, while users can verify that the new randomized pseudonym is signed, they cannot link it to any other pseudonym used during registration, chirping, or uploading. Mercurial signatures further allow for signing messages which then verify under pseudonyms (where the pseudonyms can be signed by a trusted public key). We will use this property to allow users to sign commitments to the current time and location for chirping preventing replay attacks. A clever reader will see that this could potentially cause problems during upload as a user could potentially rerandomize their pseudonym many times in order to pretend to be many users, thus negating any rate-limiting we attempted. We solve this problem using *verifiable random functions* (VRFs) [DY05, CHK$^+$06]. This allows us to compute a pseudorandom function (PRF) of the current time and a function of the user's secret key which remains constant across all pseudonyms. Thus, if a user only uploads at most once per epoch (as honest users do) the output will look random. But when a malicious user attempts to upload twice in the same time period, the

VRF will output the same value, thus exposing their misbehavior. We also use VRFs to ensure that users do not chirp more than once per chirp interval. This is shown in blue text and stops an obscure attack on our scheme described in Appendix E but can be removed if this attack is deemed impractical.

Clone protection. To solve the event attack described in Section 1.2, we need to utilize time and location. The idea is that devices will use the current time as well as their location when creating chirps. Listeners can then compute the time and location themselves and verify that the chirp was indeed meant for the given time and their location. This immediately prevents chirp relay/replay attacks as a relayed chirp would have an incorrect location and a replayed chirp would have an incorrect time, allowing the listener to discard the chirp. This is done in [CKL+20]. The key improvement that we add is using this time and location data when these chirps are uploaded to the server. During upload, we have the server check if an upload suggests that they were in two different locations at the same time (thus preventing cloned devices). To make this information private while still allowing verification, we form *randomizable commitments* to the time and location data. An uploader then proves to the server that the time and locations of the commitments does not violate clone protection. We introduce two new definitions and constructions to make this possible and efficient: *set commitments on equivalence classes* (CoECs) and *zero knowledge subset proofs over commitments* (zk-SPoCs). These are defined in Sections 3.3 and 3.4 and constructed in Appendices B.1 and B.2.

Our construction proceeds as follows: users generate a mercurial key pair and have the public portion signed by the registration party. To create a chirp, the user randomizes their public key and signature from the registration party, and signs a commitment to the current time and location. They then broadcast this chirp (including the opening of the commitment) to nearby users. Listeners can then verify that the commitment is valid (for the current time and location) and then verify that it is signed by the chirper and that the chirper's public key is signed by the registration party. Later, during upload, a user takes these heard chirps and randomizes all elements: the heard public key, the signature, and the commitment. These are uploaded to the public database all at once in a batch. Then, any other users can query the database and recognize their public key (and verify their signature) to determine that they had were exposed. Because these chirps were randomized (and the commitments are class-hiding), users who check the database cannot link any exposure to a specific chirp, time, or location.

Uploaders must prove that the times and locations committed to by the chirps do not violate clone protection, i.e., no two commitments have the same time but different locations. A trivial way to do this is to run a zero-knowledge proof across each pair of commitments. But this requires quadratic complexity. We can do better using zero knowledge subset proofs over commitments. Instead, an uploader will create a shuffled copy of the set of commitments across all heard chirps and remove any duplicates. This copy is now a superset of

the commitments in the heard chirps as it contains all attribute sets from the chirps. They upload this copy as well as the commitments from the heard chirps along with a proof of subset (zk-SPoC). They then compute a VRF on each *time value* committed to by this shuffled set. This effectively proves that the batch uploaded does not violate clone protection as a commitment to the same time at two locations would mean a second commitment to this time would remain in the superset (since t, l and t, l' are distinct values) and thus the computed VRF would match. An honest user can remove all duplicates of any t, l, thus making all their VRF computations distinct and random.

Chirpers also sign a nonce and uploaders compute a VRF on this value to ensure that they do not rerandomize the same chirp multiple times. We compute this nonce by hashing the randomized public key to prevent a de-anonymization attack described in Appendix E.

4.2 ProvenParrot

We name our construction: "ProvenParrot" as it is similar to CerfifiedCleverParrot from [CKL+20] but requires the uploader to prove properties of their upload (that their notifications do not violate clone protection).

We use mercurial signatures as summarized in Section 2 and defined in Appendix A We define a function, $f^{\mathsf{MS}}_{(\cdot)}(\cdot)$ as a VRF as described in 2, but this function will take in a mercurial secret key and compute the PRF using $\mathsf{dlog}_{sk_0}(sk_1)$, thus ensuring that, given any equivalence class representation of a key, it computes the PRF with the same key no matter how the key has been randomized. We show a more complete zero knowledge construction of VRFs on mercurial signatures in Appendix B.3. We implicitly pass the corresponding $PK_{\mathcal{U}}$ to any function where $sk_{\mathcal{U}}$ is used.

Definition 14 (ProvenParrot construction)

- $\mathsf{ParamGen}(1^k, e) \rightarrow (pp)$*: Generate two mercurial signature schemes such that the message space of one is the public key space of the other. This means generating the schemes so that the first scheme generates public keys in* $\mathbb{G}_1$ *and the second scheme generates them in* $\mathbb{G}_2$ *(the message space of the first scheme). The second scheme then signs messages in* $\mathbb{G}_1$*:* $pp_{\mathsf{MS},1} \leftarrow \mathsf{MS.PPGen}_1(1^k), pp_{\mathsf{MS},2} \leftarrow \mathsf{MS.PPGen}_2(1^k)$*. We generate the CoEC and zk-SPoC schemes so that they operate on the message space for the second mercurial signature scheme,* $\mathbb{G}_1$*: Initialize a commitment scheme for four attributes:* $pp_{\mathsf{Com}} \leftarrow \mathsf{InitializeCoEC}^{\mathbb{G}_1}(1^k, 4)$*. Initialize a zk-SPoC scheme* $pp_{zk-SPoC} \leftarrow \mathsf{InitZKSPoC}^{\mathbb{G}_1}(1^k)$*. Create an epoch function* Epoch *which divides and floors any given time:* $\mathsf{Epoch} : t \mapsto \lfloor \frac{t}{e} \rfloor$*. Output the public parameters:* $pp = \{\mathsf{Epoch}, pp_{\mathsf{MS},1}, pp_{\mathsf{MS},2}, pp_{\mathsf{Com}}, pp_{zk-SPoC}\}$*.*
- $\mathsf{RegPartyKeyGen}(pp) \rightarrow (sk_{rp}, PK_{rp})$*: Generate a mercurial keypair:* $sk_{rp}, PK_{rp} \leftarrow \mathsf{MS.KeyGen}_1(pp_{\mathsf{MS},1})$*, for the registration party. Construct a NIZK proving that* sk_{rp} *is known:* $\pi_{rp} =$

NIZK $\left[sk_{rp}, r : PK_{rp} \leftarrow \mathsf{MS.KeyGen}_1(pp_{\mathsf{MS},1}; r)\right]$. *Output* (sk_{rp}, PK_{rp}) *where* π_{rp} *is implicitly part of the public key.*

- $\mathsf{UserKeyGen}(pp, PK_{rp}) \rightarrow (sk_{\mathcal{U}}, PK_{\mathcal{U}})$*: First check if* π_{rp} *is correct for* PK_{rp}*, aborting and outputting* $\perp$ *if not. Run* $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{MS.KeyGen}_2(pp_{\mathsf{MS},2})$ *and output both the public and private key along with a proof simliar to* RegPartyKeyGen.

- $\mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}}) \rightarrow (cert)$*: Verify proof of* $PK_{\mathcal{U}}$*. Sign* $PK_{\mathcal{U}}$ *with* sk_{rp}*:* $\sigma \leftarrow \mathsf{MS.Sign}(sk_{rp}, PK_{\mathcal{U}})$ *and return* σ *as the certificate,* $cert$*.*

- $\mathsf{Chirp}(pp, sk_{\mathcal{U}}, PK_{rp}, cert, t, l) \rightarrow (\mathsf{c})$*:*
  - *Verify that our certificate is valid (that* $PK_{\mathcal{U}}$ *verifies under* PK_{rp} *with* $cert = \sigma$*). If not, output* $\perp$*.*
  - *Derive an epoch from* t*:* $d \leftarrow \mathsf{Epoch}(t)$*.*
  - *Randomize the user's public key and certificate:* $\rho \leftarrow_{\$} \mathbb{Z}_p^*$; $PK_{\mathcal{U}}', cert' \leftarrow \mathsf{MS.ChangeRep}(PK_{rp}, PK_{\mathcal{U}}, cert, \rho)$
  - *Generate a random nonce from the randomized public key:* $r = H(PK_{\mathcal{U}}')$ *where* H *is a hash function:* $H : (\mathbb{G}_2)^2 \rightarrow \mathbb{Z}_p$*.*
  - *Generate a commitment:*

$$C, O = \mathsf{Commit}(pp_{\mathsf{Com}}, (t, l, d, r))$$

  - *Sign* C *with a randomized secret key and certificate using the blinding factor from the randomization of the public key,* ρ*:* $sk_{\mathcal{U}}' \leftarrow \mathsf{MS.ConvertSK}(sk_{\mathcal{U}}, \rho)$ $\sigma_{\mathsf{c}} \leftarrow \mathsf{MS.Sign}(sk_{\mathcal{U}}', C)$
  - *Compute a VRF on the secret key and the time and prove that this is correctly computed:*
    $\pi^t = NIZK[sk_{\mathcal{U}}' : Y^t = f^{\mathsf{MS}}_{sk_{\mathcal{U}}'}(t)$
    $\wedge\ PK_{\mathcal{U}}' = sk_{\mathcal{U}}' P]$
  - *Broadcast* $\mathsf{c} = (C,\ O,\ \sigma_{\mathsf{c}},\ PK_{\mathcal{U}}',\ cert',\ \pi^t,\ Y^t\ )$*.*

- $\mathsf{Listen}(pp, PK_{rp}, t, l, \mathsf{c}) \rightarrow (1 \text{ or } 0)$*: Use the* t *given as input and recompute* $d = \mathsf{Epoch}(t)$*. Recompute the nonce:* $r = H(PK_{\mathcal{U}}')$*. Use the location given as input,* l*, along with the opening,* O*, from the chirp to ensure that* $\mathsf{Open}(pp_{\mathsf{Com}}, C, (t, l, d, r), O) \stackrel{?}{=} 1$*. Verify that* $1 \stackrel{?}{=} \mathsf{MS.Verify}(C, PK_{\mathcal{U}}, \sigma_{\mathsf{c}})$ *and* $1 \stackrel{?}{=} \mathsf{MS.Verify}(C, PK_{rp}, cert)$*. Verify the NIZK with the given public key and current time and check the listener's state to ensure that* Y^t *is new. If all these checks pass, output* 1*. Otherwise, output* 0*. Remember the chirp along with the time and location for uploading later.*

- $\mathsf{Notify}(pp, sk_{\hat{\mathcal{U}}}, \{\mathsf{c}_i\}_{i \in [n]}, d_{\mathsf{now}}) \rightarrow (B)$*: (Note: we've labeled the key pair for the executor of this function (the uploader) as:* $sk_{\hat{\mathcal{U}}}, PK_{\hat{\mathcal{U}}}$*, to distinguish it from the public keys in the chirps).*

– *Compute a randomized copy of the uploader's keys and certificate:*
$PK'_{\hat{\mathcal{U}}}, cert' \leftarrow \mathsf{MS.ChangeRep}(pp_{\mathsf{MS},2}, PK_{\hat{\mathcal{U}}}, cert, \hat{\rho})$
$sk'_{\hat{\mathcal{U}}} \leftarrow \mathsf{MS.ConvertSK}(pp_{\mathsf{MS},2}, sk_{\hat{\mathcal{U}}}, \hat{\rho})$
where $\hat{\rho} \leftarrow_{\$} \mathbb{Z}_p^*$ *is a blinding factor.*

– *Compute and prove a PRF on* $sk'_{\hat{\mathcal{U}}}$ *and* d_{now}*:*
$\pi^{PK} = NIZK[sk'_{\hat{\mathcal{U}}} : Y^{PK} = f^{\mathsf{MS}}_{sk'_{\hat{\mathcal{U}}}}(d_{\mathsf{now}}) \wedge PK'_{\hat{\mathcal{U}}} = sk'_{\hat{\mathcal{U}}}P]$

– *Randomize all n chirps heard. This means randomizing public keys, signatures, certificates, and commitments:*
$\forall i \in [n]$,

$\rho_i \leftarrow_{\$} \mathbb{Z}_p^*, \mu_i \leftarrow_{\$} \mathbb{Z}_p^*$
$PK_{\mathcal{U}i}' \leftarrow \mathsf{MS.ConvertPK}(PK_{\mathcal{U}i}, \rho_i)$
$\sigma^*_{\mathsf{c},i} \leftarrow \mathsf{MS.ConvertSig}(PK_{\mathcal{U}i}, C_i, \sigma_{\mathsf{c},i}, \rho_i)$
$C'_i, \sigma'_{\mathsf{c},i} \leftarrow \mathsf{MS.ChangeRep}(PK_{\mathcal{U}i}, C_i, \sigma^*_{\mathsf{c},i}, \mu_i)$
$O'_i = O_i * \rho_i$
$cert'_i = cert_i$

– *Prove that every commitment has* $d = d_{\mathsf{now}}$ *each resulting in a proof,* π_i^d*:*
$\forall i \in [n]$,

$\pi_i^d = NIZK[O'_i, t_i, l_i, r_i :$
$\mathsf{Open}(C'_i, O'_i, t_i, l_i, d_{\mathsf{now}}, r_i) = 1]$

– *For each of the nonces in the chirp commitments,* r_i*, compute* Y_i^r *and prove that is is the output of a PRF on a normalized* $sk'_{\hat{\mathcal{U}}}$ *and* r_i*:*
$\forall i \in [n]$,

$\pi_i^r = NIZK[O'_i, sk_{\hat{\mathcal{U}}}, t_i, l_i, r_i : \mathsf{Open}(C'_i, O'_i, \{t_i, l_i, d_i, r_i\}) = 1$
$\wedge Y_i^r = f^{\mathsf{MS}}_{sk'_{\hat{\mathcal{U}}}}(r_i), sk'_{\hat{\mathcal{U}}} = PK'_{\hat{\mathcal{U}}}P]$

– *To prove clone protection, create commitments to* t, l *for each* C'_i*, resulting in a set* $\{C_i^{tl}\}_{i\in[n]}$*. Prove equivalency for this new set of commitments. Create a second set of commitments,* $\{C_i^*\}_{i\in[n]}$*, such that:* $\{C_i^*\}_{i\in[n]} \approx \{C_i^{tl}\}_{i\in[n]}$ *but do not prove equivalency. Iterate through each duplicate commitment (two commitments to the same* t, l*) in* $\{C_i^*\}_{i\in[n]}$ *and replace each with a new commitment to random attributes,* $t, l \leftarrow_{\$} \mathbb{Z}_p^2$*. After this is done,* $\{C_i^*\}_{i\in[n]}$ *should include a commitment to each* t, l *from* $\{C_i^{tl}\}_{i\in[n]}$*, but not have any duplicates. Compute a proof,* π^{tl}*, that* $\{C_i^*\}$ *is committed to a superset of the values committed to by* $\{C_i^{tl}\}_{i\in[n]}$ *using the* ProveSubset *function from Section 3.4. Compute a VRF of each* t *committed to by* $\{C_i^*\}_{i\in[n]}$ *combined with the uploader's secret key,* $sk_{\hat{\mathcal{U}}}$*:*
$\pi_i^t = NIZK[O'_i, sk'_{\hat{\mathcal{U}}}, \{t_i^*, l_i^*\} : \mathsf{Open}(C_i^*, O_i^*, \{t_i^*, l_i^*\}) = 1$
$\wedge Y_i^t = f^{\mathsf{MS}}_{sk'_{\hat{\mathcal{U}}}}(t_i^*), sk'_{\hat{\mathcal{U}}}P = PK'_{\hat{\mathcal{U}}}]$

– *Shuffle and output each chirp with randomized public keys and signatures*

along with their proofs. This is the batch which is outputted by this function, B:
$B = \big(PK'_{\hat{\mathcal{U}}}, cert', Y^{PK}, \pi^{PK}, \pi^{tl},$
$\{PK_{\mathcal{U}}{}'_i, \sigma'_{c,i}, C'_i, C_i^{tl}, \pi_i^d, \pi_i^r, Y_i^r\}_{i\in[n]},$
$\{C_i^*, \pi_i^t, Y_i^t\}_{i\in[n]}\big)$

- $\mathsf{VerifyBatch}(pp, PK_{rp}, DB, B) \rightarrow (1 \text{ or } 0)$*: Verify that the certificate in the batch is a valid signature for $PK'_{\hat{\mathcal{U}}}$ under PK_{rp} using $1 \stackrel{?}{=} \mathsf{MS.Verify}(PK_{rp}, PK'_{\hat{\mathcal{U}}}, cert')$. Verify all NIZK proofs. Ensure there's no duplicates in $\{Y_i^r\}_{i\in[n]}$ or $\{Y_i^t\}_{i\in[n]}$ across the batch. Ensure there's no duplicate Y^{PK} across the database.*

- $\mathsf{CountExposures}(pp, sk_{\mathcal{U}}, PK_{rp}, DB) \rightarrow k$*: Iterate through each pair of public key, $PK'_{\mathcal{U},i}$, and signature, $\sigma'_{c,i}$, in each batch and run $\mathsf{MS.Recognize}(pp_{\mathsf{MS}}, sk_{\mathcal{U}}, PK'_{\mathcal{U},i})$. If this passes, ensure that $\mathsf{MS.Verify}(PK'_{\mathcal{U},i}, C'_i, \sigma'_{c,i})$ is true. If both are true, this is a valid exposure. Sum the number of valid exposures across the database and output the resulting number.*

4.3 Proofs of security

We introduce the theorem here that we then prove in Appendix D.1 after introducing more formal versions of our security definitions.

Theorem 5 *The ProvenParrot scheme described in Definition 14, using the set of extractors: $\mathcal{E}$ from Definition 33, has clone integrity (meeting Definition 3) and is chirp-private (meeting Definition 4) and is upload-private (meeting Definition 5).*

References

[ABIV20] Gennaro Avitabile, Vincenzo Botta, Vincenzo Iovino, and Ivan Visconti. Towards defeating mass surveillance and sars-cov-2: The pronto-c2 fully decentralized automatic contact tracing system. Cryptology ePrint Archive, Paper 2020/493, 2020. `https://eprint.iacr.org/2020/493`.

[ACHdM05] Giuseppe Ateniese, Jan Camenisch, Susan Hohenberger, and Breno de Medeiros. Practical group signatures without random oracles. Cryptology ePrint Archive, Paper 2005/385, 2005. `https://eprint.iacr.org/2005/385`.

[ACK+20] Benedikt Auerbach, Suvradip Chakraborty, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and

Michelle Yeo. Inverse-sybil attacks in automated contact tracing. Cryptology ePrint Archive, Paper 2020/670, 2020. https://eprint.iacr.org/2020/670.

[AFV20] Gennaro Avitabile, Daniele Friolo, and Ivan Visconti. Terrorist attacks for fake exposure notifications in contact tracing systems. Cryptology ePrint Archive, Paper 2020/1150, 2020. https://eprint.iacr.org/2020/1150.

[BCK+20] Jean-François Biasse, Sriram Chellappan, Sherzod Kariev, Noyem Khan, Lynette Menezes, Efe Seyitoglu, Charurut Somboonwit, and Attila Yavuz. Trace-σ: a privacy-preserving contact tracing app. Cryptology ePrint Archive, Paper 2020/792, 2020. https://eprint.iacr.org/2020/792.

[BDH+20] Wasilij Beskorovajnov, Felix Dörre, Gunnar Hartung, Alexander Koch, Jörn Müller-Quade, and Thorsten Strufe. Contra corona: Contact tracing against the coronavirus by bridging the centralized–decentralized divide for stronger privacy. Cryptology ePrint Archive, Report 2020/505, 2020. https://ia.cr/2020/505.

[BRS20] Samuel Brack, Leonie Reichert, and Björn Scheuermann. Caudht: Decentralized contact tracing using a dht and blind signatures. Cryptology ePrint Archive, Paper 2020/398, 2020. https://eprint.iacr.org/2020/398.

[CBB+20a] Claude Castelluccia, Nataliia Bielova, Antoine Boutet, Mathieu Cunche, Cédric Lauradoux, Daniel Le Métayer, and Vincent Roca. DESIRE: A third way for a european exposure notification system leveraging the best of centralized and decentralized systems. *CoRR*, abs/2008.01621, 2020.

[CBB+20b] Claude Castelluccia, Nataliia Bielova, Antoine Boutet, Mathieu Cunche, Cédric Lauradoux, Daniel Le Métayer, and Vincent Roca. Robert: Robust and privacy-preserving proximity tracing. 2020.

[CFG+20] Justin Chan, Dean P. Foster, Shyam Gollakota, Eric Horvitz, Joseph Jaeger, Sham M. Kakade, Tadayoshi Kohno, John Langford, Jonathan Larson, Sudheesh Singanamalla, Jacob E. Sunshine, and Stefano Tessaro. PACT: privacy sensitive protocols and mechanisms for mobile contact tracing. *CoRR*, abs/2004.03544, 2020.

[CHK+06] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 201–210, New York, NY, USA, 2006. Association for Computing Machinery.

[CKL+20] Ran Canetti, Yael Tauman Kalai, Anna Lysyanskaya, Ronald L. Rivest, Adi Shamir, Emily Shen, Ari Trachtenberg, Mayank Varia, and Daniel J. Weitzner. Privacy-preserving automated exposure notification. 2020. https://ia.cr/2020/863.

[CL18] Elizabeth C. Crites and Anna Lysyanskaya. Delegatable anonymous credentials from mercurial signatures. Cryptology ePrint Archive, Report 2018/923, 2018. https://ia.cr/2018/923.

[CM99] Jan Camenisch and Markus Michels. Separability and efficiency for generic group signature schemes. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 1999.

[COPZ22] Melissa Chase, Michele Orrù, Trevor Perrin, and Greg Zaverucha. Proofs of discrete logarithm equality across groups. Cryptology ePrint Archive, Paper 2022/1593, 2022. https://eprint.iacr.org/2022/1593.

[Cos12] Craig Costello. Pairings for beginners. Online, 2012. https://www.craigcostello.com.au/s/PairingsForBeginners.pdf.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 410–424, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Dam10] Ivan Damgard. On σ protocols, 2010. https://cs.au.dk/~ivan/Sigma.pdf.

[Daw20] Ellie Daw. Component-based comparison of privacy-first exposure notification protocols. Cryptology ePrint Archive, Paper 2020/586, 2020. https://eprint.iacr.org/2020/586.

[DDL+20] Noel Danz, Oliver Derwisch, Anja Lehmann, Wenzel Puenter, Marvin Stolle, and Joshua Ziemann. Provable security analysis of decentralized cryptographic contact tracing. Cryptology ePrint Archive, Paper 2020/1309, 2020. https://eprint.iacr.org/2020/1309.

[DSM+22] Sumit Kumar Debnath, Vikas Srivastava, Tapaswini Mohanty, Nibedita Kundu, and Kouichi Sakurai. Quantum secure privacy preserving technique to obtain the intersection of two datasets for contact tracing. *Journal of Information Security and Applications*, 66:103127, 2022.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 416–431, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[FHS14] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. Structure-preserving signatures on equivalence classes and constant-size anonymous credentials. Cryptology ePrint Archive, Paper 2014/944, 2014. `https://eprint.iacr.org/2014/944`.

[GA20] Google and Apple. Privacy-preserving contact tracing, 2020. `https://covid19.apple.com/contacttracing`.

[GhP+20] Giuseppe Garofalo, Tim Van hamme, Davy Preuveneers, Wouter Joosen, Aysajan Abidin, and Mustafa A. Mustafa. Pivot: Private and effective contact tracing. Cryptology ePrint Archive, Paper 2020/559, 2020. `https://eprint.iacr.org/2020/559`.

[GPS06] S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers. Cryptology ePrint Archive, Paper 2006/165, 2006. `https://eprint.iacr.org/2006/165`.

[HL10] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Information Security and Cryptography. Springer Berlin Heidelberg, 2010.

[JBQ20] Alvin Tan Chai Sheng Hau Lai Yongquan Janice Tan Jason Bay, Joel Kek and Tang Anh Quy. Bluetrace: A privacy-preserving protocol for community-driven contact tracing across borders, 2020. `https://eprint.iacr.org/2020/55://bluetrace.io/static/bluetrace_whitepaper-938063656596c104632def383eb33b3c.pdf`.

[NMD+22] Thien Duc Nguyen, Markus Miettinen, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Ivan Visconti. Digital contact tracing solutions: Promises, pitfalls and challenges. *IEEE Transactions on Emerging Topics in Computing*, pages 1–12, 2022.

[Org22] World Health Organization. Who coronavirus (covid-19) dashboard, 2022. `https://covid19.who.int/`.

[PAJ20] Deepraj Pandey, Nandini Agrawal, and Mahabir Prasad Jhanwar. Covidbloc: A blockchain powered exposure database for contact tracing. Cryptology ePrint Archive, Paper 2020/1543, 2020. `https://eprint.iacr.org/2020/1543`.

[Ped92] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[PR21] Benny Pinkas and Eyal Ronen. Hashomer – privacy-preserving bluetooth based contact tracing scheme for hamagen. In *Real World Crypto and NDSS Corona-Def Workshop*, 2021.

[RAC+20] Ronald L. Rivest, Hal Abelson, Jon Callas, Ran Canetti, Kevin Esvelt, Daniel Kahn Gillmor, Louise Ivers, Yael Tauman Kalai, Anna Lysyanskaya, Adam Norige, Bobby Pelletier, Ramesh Raskar, Adi Shamir, Emily Shen, Israel Soibelman, Michael Specter, Vanessa Teague, Ari Trachtenberg, Mayank Varia, Marc Viera, Daniel Weitzner, John Wilkinson, and Marc Zissman. The pact protocol specification, 2020. `https://pact.mit.edu/wp-content/uploads/2020/11/The-PACT-protocol-specification-2020.pdf`.

[RBS20] Leonie Reichert, Samuel Brack, and Björn Scheuermann. Privacy-preserving contact tracing of covid-19 patients. Cryptology ePrint Archive, Paper 2020/375, 2020. `https://eprint.iacr.org/2020/375`.

[RBS21] Leonie Reichert, Samuel Brack, and Björn Scheuermann. Ovid: Message-based automatic contact tracing. 01 2021.

[RG20] James Krellenstein Rosario Gennaro, Adam Krellenstein. Exposure notification system may allow for large-scale voter suppression. 2020. `https://static1.squarespace.com/static/5e937afbfd7a75746167b39c/t/5f47a87e58d3de0db3da91b2/1598531714869/Exposure_Notification.pdf`.

[Tan20] Qiang Tang. Privacy-preserving contact tracing: current solutions and open questions. Cryptology ePrint Archive, Paper 2020/426, 2020. `https://eprint.iacr.org/2020/426`.

[TPH+20] Carmela Troncoso, Mathias Payer, Jean-Pierre Hubaux, Marcel Salathé, James R. Larus, Edouard Bugnion, Wouter Lueks, Theresa Stadler, Apostolos Pyrgelis, Daniele Antonioli, Ludovic Barman, Sylvain Chatel, Kenneth G. Paterson, Srdjan Capkun, David A. Basin, Jan Beutel, Dennis Jackson, Marc Roeschlin, Patrick Leu, Bart Preneel, Nigel P. Smart, Aysajan Abidin, Seda F. Gürses, Michael Veale, Cas Cremers, Michael Backes, Nils Ole Tippenhauer, Reuben Binns, Ciro Cattuto, Alain Barrat, Dario Fiore, Manuel Barbosa, Rui Oliveira, and José Pereira. Decentralized privacy-preserving proximity tracing. *CoRR*, abs/2005.12273, 2020.

[Tra20] Tracetogether, 2020. `https://www.tracetogether.gov.sg/`.

[TSS+20] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy, 2020.

[Vau20a] Serge Vaudenay. Analysis of dp3t. Cryptology ePrint Archive, Paper 2020/399, 2020. https://eprint.iacr.org/2020/399.

[Vau20b] Serge Vaudenay. Centralized or decentralized? the contact tracing dilemma. Cryptology ePrint Archive, Paper 2020/531, 2020. https://eprint.iacr.org/2020/531.

[WL20] Zhiguo Wan and Xiaotong Liu. Contactchaser: A simple yet effective contact tracing scheme with strong privacy. Cryptology ePrint Archive, Paper 2020/630, 2020. https://eprint.iacr.org/2020/630.

A Additional preliminaries

Note that we switch to using the additive notation for group operations in this appendix. Thus, an "exponentiation" of $P \in \mathbb{G}_1$ by $a \in \mathbb{Z}_p$ is denoted as: aP and a "discrete log" between two group elements, P and P' is denoted: $\frac{P'}{P}$.

A.1 Verifiable random functions (VRFs)

A VRF [DY05] allows a user to compute and prove the output of a pseudorandom function using their secret key with some public input where the verifier only sees a (potentially randomized) commitment to the public key. A VRF is composed of three functions (and an initialization function):

- $\mathsf{Initialize}(1^k)$: Outputs the public parameters we implicitly use in the other functions.
- $\mathsf{VRF}_{sk}(x) \to Y$: Outputs a deterministic pseudorandom string computed with key sk on input x. We also write this as $f_{sk}(x)$ when it is unambiguous.
- $\mathsf{Prove}(sk, PK, o, Y, x) \to \pi$: Outputs a proof that Y is correct for the given randomized public key, PK and x.
- $\mathsf{Verify}(PK, Y, x, \pi) \to \{0 \text{ or } 1\}$: Verifies a proof that Y is correct for the given randomized public key, PK and x.

Definition 15 (Correctnesss)
If $\mathsf{VRF}_{sk}(x) = Y$ *and* $\mathsf{Prove}(sk, PK, o, Y, x) = \pi$*, where* o *is a valid opening of* sk *on commitment* PK*, then* $\mathsf{Verify}(PK, Y, x, \pi) = 1$.

Definition 16 (VRF Uniqueness)
A verifiable random function is unique if no adversary can compute $(PK, x, Y_1, Y_2, \pi_1, \pi_2)$ *such that* $\mathsf{Verify}(PK, x, Y_1, \pi_1) = \mathsf{Verify}(PK, x, Y_2, \pi_2) = 1$ *and* $Y_1 \neq Y_2$.

Definition 17 (VRF Pseudorandomness)
A verifiable random function is pseudorandom if for all PPT advesaries, the following probaiblity is negligible:
$\Pr[b = b' :$

$(PK, sk, o) \leftarrow \mathsf{Commit}(1^k);$

$(x, st) \leftarrow \mathcal{A}^{\mathsf{VRF}(sk,\cdot),\mathsf{Prove}(sk,PK,o,\cdot,\cdot)}(PK);$

$Y_0 = \mathsf{VRF}_{sk}(x); Y_1 \leftarrow \{0,1\}^k;$

$b \leftarrow \{0,1\};$

$b' \leftarrow \mathcal{A}^{\mathsf{VRF.Prove}(\cdot)}(Y_b, st)] \leq \frac{1}{2} + negl(k)$

VRF injectivity Let the group $\mathbb{G}$ be an elliptic curve group of prime order. Let $\mathbb{G}$ be generated by an elliptic curve point such that $\mathbb{G} = \langle P \rangle$. Let this group be a R-module with the ring, $\mathbb{Z}_p$, where $p = |\langle P \rangle|$ and p is prime. These are the types of groups used by the Weil and Tate pairings, yielding efficient Type-III bilinear pairings [Cos12]. To prove that Dodis-Yampolskiy VRFs [DY05] are injective, we assume the opposite and come to a contradiction: If this VRF weren't injective, this would imply that: $1/(s+x)P = 1/(s+x')P$ where $x \neq x'$. This implies that: $1/(s+x) = 1/(s+x')$, since assuming otherwise would imply that $aP = a'P$ (where $a = 1/(s+x)$ and $a' = 1/(s+x')$). WLOG, let $a' < a < p$. This implies that $(a-a')P = \mathcal{O}$, but $(a-a') < p$ and we know the order of P is p, so we have a contradiction. We can see that $\mathbb{Z}_p^*$ is a group on multiplication and $\mathbb{Z}_p$ is a group on addition. Groups have unique inverses and Z_p^* has the cancellation property. This allows us to make the following implications:
$\big(1/(s+x) = 1/(s+x')\big) \implies \big((s+x) = (s+x')\big) \implies \big(x = x'\big)(5)$ Which leads to a contradiction.

A.2 Mercurial signatures

A mercurial signature scheme is comprised of the functions: MS.PPGen, MS.KeyGen, MS.Sign, MS.Verify, MS.ConvertSK, MS.ConvertPK, MS.ConvertSig, and MS.ChangeRep. A mercurial signature scheme is parameterized by a length, ℓ, which determines how large of messages can be signed.

- $\mathsf{MS.PPGen}(1^k) \rightarrow pp$: Outputs public parameters, pp, including parameterized equivalence relations for the message, public key, and secret key space: $\mathcal{R}_M$, $\mathcal{R}_{pk}$, $\mathcal{R}_{sk}$ and the sample space for key and message converters.
- $\mathsf{MS.KeyGen}(pp, \ell) \rightarrow (pk, sk)$: Generates a key pair of length ℓ.
- $\mathsf{MS.Sign}(sk, M) \rightarrow \sigma$: Signs a message, M with the given secret key.
- $\mathsf{MS.Verify}(pk, M, \sigma) \rightarrow$ (0 or 1): Given a signature, σ, verify that M was signed by pk.

- $\mathsf{MS.ConvertPK}(pk, \rho) \rightarrow pk'$: Given a key converter, ρ, randomize a public key so that is it unlinkable to any other public key (including pk).
- $\mathsf{MS.ConvertSK}(sk, \rho) \rightarrow sk'$: Randomize a secret key such that it now corresponds to a public key which has been randomized with the same ρ (i.e. signatures by $sk' = \mathsf{MS.ConvertSK}(sk, \rho)$ verify by the randomized $pk' = \mathsf{MS.ConvertPK}(pk, \rho)$).
- $\mathsf{MS.ConvertSig}(pk, M, \sigma, \rho) \rightarrow \sigma'$: Randomize the signature so that it verifies with a randomized pk' (which has been randomized with the same ρ) and M, but σ' is unlinkable to any other signature (including σ).
- $\mathsf{MS.ChangeRep}(pk, M, \sigma, \mu) \rightarrow (M', \sigma')$: Randomize the message and signature so that they are unlinkable, but still verify with each other.

Mercurial signatures define relations which for messages and keys as:

Definition 18 (Mercurial signatures equivalence classes)
$\mathcal{R}_M = \{(M, M') \in (\mathbb{G}_1^*)^\ell \times (\mathbb{G}_1^*)^\ell | \exists r \in \mathbb{Z}_p^* \ s.t. \ M' = rM\}$
$\mathcal{R}_{pk} = \{(pk, pk') \in (\mathbb{G}_2^*)^\ell \times (\mathbb{G}_2^*)^\ell | \exists r \in \mathbb{Z}_p^* \ s.t. \ pk' = rpk\}$
$\mathcal{R}_{sk} = \{(sk, sk') \in (\mathbb{Z}_p^*)^\ell \times (\mathbb{Z}_p^*)^\ell | \exists r \in \mathbb{Z}_p^* \ s.t. \ sk' = rsk\}$

These relations define equivalence classes which are represented as $[M]_{\mathcal{R}_M}$ so that: $[M]_{\mathcal{R}_M} = [M']_{\mathcal{R}_M}$ if $(M, M') \in \mathcal{R}_M$.

Definition 19 (Mercurial Signatures Unforgeability) *A mercurial signature scheme (PPGen, KeyGen, Sign, Verify, ConvertSK, ConvertPK, ConvertSig, ChangeRep) for parameterized equivalence relations $\mathcal{R}_\mathcal{M}$, $\mathcal{R}_{pk}$, and $\mathcal{R}_{sk}$ is unforgeable if for all polynomial- length parameters (k) and all probabilistic, polynomial-time (PPT) algorithms A having access to a signing oracle, there exists a negligible function ν such that:*
$\Pr[\mathsf{PP} \leftarrow \mathsf{PPGen}(1k); (pk, sk) \leftarrow \mathsf{KeyGen}(PP, (k));$
$(Q, pk^*, M^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk, \cdot)}(pk) :$
$\forall M \in Q, [M^*]_{\mathcal{R}_M} \neq [M]_{\mathcal{R}_M} \wedge [pk^*]_{\mathcal{R}_{pk}} = [pk]_{\mathcal{R}_{pk}}$
$\wedge \mathsf{Verify}(pk^*, M^*, \sigma^*) = 1] \leq \nu(k)$

Definition 20 (Message class-hiding) *For all polynomial-length parameters $\ell(k)$ and all PPT adversaries A, there exists a negligible ν such that:*

$$\Pr[pp \leftarrow PPGen(1k); M_1 \leftarrow \mathcal{M}; M_2^0 \leftarrow \mathcal{M}; M_2^1 \leftarrow [M_1]_{\mathcal{R}_M}; \\ b \leftarrow \{0,1\}; b' \leftarrow \mathcal{A}(pp, M_1, M_2^b) : b'] = b \leq \frac{1}{2} + \nu(k) \quad (6)$$

Definition 21 (Public key class-hiding) *For all polynomial-length parameters $\ell(k)$ and all PPT adversaries A, there exists a negligible ν such that:*

$$\begin{aligned}\Pr[pp \leftarrow PPGen(1k); (PK_1, sk_1) \leftarrow KeyGen(pp, \ell(k));\\ (PK_2^0, sk_2^0) \leftarrow KeyGen(pp, \ell(k)); \rho \leftarrow sample_\rho(pp);\\ PK_2^1 = ConvertPK(PK_1, \rho); sk_2^1 = ConvertSK(sk_1, \rho); b \leftarrow \{0,1\};\\ b' \leftarrow \mathcal{A}^{Sign(sk_1,\cdot),Sign(sk_2^b,\cdot)}(pp, PK_1, PK_2^b) : b'] = b \leq \frac{1}{2} + \nu(k) \quad (7)\end{aligned}$$

Definition 2 (Correctness) A mercurial signature scheme (PPGen, KeyGen, Sign, Verify, ConvertSK, ConvertPK, ConvertSig, ChangeRep) for parameterized equivalence relations $\mathcal{R}_M, \mathcal{R}_{\text{pk}}, \mathcal{R}_{\text{sk}}$ is correct if it satisfies the following conditions for all k, for all $pp_{\mathsf{MS}} \in \mathsf{MS.PPGen}\left(1^k\right)$, for all $\ell > 1$, for all (pk, sk) $\in$ KeyGen (pp_{MS}, ℓ) :

Verification For all $M \in \mathcal{M}$, for all $\sigma \in$ Sign(sk, M), Verify(pk, $M, \sigma) = 1$.

Key conversion For all $\rho \in$ sample $_\rho$, (ConvertPK (pk, ρ), ConvertSK (sk, ρ)) $\in$ KeyGen(PP, ℓ). Moreover, ConvertSK (sk,ρ) $\in$ [sk]$_{\mathcal{R}_{\text{sk}}}$ and ConvertPK (pk,ρ) $\in$ [pk]$_{\mathcal{R}_{\text{pk}}}$

Signature conversion For all $M \in \mathcal{M}$, for all σ such that Verify (pk, M, σ) = 1, for all $\rho \in$ sample $_\rho$, for all $\tilde{\sigma} \in$ ConvertSig(pk, M, σ, ρ), Verify (ConvertPK (pk, ρ), $M, \tilde{\sigma}) = 1$

We have to define an extra version of correctness for Mercurial signatures which was implicitly used in the original paper:

Definition 22 (Cross-scheme correctness) *Let* $pp_1 \leftarrow \mathsf{MS.PPGen}_{\mathbb{G}_1}$ *and* $pp_2 \leftarrow \mathsf{MS.PPGen}_{\mathbb{G}_2}$. *Let* $sk_1, PK_1 \leftarrow \mathsf{MS.KeyGen}(pp_1)$ *and* $sk_2, PK_2 \leftarrow \mathsf{MS.KeyGen}(pp_2)$. *Let* $\sigma_1 \leftarrow \mathsf{MS.Sign}(pp_2, sk_2, PK_1)$. *Let* $\mu \leftarrow \mathbb{Z}_p^*$.
Let $PK_1'\sigma_1' \leftarrow \mathsf{MS.ChangeRep}(pp_2, PK_1, \sigma_1, \mu)$. *Let* $sk_1' \leftarrow \mathsf{MS.ConvertSK}(pp_1, PK_1, \mu)$. *Let* M *me any message in* $\mathbb{G}_2$. *Let* $\sigma_M \leftarrow \mathsf{MS.Sign}(pp_1, sk_1', M)$. *Ensure that* $\mathsf{MS.Verify}(pp_2, PK_2, PK_1', \sigma_1) = 1 \wedge \mathsf{Verify}(pp_1, PK_1', M, \sigma_M) = 1$.

Cryptographic bilinear pairings A bilinear pairing is a set of groups along with a "pairing function," such that: $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where $\mathbb{G}_T$ is the "target group." In this work, we instantiate type III pairings, which means that there is no efficient, non-trivial homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. The pairing function is efficiently computable and has a bilinear property such that if $\langle P \rangle = \mathbb{G}_1$ and $\langle \hat{P} \rangle = \mathbb{G}_2$, then $e(aP, b\hat{P}) = e(P, \hat{P})^{ab}$. We instantiate bilinear pairings using elliptic curves and some pairing similar to the tate pairing which allows $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = p$ for some prime where all 3 groups are cyclic. In the bilinear pairing we use, the decision Diffie-Hellman assumption holds in the related groups, such that: $(aP, bP, abP) \sim (aP, bP, cP)$ and $(a\hat{P}, b\hat{P}, ab\hat{P}) \sim (a\hat{P}, b\hat{P}, c\hat{P})$.

The Symmetric eXternal Diffie-Hellman assumption (SXDH) [ACHdM05] The decisional DH problem holds in $\mathbb{G}_1$ and $\mathbb{G}_2$ for the cryptographic bilinear pairing based on elliptic curves described above.

A.3 FHS19 Commitments

A set-commitment scheme from [FHS14] has the functions: $\mathsf{Setup}(1^k, 1^s)$, $\mathsf{Commit}(pp, S)$, $\mathsf{Open}(pp, C, S, O)$, $\mathsf{OpenSubset}(pp, C, S, O, T)$, $\mathsf{VerifySubset}(pp, C, T, W)$ where S is a set of attributes (of max size s), C is a commitment, O is an opening, T is a subset of attributes, and W is a witness.

- $\mathsf{Setup}(1^k, 1^s) \rightarrow pp$: Generate a bilinear pairing, $\mathsf{BG} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P, \hat{P}, e) \leftarrow \mathsf{BGGen}(1^k)$ where the groups are of size p. Pick $\alpha \leftarrow \mathbb{Z}_p$ and output $(\mathsf{BG}, (P^{\alpha^i}, \hat{P}^{\alpha^i})_{i\in[s]})$ as the public parameters and α as the trapdoor.
- $\mathsf{Commit}(pp, S) \rightarrow C$: On a set S of messages, pick a random opening, $\rho \leftarrow \mathbb{Z}_p^*$, compute $C \leftarrow P^{f_S(\alpha)\rho}$ using the public parameters where $f_S(\alpha)$ is $\prod_{i\in[s]}(\alpha - S_i)$. Output commitment C and opening information $O = \rho$.
- $\mathsf{Open}(pp, C, S, O) \rightarrow$ (1 or 0): Recompute the commitment using $C' \leftarrow \mathsf{Commit}(pp, \mathsf{S})$ with $\rho = O$ and check if $C = C'$.
- $\mathsf{OpenSubset}(C, S, O, T) \rightarrow W$: Ensure that T is a subset of S and compute the witness: $W = P^{f_{S\setminus T}(\alpha)}$ using the public parameters similar to Commit.
- $\mathsf{VerifySubset}(C, T, W) \rightarrow$ (1 or 0): Ensure that $e(W, \hat{P}^{f_T(\alpha)}) = e(C, \hat{P})$.

We make use of the structure of the $\mathsf{VerifySubset}$ in our zk-SPoC construction in Section B.2. The set-commitment scheme has the following properties:

Definition 23 (Correctness)
A FHS19 set-commitment scheme is correct if for all S,T where $T \subset S$, the following holds:

$$\Pr[(C, O) \leftarrow \mathsf{Commit}(pp, S) : \mathsf{Open}(pp, C, S, O) = 1] = 1$$

$$\Pr[(C, O) \leftarrow \mathsf{Commit}(pp, S); (W) \leftarrow \mathsf{OpenSubset}(pp, C, S, O, T) : \mathsf{VerifySubset}(pp, C, T, W) = 1] = 1$$

Definition 24 (Binding)
A FHS19 set-commitment scheme is binding if for all polynomial adversaries, $\mathcal{A}$, the following holds:

$$\Pr[(C, S, O, S', O') \leftarrow \mathsf{Adv}(pp) : \mathsf{Open}(pp, C, S, O) = 1 \wedge \mathsf{Open}(pp, C, S', O') = 1] \leq negl(k)$$

Definition 25 (Subset soundness)
A FHS19 set-commitment scheme has subset soundness if for all polynomial adversaries, $\mathcal{A}$, the following holds:
$\Pr[(C, S, O, T, W) \leftarrow \mathsf{Adv}(pp) :$
$\mathsf{Open}(pp, C, S, O) = 1$
$\wedge\, \mathsf{VerifySubset}(pp, C, T, W) = 1 \wedge T \not\subseteq S] \leq negl(k)$

Definition 26 (Hiding)
A FHS19 set-commitment scheme is hiding if for all polynomial adversaries, $\mathcal{A}$, the following holds:
$\Pr[(S_0, S_1, st) \leftarrow \mathsf{Adv}(pp);$
$b \leftarrow \{0, 1\};$
$(C, O) \leftarrow \mathsf{Commit}(pp, S_b) :$
$(b') \leftarrow \mathsf{Adv}^{\mathsf{OpenSubset}(pp, \mathsf{C}, \mathsf{S_b}, \cdot \cap \mathbb{S}_0 \cap \mathsf{S_1})}(pp, C, st);$
$b' = b] \leq \frac{1}{2} + negl(k)$

B Additional constructions

B.1 Set commitments on equivalence classes (CoEC) construction

Here we construct a scheme which satifies the definitions in Section 3.3.

CoEC scheme intuition We're going to generate a number of random points (we'll call them "bases"). Then, given a set of messages, M, a committer will construct two group elements such that their discrete log is dependent on the messages as well as the discrete log of the random points (relative to a generator of the group, P). Because the discrete log of the commitment is dependent on the discrete log of these random points, we'll see that an attempt to forge or distinguish commitments reduces to the computational and decisional Diffie-Hellman problems. Because mercurial signatures use type III bilinear pairings, a hash function exists for at least one of the bilinear groups [GPS06, Cos12]. Thus, in practice, we can create these random bases verifiably using the hash function. Though we can also use a CRS to generate these.

Definition 27 (CoEC construction) *A commitment scheme on equivalence classes has the following functions:*

- $\mathsf{InitializeCoEC}(1^k, 1^s) \rightarrow pp$*: Compute pairs of random points:* $(B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}, ..., B_{s-1,1}) \xleftarrow{\$} (\mathbb{G}^2)^s$*. Output these points as* pp*.*
- $\mathsf{Commit}(pp, M) \rightarrow (C, O)$*: Let* $M = \{m_0, m_1, ..., m_{s-1}\}$*. Generate a random* $\alpha \leftarrow \mathbb{Z}_p^*$*. Compute a vector of size* 2*:* $C = (C_0, C_1)$*,* $C_0 = \left(\sum_{i=0}^{s-1} A_{i,0}\right)$ *and*

$C_1 = \left(\sum_{i=0}^{s-1} A_{i,1}\right)$ *where* $A_{i,0} = (\alpha B_{i,0})$ *and* $A_{i,1} = (\alpha m_i B_{i,1})$ *and* $O = \alpha$.

- $\mathsf{RandomizeCom}(pp, C, O)$*:* $\beta \leftarrow \mathbb{Z}_p^*$, $C' = \beta C$, $O = \beta * \alpha$.
- $\mathsf{Open}(pp, C, M, O)$*: Compute* $C' = \mathsf{Commit}(pp, M; O)$ *where* $\alpha = O$ *when computing* Commit*. Ensure that* $C' = C$.

We prove Theorem 1 in Appendix D.2.

B.2 Zero knowledge subset proofs over commitments (zk-SPoC) construction

We'll use the commitment scheme from [FHS14] for this scheme which we'll refer to as the "FHS19 commitment scheme." This commitment scheme is described in Appendix A.3. To help the reader understand zk-SPoCs, we'll first construct them with trivial blinding factors (equal to 1) and have each commitment be to a single message instead of a set. This will allow us to show off the complicated parts of the scheme in the simplest manner.

Reviewing the goal of zk-SPoCs, we want to create a witness based on two vectors of commitments $(\vec{A}, \vec{B})$ to two vectors of messages $(\vec{L}, \vec{M})$ of size m and n such that $\forall i \in [m], \exists r_i : \mathsf{Open}(\vec{A}_i, \vec{L}_i, r_i) = 1$ and $\forall i \in [n], \exists s_i : \mathsf{Open}(\vec{B}_i, \vec{M}_i, s_i) = 1$. In this instantiation, we consider commitments in $\mathbb{G}_2$ but there is a symmetric scheme for commitments in $\mathbb{G}_1$. Let $\mathcal{L}$ be the set of messages $\{\vec{L}_i\}_{i\in[m]}$ and let $\mathcal{M}$ be the set of messages $\{\vec{M}_i\}_{i\in[n]}$. We want any verifier with our witness to be convinced that $\mathcal{L} \subseteq \mathcal{M}$ (and that all the above relations hold such as a known opening) given only the commitments and the witness.

Let us define $\mathcal{N}_i = \{\vec{M}_1, \vec{M}_2, ..., \vec{M}_i\}$ ($\mathcal{N}_i$ is the set of messages with index less than or equal to i). We will compute a commitment, $\vec{C}_i$, to each $\mathcal{N}_i$. Thus, $\vec{C}_n$ is a commitment to $\mathcal{N}_n = \mathcal{M}$. Next, we use each element of $\vec{B}$ as a "witness" in a FHS19-style subset opening: $e(\vec{C}_i, \vec{B}_{i+1}) = e(\vec{C}_{i+1}, \hat{P})$. Starting with $e(g, \vec{B}_0) = e(\vec{C}_0, \hat{P})$ and ending with: $e(\vec{C}_{n-1}, \vec{B}_n) = e(\vec{C}_n, \hat{P})$. Thus, each successive $\vec{C}_i$ is a commitment to a superset of the last commitment and includes the message from another commitment in $\vec{B}$. Using each $\vec{C}_i$ to recompute these equivalencies, a verifier should be convinced that $\vec{C}_n$ is a commitment to $\mathcal{M}$. Next, we iterate through each commitment in $\vec{A}$ and prove that its corresponding message is committed to by $\vec{C}_n$: $e(\vec{W}_i, \vec{A}_i) = e(\vec{C}_n, \hat{P})$. Where the witness, $\vec{W}_i$ is an FHS19 commitment to $\mathcal{M} \setminus \{\vec{L}_i\}$. To take your intuition from this toy example to the full construction, there are three more important parts: (1) we use a concatenation function to merge the attributes in each set into a single attribute for a larger commitment scheme (a technique from [CHK$^+$06]). This allows our real construction to work for sets of messages instead of single messages. (2) We use blinding factors and need to include NIZKs to ensure they can be extracted. (3) We create a NIZK for proving the bilinear relations. This NIZK ensures that the construction composes well into larger protocols.

The concatenation function requires that the size of the elliptic curves used for zk-SPoCs can contain all attributes, meaning it needs enough bits for attributes. If attributes need to use the 256 bits for each attribute, meaning we have commitments in a group of size 256 in bits, we can use a proof of equivalent discrete log [CM99, COPZ22] in order to still compute this concatenation function. For example, if we want to combine two attributes of size 256, we can use elliptic curve groups of size 512 bits for the zk-SPoC scheme and use a proof of equivalent discrete log to first convert the regular 256 bit commitments into commitments of size: 512 bits and then proceed with the concatenation function with no issues.

Definition 28 (A zk-SPoC scheme)

- $\mathsf{ProveSubset}(pp, \{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]}) \rightarrow (W)$*:*
  1. *First, create new commitments, A'_i, B'_i, to $L'_i = c(L_i)$ and $M'_i = c(M_i)$ respectively with openings O'_i, P'_i such that $|A'| = |A|$ and $|B'| = |B|$. Compute NIZKs to prove equivalence of these commitments.*
  2. *Define variables as described in the toy example: Define $\mathcal{N}_i = \{M'_1, M'_2, ..., M'_i\}$, i.e.: $\mathcal{N}_i$ is the set of messages with index less than or equal to i.*
  3. *Compute a commitment, C_i, to each $\mathcal{N}_i$, in $\mathbb{G}_1$.*
  4. *Prove that each C_i is a commitment to both $\mathcal{N}_{i-1}$ and M'_i by using the given commitments as witnesses: $e(C_{i-1}, B'_i) \cong e(C_i, \hat{P})$. Where $\cong$ is a NIZK proof that the discrete log is known between the two values.*
  5. *Iterate through each element in A and construct a witness to prove that it is committed to a subset of C_n: $e(W_i, A'_i) \cong e(C_n, \hat{P})$. This is done by computing W_i as an FHS19 commitment to the vector of messages: $\{M'_i\}_{j\in[m]} \setminus \{L_i\}$.*
  6. *Output each iterative commitment, C_i, witness, W_i, and NIZK proofs, π_i.*
- $\mathsf{VerifSubset}(pp, \vec{A}, \vec{B}, W) \rightarrow (1$ *or* $0)$*: Use the NIZKs in W to verify the commitments $\vec{B'}$ and $\vec{A'}$. Recompute all values in $\mathbb{G}_T$ from the given commitments: $e(\vec{C}_{i-1}, \vec{B}'_i)$, $e(\vec{C}_i, \hat{P})$, $e(\vec{W}_i, \vec{A}'_i)$, and $e(\vec{C}_n, \hat{P})$ for all i. Use the NIZKs in W to verify that: $e(\vec{C}_{i-1}, \vec{B}'_i) \cong e(\vec{C}_i, \hat{P})$ and $e(\vec{W}_i, \vec{A}'_i) \cong e(\vec{C}_n, \hat{P})$.*

B.3 Verifiable random functions for mercurial signatures

Given a mercurial public key: $PK = PK_0, PK_1 \in \mathbb{G}_1$, secret key: $sk = sk_0, sk_1 \in \mathbb{Z}_p$, and input to the VRF, x, where $\mathbb{G}_1 = \langle P \rangle$, compute a VRF: $\pi_Y = NIZK\left[s = \mathsf{dlog}_{sk_0}(sk_1) : PK_0 * s = PK_1 \wedge (1/(s+x))P = Y\right]$ The latter part of the AND proof can be computed using discrete log proofs in parallel [HL10, Dam10].

C Formal definition of PACIFIC

In this section, we restate the theorems and definitions from Section 3.1 in more formal terms in order to have a more precise proof.

Labels. We use a number of ciphertext spaces to define sets and maps: $\mathcal{CH}$ is the set of all chirps (i.e. $\mathcal{CH} = \mathsf{Image}(\mathsf{Chirp})$), $\mathcal{PK}$ is the set of all public keys, $\mathcal{SK}$ is the set of all secret keys, $\mathcal{BA}$ is the set of all batches, and $\mathcal{CRT}$ is the set of all user certificates. $\mathcal{ID}$ is the set of all user identities. $\mathcal{T}$ is a set of all possible times (totally ordered). $\mathcal{L}$ is the set of all possible locations.

$\mathsf{SetupGame}(1^\lambda, e) \to (pp, PK_{rp})$: ***Initialize global state for the game.*** *To store the state, we use a number of (initially empty) lists and maps shown below. Maps initially send all user handles to $\perp$. The set of user handles, times, and locations are defined by the construction but must be at least exponential in the size of the security paramter. Each of these sets can be thought of as a large set of integers.*

- $\mathsf{HU} : \mathcal{ID} \to \mathcal{PK}$: ***Honest user public keys.*** *A map from user handles to user public keys.*
- $\mathsf{HU}_{sk} : \mathcal{ID} \to \mathcal{SK}$: ***Honest user secret keys.*** *A map from user handles to user secret keys.*
- $\mathsf{HCert} : \mathcal{ID} \to \mathcal{CRT}$: ***Honest user certificates.*** *A map from user handles to user certificates.*
- $\mathsf{HDB} : \mathcal{ID} \to \mathcal{BA}$: ***Batches uploaded by honest users.*** *Maps honest user handles to batches.*
- $\mathsf{RC} \subseteq (\mathcal{SK} \times \mathcal{T} \times \mathcal{L} \times \mathcal{CH})$: ***Received chirps*** *A multiset containing tuples that represent chirps received by the adversary. Each tuple contains an honest user's secret key, time, location, and a chirp. The user handle corresponds to the honest user that sent the chirp.*
- $\mathsf{SC} \subseteq (\mathcal{SK} \times \mathcal{T} \times \mathcal{L} \times \mathcal{CH})$: ***Sent chirps*** *A multiset containing tuples that represent chirps sent by the adversary that were accepted. Each tuple contains an honest user's secret key, time, location, and a chirp. The user handle corresponds to the honest user that received the chirp.*
- $\mathsf{HI} \subseteq (\mathcal{SK} \times \mathcal{SK} \times \mathcal{T} \times \mathcal{L} \times \mathcal{CH})$: ***Honest interactions*** *A set containing tuples that represent interactions between honest users that the adversary did not hear or modify. Each tuple contains two user handles (indicating the sender and then the recipient), a time, a location, and a chirp.*
- $\mathsf{CU} \subseteq \mathcal{PK}$: ***Corrupted users.*** *A set of public keys of corrupted users.*
- $\mathsf{CDB} \subseteq \mathcal{BA}$: ***Batches uploaded by corrupted users.*** *A set that records batches uploaded by corrupted users.*

- t_{now}: ***The current time.*** *An integer that is initialized to* 0.
- pp: ***Public parameters for the game.*** *Initialized to* $pp \leftarrow \mathsf{ParamGen}(1^\lambda, e)$.
- PK_{rp}, sk_{rp}: ***Registration party keys.*** *Initialized to* $sk_{rp}, PK_{rp} \leftarrow \mathsf{RegPartyKeyGen}(pp)$.

Return (pp, PK_{rp}).

Below, we describe the set of oracles that can be called by the adversary in our games. We will use different sets of oracles for different games. Each oracle shares global state and is run by a challenger.

- $\mathsf{RegisterHonest}(1^\lambda) \to (i, PK_{\mathcal{U}})$: **Generate and register an honest user.** Generate a key pair, $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{UserKeyGen}(pp, PK_{rp})$. If this outputs $\perp$, abort and return $\perp$. Pick an $i \in \mathcal{ID}$ such that $\mathsf{HU}(i) = \perp$. Update HU such that $\mathsf{HU}(i) = PK_{\mathcal{U}}$. Update HU_{sk} such that $\mathsf{HU}_{sk}(i) = sk_{\mathcal{U}}$. Generate a certificate for user i: $cert \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}})$. Store their certificate by defining: $\mathsf{HCert}(i) = cert$. Return the handle for this honest user, i and the public key, $PK_{\mathcal{U}}$.
- $\mathsf{RegisterCorrupt}(PK_{\mathcal{U}}) \to (cert)$: **Register a corrupted user.** Add $PK_{\mathcal{U}}$ to CU and return a certificate on $PK_{\mathcal{U}}$: $cert \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}})$.
- $\mathsf{RecvChirp}(i, l) \to (\mathsf{c})$: **Receive a chirp from an honest user.** If $\mathsf{HU}(i) = \perp$ or a tuple like $(\mathsf{HU}_{sk}(i), t_{\mathsf{now}}, *)$ exists in RC, abort and return $\perp$. Compute: $\mathsf{c} \leftarrow \mathsf{Chirp}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, \mathsf{Hcert}(i), t_{\mathsf{now}}, l)$. Add $(\mathsf{HU}_{sk}(i), t_{\mathsf{now}}, l, \mathsf{c})$ to RC. Return c.
- $\mathsf{SendChirp}(i, l, \mathsf{c}) \to (\perp)$: **Send a chirp to an honest user.** If $\mathsf{HU}(i) = \perp$ or a tuple like $(\mathsf{HU}_{sk}(i), t_{\mathsf{now}}, l')$ exists in SC such that $l' \neq l$, abort and return $\perp$. Compute: $\mathsf{Listen}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, t_{\mathsf{now}}, l, \mathsf{c})$ and if this outputs 1, we add $(\mathsf{HU}_{sk}(i), t_{\mathsf{now}}, l, \mathsf{c})$ to SC.
- $\mathsf{HonestInteraction}(i, j, l) \to (\perp)$: **Have two honest users interact.** Run $\mathsf{c} \leftarrow \mathsf{RecvChirp}(i, l)$ but don't update RC. If $\mathsf{c} \neq \perp$, run $\mathsf{SendChirp}(j, l, \mathsf{c})$ but don't update SC. If $\mathsf{HU}_{sk}(i), \mathsf{HU}_{sk}(j) \neq \perp$, add $(\mathsf{HU}_{sk}(i), \mathsf{HU}_{sk}(j), t, l, \mathsf{c})$ to HI.
- $\mathsf{IncrementTime}(1^\lambda) \to (\perp)$: **Increment time.** Set $t_{\mathsf{now}} = t_{\mathsf{now}} + 1$. If, after the increment, $\mathsf{Epoch}(t_{\mathsf{now}}) > \mathsf{Epoch}(t_{\mathsf{now}} - 1)$, reset the database by setting: $\mathsf{HDB}(i) = \perp$ for all $i \in \mathcal{ID}$ and resetting $\mathsf{SC}, \mathsf{RC}, \mathsf{CDB}$ each to an empty set.
- $\mathsf{HonestUpload}(i) \to (B)$: **Have an honest user upload to the server.** If $\mathsf{HDB}(i) \neq \perp$, abort and return $\perp$. Otherwise, return a batch computed by the honest user on all the chirps they heard and verified during the game:
  $C = \{(\mathsf{c}) : (\mathsf{HU}_{sk}(i), *, *, \mathsf{c}) \in \mathsf{RC}\} \cup \{(\mathsf{c}) : (\mathsf{HU}_{sk}(i), *, *, *, \mathsf{c}) \in \mathsf{HI}\})$
  $B = \mathsf{Notify}(pp, \mathsf{HU}_{sk}(i), C)$
  Set $\mathsf{HDB}(i) = B$. Return B.

- $\mathsf{CorruptedUpload}(B) \to (\perp)$: **A corrupted user uploads to the database.** Compute $DB = \left(\bigcup_{i \in \mathcal{ID}} \mathsf{HDB}(i)\right) \cup \mathsf{CDB}$. If $\mathsf{VerifyBatch}(pp, PK_{rp}, DB, B, \mathsf{Epoch}(t_{\mathsf{now}}))$ outputs 1, set $\mathsf{CDB} = \mathsf{CDB} \cup \{B\}$. Return $\perp$.

Game 5 (Correctness game) *Run* $(pp, PK_{rp}) \leftarrow \mathsf{SetupGame}(1^k, e)$. *Run* $\mathcal{A}^{O_{\mathsf{corr}}}(pp, PK_{rp}, 1^k)$, *where* $O_{\mathsf{corr}} = \{\mathsf{RegisterHonest}, \mathsf{HonestInteraction}, \mathsf{IncrementTime}, \mathsf{HonestUpload}\}$. *After* $\mathcal{A}$ *exits, ensure that the counted exposures matches the interactions from the game that were uploaded:*

$$
\begin{aligned}
&DB = \bigcup_{i \in \mathcal{ID}} \mathsf{HDB}(i) \\
&\forall i \in \mathcal{ID}, \\
&\quad \mathsf{CountExposures}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, DB, t_{\mathsf{now}}) \\
&\quad = \#\{(\mathsf{HU}_{sk}(i), \mathsf{HU}_{sk}(j), t, l) \in \mathsf{HI} : \mathsf{HDB}(j) \neq \perp\}
\end{aligned}
$$

If this check fails, output 1 *indicating that the adversary won, otherwise, output 0.*

Definition 29 (Correctness) *An automated exposure notification scheme,* Π*, is correct if no probabilistic-polynomial-time adversary can win Game 5 with probability greater than negligible for all valid epoch values.*

C.1 Clone integrity

To define integrity and privacy, we define some extractors:

- $\mathcal{E}_{PK_{\mathcal{U}}} : \mathcal{PK} \to \mathcal{SK}$: Takes a public key and outputs the secret key for this public key.
- $\mathcal{E}_{\mathsf{c}} : \mathcal{CH} \to \mathcal{SK}$: Takes in a chirp and outputs a secret key of the user who created the chirp.
- $\mathcal{E}_B : \mathcal{BA} \to (\mathcal{SK} \times \mathcal{SK} \times \mathcal{T} \times \mathcal{L})^*$: Takes in a batch and outputs a list of notifications of variable size where each notification is a tuple which holds a chirper's secret key, a receiver's secret key, a time, and a location (in that order).
- $\mathcal{E}_{DB} : \mathcal{DB} \to (\mathcal{SK} \times \mathcal{SK} \times \mathcal{T} \times \mathcal{L})^*$: Takes in a database and outputs a list of notifications of variable size in a similar form to $\mathcal{E}_B$.

Game 6 (Clone integrity game) *Run* $(pp, PK_{rp}) \leftarrow \mathsf{SetupGame}(1^\lambda, e)$. *Run* $\mathcal{A}^{O_{\mathsf{integrity}}}(pp, PK_{rp}, 1^\lambda)$, *where* $O_{\mathsf{integrity}} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{HonestInteraction}, \mathsf{IncrementTime}, \mathsf{HonestUpload}, \mathsf{CorruptedUpload}\}$. *When* $\mathcal{A}$ *exits, the challenger uses the resulting global state to determine if the adversary won.*

- *The challenger computes all* ***possible interations*** *where an honest user could be exposed:*

$$\begin{aligned}\mathsf{PI} = \{&(\mathsf{HU}_{sk}(i), sk_j, l, t) \forall sk_j, \in \mathcal{E}_{PK}(\mathsf{CU}), (\mathsf{HU}_{sk}(i), l, t, *) \in \mathsf{RC}\} \\ &\cup \{(\mathsf{HU}_{sk}(i), \mathsf{HU}_{sk}(j), *, *) \in \mathsf{SC} \cup \mathsf{HI} : \mathsf{HDB}(j) \neq \perp\}\end{aligned} \tag{8}$$

- *The challenger computes the set of* ***extracted interactions*** *from the database which would notify an honest user:*

$$\mathsf{EI} = \{(sk_{\mathcal{U}}, *, *, *) \in \mathcal{E}_{DB}(pp, DB) : sk_{\mathcal{U}} \in \mathsf{HU}_{sk}\} \tag{9}$$

We are now ready to check conditions and output 1 *if any fail, indicating that the adversary wins. Otherwise, output* 0*.*

1. ***Correct exposure count.*** *Ensure the extracted interactions match the counted interactions:*

$$\begin{aligned}&\forall i \in \mathbb{Z}, \\ &\quad \mathsf{CountExposures}(pp, \mathsf{HU}_{sk}(i), PK_{rp}, DB, t_{\mathsf{now}}) \\ &\quad = \#\{(\mathsf{HU}_{sk}(i), *, *, *) \in \mathsf{EI}\}\end{aligned} \tag{10}$$

2. ***Database contains a subset of possible interactions.*** *Ensure these extractions are within the set of chirps sent by honest users during the game:*

$$\mathsf{EI} \subseteq \mathsf{PI} \tag{11}$$

3. ***Clone protection.*** *Ensure that no corrupted user was in two locations at the same time:*

$$\begin{aligned}&\nexists \big((*, sk_{\mathcal{U}}, t, l), (*, sk_{\mathcal{U}}{}', t', l')\big) \in \mathsf{EI} \\ &\quad s.t.\ sk_{\mathcal{U}} = sk_{\mathcal{U}}{}' \wedge t = t' \wedge l \neq l'\end{aligned} \tag{12}$$

Definition 30 (Clone integrity) *An automated exposure notification scheme,* Π*, has clone integrity if there exists a set of extractors,* $\mathcal{E}$*, such that no PPT adversary can win Game 6 with probability greater than negligible for all valid epoch values.*

C.2 Privacy definitions

To define upload privacy, we first define two oracles which are simulated versions of the oracles they replace.[5]

- $\mathsf{RecvChirp}^{sim}(\perp, l) \to (\mathsf{c})$: Functions exactly like the real version, RecvChirp, except this simulated version computes the chirp with a new user. I.e.: to compute the chirp, first generate a new keypair: $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{UserKeyGen}(PK_{rp})$. Then register this user: $cert \leftarrow \mathsf{RegisterUser}(sk_{rp}, PK_{\mathcal{U}})$. Compute a chirp for this user on the given time and location, $\mathsf{c} \leftarrow \mathsf{Chirp}(sk_{\mathcal{U}}, PK_{rp}, cert, t_{\mathsf{now}}, l)$. Output this chirp, c.

[5]Note that while a user handle is given to $\mathsf{HonestUpload}^{sim}$, it is only used to compute leakage such as batch size.

- $\mathsf{HonestUpload}^{sim}(i) \rightarrow (B)$: First, compute a new uploader key-pair, $sk_{\hat{\mathcal{U}}}, PK_{\hat{\mathcal{U}}} \leftarrow \mathsf{UserKeyGen}(pp)$ and register this user: $cert_{\hat{\mathcal{U}}} \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\hat{\mathcal{U}}})$. Next, reconstruct a simulated set of chirps, C, to be used by the Notify function. For each tuple in the receiver chirps, SC, and honest interaction chirps, HI, that indicate this uploader was the receiver, extract the secret key of the sender from the chirp: $sk_{\mathcal{U}} = \mathcal{E}_{\mathsf{c}}(\mathsf{c})$. Put the resulting interactions in a set, RC^i. , removing any duplicates. [6]
$$\begin{aligned}\mathsf{RC}^i = &\{(sk_{\mathcal{U}}, t, l) : (sk_{\mathcal{U}}, t, l, c) \in \mathsf{SC} \wedge sk_{\mathcal{U}} = \mathcal{E}_{\mathsf{c}}(c)\} \\ &\cup \{(sk_{\mathcal{U}}, t, l) : (i, j, t, l, c) \in \mathsf{HI} \wedge sk_{\mathcal{U}} = \mathsf{HU}_{sk}(j)\}\end{aligned} \tag{13}$$
Then count the number of honest senders in this set: $K_{\mathsf{HU}} = \#\{(sk_{\mathcal{U}}, t, l) \in \mathsf{RC}^i : sk_{\mathcal{U}} \in \mathsf{HU}_{sk}\}$. And then, for each adversarial user, count the number of interactions with that user: $\forall sk_{\mathcal{U}} \in \mathcal{E}_{sk_{\mathcal{U}}}(\mathsf{CU}), K_{sk} = \#\{(sk, t, l) \in \mathsf{RC}^i\}$. Then compute K_{HU} new chirps for new random users, times and locations, using $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{UserKeyGen}$, $cert \leftarrow \mathsf{RegisterUser}(sk_{rp}, PK_{\mathcal{U}})$, $t \leftarrow \mathcal{T}$, $l \leftarrow \mathcal{L}$, and $\mathsf{c} \leftarrow \mathsf{Chirp}(sk_{\mathcal{U}}, PK_{rp}, cert, t, l)$, adding the resulting chirp (c) to our simulated chirps, C. Then, for each adversarial secret key ($\forall sk_{\mathcal{U}} \in \mathcal{E}_{sk_{\mathcal{U}}}(\mathsf{CU})$) compute $K_{sk_{\mathcal{U}}}$ new chirps for the extracted adversarial secret key, with random new times and locations, doing the same as was done for honest chirps, but skipping the key generation and using the corrupted key instead (assuming the public key is included in the secret key). Label the combination of these new random chirps from honest and corrupted users as C. Now compute a batch using the new uploader and simulated chirps: $B \leftarrow \mathsf{Notify}(sk_{\hat{\mathcal{U}}}, PK_{rp}, C, t_{\mathsf{now}})$. Output this batch, B.

We are now ready to define the game:

Game 7 (Upload-privacy game) *Run* $(pp, PK_{rp}) \leftarrow \mathsf{SetupGame}(1^\lambda, e)$. *Flip a random bit,* $b \xleftarrow{\$} \{0,1\}$. *If* $b = 0$, *run* $st \leftarrow \mathcal{A}^{O_{real}}(pp, PK_{rp}, 1^\lambda)$, *otherwise, if* $b = 1$, *run* $st \leftarrow \mathcal{A}^{O_{sim}}(pp, PK_{rp}, 1^\lambda)$, *where:* $O_{real} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{IncrementTime}, \mathsf{HonestUpload}\}$ *and* $O_{sim} = \{\mathsf{RegisterHonest}, \mathsf{RegisterCorrupt}, \mathsf{RecvChirp}^{sim}, \mathsf{SendChirp}, \mathsf{IncrementTime}, \mathsf{HonestUpload}^{sim}\}$. *Next, run* $b' \leftarrow \mathcal{A}(sk_{rp}, st)$. *The adversary wins if* $b = b'$.

Definition 31 (Upload privacy) *An automated contact tracing scheme,* Π, *is privacy-preserving if there exists a set of extractors,* $\mathcal{E}$, *such that no PPT adversary has greater than* $\frac{1}{2} + negl(\lambda)$ *advantage in Game 7 for all valid epoch values.*

In the chirp privacy game, we allow the adversary to create a corrupted registration party. The registration party secret key is extracted to be used in the $\mathsf{RecvChirp}^{sim}$ oracle. We define a function that allows the adversary to create their own certificates for honest users.

[6] In our construction, meeting this requirement of removing duplicates naively requires a NIZK in the chirp. If we allow duplicates, we can remove this NIZK.

- $\mathsf{RegisterHonest}^{mal}(1^\lambda) \rightarrow (i, PK_\mathcal{U})$: Generate a key pair, $sk_\mathcal{U}, PK_\mathcal{U} \leftarrow \mathsf{UserKeyGen}(pp, PK_{rp})$. If this outputs $\perp$, abort and return $\perp$. Pick an $i \in \mathbb{Z}$ such that $\mathsf{HU}(i) = \perp$. Update HU such that $\mathsf{HU}(i) = PK_\mathcal{U}$. Update HU_{sk} such that $\mathsf{HU}_{sk}(i) = sk_\mathcal{U}$. Call the adversary with $PK_\mathcal{U}$ to receive *cert*. Store their certificate by defining: $\mathsf{HCert}(i) = cert$. Return the handle for this honest user, i and the public key, $PK_\mathcal{U}$.

Game 8 (Chirp privacy game) *Run* $(pp, *) \leftarrow \mathsf{SetupGame}(1^\lambda, e)$. *Run* $(PK'_{rp}, st') \leftarrow \mathcal{A}(pp, 1^\lambda)$ *and replace the* PK_{rp} *in the global state of the game with* PK'_{rp}. *Extract* $sk_{rp} = \mathcal{E}_{PK_{rp}}(PK_{rp})$. *Flip a random bit,* $b \xleftarrow{\$} \{0, 1\}$. *If* $b = 0$, *run* $b' \leftarrow \mathcal{A}^{O_{real}}(pp, 1^\lambda, st)$, *otherwise, if* $b = 1$, *run* $b' \leftarrow \mathcal{A}^{O_{sim}}(pp, 1^\lambda, st)$, *where:* $O_{real} = \{\mathsf{RegisterHonest}^{mal}, \mathsf{RecvChirp}, \mathsf{SendChirp}, \mathsf{IncrementTime}\}$ *and* $O_{sim} = \{\mathsf{RegisterHonest}^{mal}, \mathsf{RecvChirp}^{sim}, \mathsf{SendChirp}, \mathsf{IncrementTime}\}$. *The adversary wins if* $b = b'$.

Definition 32 (Chirp privacy) *An automated exposure notification scheme,* Π, *is privacy-preserving with respect to chirps if there exists a set of extractors,* $\mathcal{E}$, *such that no PPT adversary has greater than* $\frac{1}{2} + negl(\lambda)$ *advantage in Game 8 for all valid epoch values.*

We now restate the theorems from Section 4.3 for our formal definitions.

Theorem 6 *The ProvenParrot scheme described in Definition 14 has clone integrity (meeting Definition 30) using the set of extractors,* $\mathcal{E}$ *from Definition 33.*

Theorem 7 *The ProvenParrot scheme described in Definition 14 is chirp-private (meeting Definition 32) using the set of extractors,* $\mathcal{E}$ *from Definition 33.*

Theorem 8 *The ProvenParrot scheme described in Definition 14 is upload-private (meeting Definition 31) using the set of extractors,* $\mathcal{E}$ *from Definition 33.*

D Security proofs

D.1 ProvenParrot proofs

D.1.1 Proof of Clone integrity

Let $\mathcal{E}$ be the following set of extractors. We define an extra extractor that is not required for the definition, $\mathcal{E}_B$, but we'll see that this is useful for defining the proof.

Definition 33 (Extractors for security proofs)

- $\mathcal{E}_{PK}(pp, PK)$: *Use the trapdoor to compute* ${sk_\mathcal{U}}'$ *such that* ${sk_\mathcal{U}}' P = PK$. *Normalize this so that* $sk_{\mathcal{U}0} = 1$ *(multiply* ${sk_\mathcal{U}}'_1$ *by* $({sk_\mathcal{U}}'_0)^{-1}$ *to derive* $sk_{\mathcal{U}1}$*). Output* $sk_\mathcal{U}$.

- $\mathcal{E}_c(pp, c)$*: Use* $\mathcal{E}_{PK}$ *for the given* $PK_{\mathcal{U}}$*.*
- $\mathcal{E}_{PK_{rp}}(pp, c)$*: Use* $\mathcal{E}_{PK}$ *for the given* PK_{rp}*.*
- $\mathcal{E}_B(pp, B)$*: Take* $PK'_{\hat{\mathcal{U}}}$ *from the batch and use* $\mathcal{E}_{PK}$ *to extract a secret key,* $sk_{\hat{\mathcal{U}}}$*. Also extract the attributes,* t, l*, from the NIZKs for each chirp. Extract sk from the public key in each chirp. Output a set of each interaction with* $sk_{\hat{\mathcal{U}}}$ *as the receiver.*
- $\mathcal{E}_{DB}(pp, DB)$*: Iterate through each* $B \in DB$ *and compute* $\mathcal{E}_B(B)$ *and union the result i.e. output:* $\bigcup_{B \in DB} \mathcal{E}_B(B)$*.*

The adversary in Game 6 can win if after the game, any of the 3 conditions (Equations 10, 11, and 12) in the integrity definition are false. We will go through each equation and prove that the probability of any PPT adversary violating them is negligible.

Proving Equation 12 - Clone protection. To prove that this equation holds, we'll break it up into two equations which each ensure a separate guarantee: The first equation ensures that clone protection holds within a batch:

$$\nexists ((*, *, t, l), (*, *, t', l')) \in (\mathcal{E}_B(pp, B))^2 \text{ s.t. } t = t' \wedge l \neq l' \tag{14}$$

and this second equation ensures that no two batches share a listener:

$$\forall B \in DB, \nexists B' \in DB \setminus \{B\} \text{ s.t. } \exists(*, sk_{\mathcal{U}}, *, *) \in B \wedge \exists(*, sk_{\mathcal{U}}, *, *) \in B' \tag{15}$$

We can see that, because $\mathsf{EI} = \bigcup \mathcal{E}_B(DB)$, if the Equations 14 and 15 hold, then Equation 12 holds. This is because for pair of tuples to exist in EI that violate Equation 12, they need to have the same listener, thus, by Equation 15, they are in the same batch and by Equation 14, then they must have distinct time values or the same locations.

First we'll prove Equation 15. Informally, we use the fact that the uploader must compute $f^{\mathsf{MS}}_{sk'_{\hat{\mathcal{U}}}}(d)$, and prove that this is related to $PK'_{\hat{\mathcal{U}}}$, resulting in Y^{PK}, π^{PK}. Y^{PK} is checked for uniqueness among batches in DB, during $\mathsf{VerifyBatch}$. Because DB is emptied when $\mathsf{Epoch}(d_{\mathsf{now}})$ changes, we know that each of these Y^{PK} across batches are computed on the same d. Because $f^{\mathsf{MS}}_{(\cdot)}(\cdot)$ normalizes the secret key before evaluation of the PRF, then, even if the user randomizes their keys, this function will still be evaluated on the same values, $\frac{sk_1}{sk_0}$ and d_{now}. Thus, if two batches are uploaded with equivalent public keys, then the Y^{PK} values in the batches will be the same. This will be detected and rejected during $\mathsf{VerifyBatch}$, thus ensuring Equation 15 can't be violated.

More formally, because Y^{PK} can be extracted (in the random oracle model using the NIZK associated with Y^{PK}) and because VRFs are deterministic, we know we can extract a distinct value for each computation on $(sk_{\hat{\mathcal{U}}}, d)$. Thus, if all Y^{PK} values are distinct, we must be able to extract distinct $sk_{\hat{\mathcal{U}}}, d$ from each. Because d is public and each is fixed for each proof in the database, π_{PK},

then each sk must be distinct. This formal proof is inspired by the soundness proof in [CHK+06].

Because our extractor uses this single $PK_{\hat{\mathcal{U}}}$ to output the listener value for each tuple in the batch, we can see that if one of these $sk_{\hat{\mathcal{U}}}$ values is not shared across any batch, then all of the $sk_{\hat{\mathcal{U}}}$ in the tuples of this batch are not shared with any other batch. Thus, Equation 15 cannot be violated.

For Equation 14, we see that the adversary proves that the pairs of times and locations committed to by the commitments in their chirps, $\{C'_i\}_{i\in[n]}$, are a subset of the pairs committed to by $\{C^*_i\}_{i\in[n]}$. Thus, if any of the pairs in $\{C'_i\}_{i\in[n]}$ collide such that $t_i = t_j, l_i \neq l_j$, then for $\{C^*_i\}_{i\in[n]}$ to be a super set of the messages, it must include two pairs where $t_i = t_j$ (to include pairs for (t_i, l_i) and (t_j, l_j)). We also see that $Y_i^t = f^{\mathsf{MS}}_{sk_{\hat{\mathcal{U}}}}(t_i)$ is computed for each t_i committed to by $\{C^*_i\}_{i\in[n]}$. Thus, because each Y_i^t is computed solely on the t value, if these values are computed honestly, we'll see that $Y_i^t = Y_j^t$ (for $i \neq j$) and we'll be able to reject this batch in $\mathsf{VerifyBatch}$ (thus, it will not be extracted, so this property cannot be violated). Attempting to include a false Y_i^t or Y_j^t requires the adversary to produce a fake proof of subset π^{tl} or a fake proof for at least one of the VRFs, π_i^t or π_j^t. Because the proof also opens the commitments, if the adversary computes a Y_i^t for a t_i which the honest chirper did not intend, we can use this extracted opening from π_i^t along with the honest adversary's opening to double open the commitment, thus breaking Definition 8.

Proving Equation 10 - Correct exposure count. This equation ensures that the count of exposures by each user matches the extracted interactions from the database, fulfilling the equality in Equation 10.

We first need to prove that there's no duplicated tuples extracted in the batch that indicate that a chirp from the same honest user at the same time and location was sent. Because $\mathsf{CountExposures}$ would double count this, while extracting a duplicate into a set would only count it once, we need to ensure this doesn't happen. So we need to prove that a batch does not contain duplicates of any interaction where the sender is honest. Let's reduce an adversary that violates the integrity in this way to an adversary that can open a commitment to two openings or a mercurial signature forgery. An honest user will never chirp twice on one t and so they will never sign two commitments that share a t and have difference nonces (r). An uploader must compute a VRF on the nonce committed to by the chirp, $f^{\mathsf{MS}}_{sk_{\hat{\mathcal{U}}}}(r)$. We can see that simply rerandomizing another chirp honestly will not yield a new PRF output as it is still committed to the same nonce and private key class. Thus, if the adversary is able to produce a distinct Y_i^r for this rerandomized proof, they must have been able to open the commitment up to another r'_i, forged a signature on a new commitment, or violated the soundness of the NIZK.

More formally, let's say this adversary produced two commitments C, C' such that we extract the same tuple from them using the extractor $\mathcal{E}_B$. If $[C]_{\mathcal{R}} = [C]'_{\mathcal{R}}$, because the adversary creates a proof of knowledge of each attribute that the commitment is committed to as well as the opening information, we can

extract this information from both openings and output this in the game for Definition 8. If these two commitments are not in the same equivalence class, and we have $t = t', r \neq r', PK_{\mathcal{U}} = PK_{\mathcal{U}}{}'$, then we have another two cases: (1) this is a forgery in the game in Definition 19, since honest users never sign two nonces for a single t in the PACIFIC game. Or (2) one of these is a double opening from another commitment the user made. Even if the adversary did not include this second commitment in the upload, we can look back on chirps that honest users made to find the opening information and attributes.

Now that we've proven that duplicates in the extraction are prevented by our assumptions on underlying schemes, we can see that Equation 10 holds from inspection of the extractor, $\mathcal{E}_B$, the CountExposures function, and MS.Recognize. CountExposures uses MS.Recognize to count exactly the number of chirper public keys in the batch where the discrete log is equivalent to the normalized user's secret key, $\mathsf{HU}_{sk}(i)$. This is exactly what we're counting in the extracted tuples, the normalized secret key from the chirper.

Proving Equation 11 - Batch reflects possible interactions. This equation ensures that the extracted interactions are a subset of a set of possible interactions.

During extraction, if we have B such that $\mathcal{E}_B(pp, B) \not\subset \mathsf{PI}$, there must exist a tuple $(sk, sk', t, l) \in \mathcal{E}_B(pp, B)$ such that (sk, sk', t, l) is not in PI (remember the format of tuples: (chirper, listener, time, location)).

Let's create a reduction $\mathcal{B}$ that plays either the commitment binding game or the mercurial signature unforgeability game. $\mathcal{B}$ acts as the challenger, setting up the fake-outbreak resistance game and providing honest responses to the oracle queries of $\mathcal{A}$ until they exit. The reduction, $\mathcal{B}$, will transform the adversary's inputs to the oracles and the adversary's state in a number of ways dependent on how the integrity game fails. We will prove that this transformation of the adversary's input will constitute a violation of either the commitment binding game or the mercurial signature unforgeability game.

We'll break down this violation into 3 different restrictions on the tuple to make the proof easier to follow. These restrictions only concern qualities of the chirper and listener and together constitute all possible combinations, thus exhausting possible pairs of uploader and listener. Let U be the set of all registered users:

$$U = \mathsf{CU} \cup \bigcup_{i \in \mathbb{Z}} \mathsf{HU}(i)$$

(Restriction 1) sk or $sk' \notin U$

(Restriction 2) $sk, sk' \in (\mathsf{HU})^2$

(Restriction 3) $sk \in \mathsf{HU}$ and $sk' \in \mathsf{CU}$.

Restriction 1 If $sk' \notin U$, then the certificate, $cert_B$, for this batch holds a forgery, since the secret keys are extracted from the public keys and signatures

of these public keys are contained within the certificates. The secret key for the uploader, sk', is extracted from $PK'_{\hat{\mathcal{U}}}$, which $cert'$ has a signature on. Also, because the batch also includes randomized versions of each chirp's certificate, if $sk \notin U$, then there exists a certificate with a forgery in the batch.

Restriction 2 Let's look at the case where $sk, sk' \in (\mathsf{HU})^2$. We know from equation 15 that no two batches share an uploader. Because honest users upload their batches before the adversary learns them, the adversary cannot upload a second batch by replaying that user's public key and signature. This is because the Y^{PK} would be the same for the second upload, and so the database would reject it in the $\mathsf{VerifyBatch}$ function. This would hold even if the adversary could recover the user's private key to recompute NIZKs in the batch. Thus this interaction must have come from an honest user's batch and is in PI due to the correctness of the scheme.

Restriction 3 Our last, but most important case: $sk \in \mathsf{HU}$ and $sk' \in \mathsf{CU}$, indicates a fake exposure. Because, for every chirp that an honest user made, we add a tuple to PI^{HC} for each $sk' \in \mathsf{CU}$, this means that the adversary never signed a commitment with t, l. Thus, the adversary must have either opened this commitment to another t, l or forged a signature. The reduction, $\mathcal{B}$, is assured that the adversary knows the t, l as well as the opening for this commitment because the NIZK proves knowledge of these witnesses in the batch. Similarly to our proof Equation 10, we'll break this into the case where this violation shares an equivalence class with another commitment and the case where it doesn't. If this violating commitment shares an equivalence class with a commitment in another chirp, the adversary violated the commitment scheme to open the commitment in a second way. Thus, $\mathcal{B}$, can recover this information from the adversary and use the original opening information and attributes from the game to produce a double opening. If this commitment doesn't share an equivalence class with any other commitment signed by the user, then it is a forgery in the mercurial signature scheme.

Note that to find these double openings from the game, we can choose a commitment in the game randomly. Because the adversary can only call a polynomial number of functions, we have a non-negligible chance of producing a double opening.

To formalize the forgery, the reduction would have to choose a user and allow the mercurial signatures challenger to sign for them instead of producing the signatures themselves. This is identical to the fake-outbreak resistance game and so the adversary acts as normal. By rerunning the game multiple times and choosing this user randomly, the reduction has a non-negligible chance that the adversary will forge a signature for this user (since the adversary can only create a polynomial number of users) and thus, we output the adversary's forgery to win. This proves that Equation 11 cannot be violated given the security definitions of our commitment scheme along with the unforgeability of mercurial signatures.

We have now proven that Equations 11, 10, and 14 cannot be violated by a PPT adversary given our construction and extractors, thus proving Theorem 6 which is part of Theorem 5.

D.1.2 Proof of chirp privacy

Proof intuition. We're going to reduce this to the mercurial public key class-hiding game. In the ideal function, $\mathsf{RecvChirp}^{ideal}$, we've used a new random user for each simulated chirp. The mercurial public key class-hiding game This means we need to construct a hybrid argument where we iteratively replace calls of $\mathsf{RecvChirp}$ with simulated values using the PK_2^b given to us from the mercurial signature public key class-hiding game. We do not need to hide the attributes since this is given as input to the chirp function in the ideal function.

Proof of Theorem 7 Let's say an adversary can win in Game 8 with non negligible probability. Using this adversary, we'll create a reduction $\mathcal{B}^{\mathsf{MS}}$ that wins the mercurial public key class-hiding game.

Let q be the maximum number of times that the adversary queries any oracle. Thus, this q bounds the number of times the adversary queries $\mathsf{RecvChirp}$ for any user and the number of honest users the adversary creates. Note that if the adversary is PPT, then q is less than $p(\lambda)$ for some polynomial, p. Let $Hybrid_{i,j}$ be similar to Game 8 but for the first j chirps computed by any user whose handle is in $\{0, ..., i-1\}$, the keys and certificate $(sk_{\mathcal{U}}, PK_{\mathcal{U}}, cert)$ for the chirp are replaced by new, randomly generated keys $(sk_{\mathcal{U}}, PK_{\mathcal{U}})$ and a freshly issued certificate on those keys. These hybrids use the original location given to $\mathsf{RecvChirp}$, l, and the time stored in the global state, t_{now}. Note that: for $i < q$, $Hybrid_{i,q+1}$ and $Hybrid_{i+1,0}$ are identical since for $j = 0$ the hybrid doesn't replace any chirps for user $i + 1$ and for $j = q + 1$, the hybrid has replaced all the chirps for user i that the adversary could have received.

Let $\mathcal{B}^{\mathsf{MS}}$ be a reduction that chooses a random $Hybrid_{i,j}$ to act like, but acts differently for the j-th chirp from user i. During the registration for user i, the reduction uses sk_1, PK_1 given by the class-hiding game. Then, for the j-th chirp for that user, the reduction uses PK_2^b for that chirp. We can see that if $q \geq j \geq 0$, this reduction looks like $Hybrid_{i,j+b}$. Thus, which hybrid we look like depends on the bit in the class-hiding scheme and thus distinguishing the hybrids is equivalent to distinguishing the public keys in the class-hiding game.

Because $Hybrid_{0,0}$ is our real game and $Hybrid_{q,q}$ is our simulated game, and we've shown that distinguishing each step is negligible, we can see that these two games are indistinguishable.

In order to complete this reduction, we need to show that sk_{rp} is accessible by a PPT reduction. This is why we have the adversary include a NIZK that they know the secret key of the registration party. In the random oracle model, this NIZK proves that sk_{rp} is somewhere in the adversary's state. If this extraction fails, the proof of the registration party's key must be incorrect and thus the adversary cannot create any honest users in the game as they will all output $\perp$, aborting the honest user creation.

D.1.3 Proof of upload privacy

Proof intuition. Similarly to the proof of chirp privacy, we will reduce this to the hiding definitions of mercurial signatures. In contrast to the proof of chirp privacy, we also need to simulate the time and locations of interactions. We'll see that distinguishing the simulator (which uses random times and locations instead of the real ones) will reduce to the class-hiding or origin-hiding properties of our CoEC commitment scheme.

Proof of Theorem 8 Let's say an adversary can win in Game 7 with non negligible probability. We'll create a reduction $\mathcal{B}^{\mathsf{MS}}$ that plays the mercurial public key class-hiding game and a reduction $\mathcal{B}^{\mathsf{Com}}$ that plays the commitment hiding game (for Set commitments on equivalence classes in Definition 9), both using this assumed adversary that can win Game 8.

Let q be defined as in the proof for Theorem 7. Let $Hybrid^{\mathsf{c}}_{i,j}$ use fresh, random users and certificates for the first j chirps from users with handles in $\{0, ..., i-1\}$ just like the hybrids in the proof of Theorem 7. Let $Hybrid^{B}_{i,j}$ use the simulator for the first j batches uploaded by user i with batches from the simulator [7]. Note that: for $i < q$, similarly to the proof of Theorem 7, $Hybrid^{\mathsf{c}}_{i,q+1} = Hybrid^{\mathsf{c}}_{i+1,0}$ and $Hybrid^{B}_{i,q+1} = Hybrid^{B}_{i+1,0}$. Also, observe that $Hybrid^{\mathsf{c}}_{q+1,q+1} = Hybrid^{B}_{0,0}$.

Claim 1 *$Hybrid^{B}_{i,j}$ is indistinguishable from $Hybrid^{B}_{i,j+1}$*

We will now construct hybrids and reductions to prove Claim 1. We construct one hybrid for each chirp given to Notify and the simulator. We first create hybrids for the honest chirps, which we'll label $Hybrid^{B,\mathcal{HU}}_{i,j,k}$. This hybrid acts like $Hybrid^{B}_{i,j}$ but, for the chirps $\{0, ..., k-1\}$, passed to Notify, we generate each of these chirps using a freshly registered user: $sk_{\mathcal{U}}, PK_{\mathcal{U}} \leftarrow \mathsf{UserKeyGen}(1^k)$, $cert \leftarrow \mathsf{RegisterUser}(pp, sk_{rp}, PK_{\mathcal{U}})$. Also, in these simulated chirps, we create commitments on random $t', l' \xleftarrow{\$} (\mathbb{Z}^*_p)^2$ and use these as input to the Chirp function. Next, we'll create hybrids for chirps from corrupted users, $Hybrid^{B,\mathcal{CU}}_{i,j,k}$. These hybrids simulate all chirps for honest users like $Hybrid^{B,\mathcal{HU}}_{i,j,q+1}$ but simulate a fraction of the corrupted chirps, $\{0, ..., k-1\}$. The corrupted chirps are simulated in the same ways as the honest chirps, but the secret keys are not regenerated. Instead the simulated chirps are created using the secret keys of the adversaries that sent chirps to this user during this time period.

Claim 2 *$Hybrid^{B,\mathsf{CU}}_{i,j,k}$ is indistinguishable from $Hybrid^{B,\mathsf{CU}}_{i,j,k+1}$*

[7] A reader that just read the integrity definition might be confused as to why there are multiple batches for a single user. This is because the adversary's decision in the privacy game can be informed by older batches from previous time periods. These batches are not considered in the integrity game.

Proof of Claim 2 If we only replace the public key and its certificate, $PK_{\mathcal{U}}, cert$, by randomizing them, the origin-hiding of $\mathsf{MS.ChangeRep}$ proves that this new public key and signature looks entirely independent from any other public key aside from the fact that it is in the same equivalence class, which the adversary already learns in the real game using $\mathsf{MS.Recognize}$ because they have the associated secret key. Thus, replacing the public key and certificate from the batch is indistinguishable. Next, we can replace the commitment and reduce this to our hiding game for CoEC by creating a reduction, $\mathcal{B}^{\mathsf{Com}}$ that chooses M_0 as the original attributes (t, l, d, r) for this chirp and M_1 as random attributes. Depending on what CoEC returned to our reduction, we would either be replacing the commitment with random attributes or not. This reduction can be done both for C_i' and C_i^{tl}. Because we use a NIZK to prove that C_i' and C_i^{tl} agree on attributes, we can simulate the NIZK to replace the two commitments one-by-one using hybrids. More formally, we create a hybrid that replaces C_i' and we simulate the NIZK that proves its equivalence to C_i^{tl}. Then, we reduce this to the CoEC hiding game as described ($\mathcal{B}^{\mathsf{Com}}$). Then we replace both C_i' and C_i^{tl} with independently random attributes, simulate the NIZK and create a reduction similar to the described $\mathcal{B}^{\mathsf{Com}}$, but this time to another commitment scheme which only has the two attributes t and l (but still has the same bases). We can see that our hiding should still hold for two attributes as this is just another valid commitment scheme for two attributes instead of four[8].

Claim 3 *$Hybrid_{i,j,k}^{B,\mathsf{HU}}$ is indistinguishable from $Hybrid_{i,j,k+1}^{B,\mathsf{HU}}$*

Proof of Claim 3 First, we consider a simulated chirp where we replace the honest user's sk with a new, randomly generated one, along with the public key PK. We can see that if the adversary can distinguish this from the honest user's public key, we can reduce to the public key class-hiding game by using the PK_1 given to the reduction in the public key class-hiding game as the user i when they are registered. Since this hybrid already has simulated chirps for all $\mathsf{RecvChirp}$ outputs, we can generate random users for these chirps as this reduction. Continuing to act like $Hybrid_{i,j,k}^{B,\mathsf{HU}}$, we simulate chirps for the Notify function accordingly using $\mathsf{MS.Sign}(sk_1, \cdot)$ for any real chirps used in Notify, except for chirp k in batch j where we use $\mathsf{MS.Sign}(sk_2^b, \cdot)$. We can see that this makes our reduction look like one of these two hybrids $Hybrid_{i,j,k+b}^{B,\mathsf{HU}}$ depending on the bit for the public key class-hiding challenger. We can also replace the attribute sets in the commitment as we did for the proof of Claim 2, reducing to the hiding of the commitment scheme.

We now have to prove that the number of corrupt and honest chirps in the batch are correct. The simulated chirps are extracted from the set of chirps sent by the adversary SC and the set of chirps sent between honest users HI. The only way we could get an inaccurate number is if the adversary were able to

[8] Our CoEC scheme uses message class-hiding to prove its hiding property and can be read in Section D.2.

include two chirps for the same time for a single user, since we only count the distinct times from each user (in Equation 13). This is impossible for tuples in HI since honest users will not chirp twice in the same location. For tuples in SC, this is prevented by including a PRF of the chirper's secret key and t. We can see that if the user wanted to have one adversarial id chirp twice for a single honest user at the same time, they would need to forge the Y^r value included in the chirp and prove that it is valid.

Claim 4 *$Hybrid^{B,\mathsf{CU}}_{i,j,q+1}$ is indistinguishable from $Hybrid^{B}_{i,j+1}$*

Proof of Claim 4 In this proof, we aren't simulating any new chirps in the batch, but rather the other elements such as $PK'_{\hat{\mathcal{U}}}$, $cert'$, Y^{PK}, π^{PK}, ... We can immediately use the indistinguishability property of VRFs/PRFs to replace all Y values with random values. This is because $f^{\mathsf{MS}}_{(\cdot)}(\cdot)$ is a PRF and each invocation by an honest user depends on secrets not known to the adversaries (the user's secret key). Since the user is honest, we know that none of these Y values collide (they never listen for the same t at multiple locations and they never upload twice for a given $\mathsf{Epoch}(t)$[9]). We can then simulate all proofs π using the simulator for the NIZK. This includes running the simulator for the proof of subset. All C^*_i can then be randomly generated as shown by the simulator and any PRFs computed on them can be simulated.

Claim 5 *$Hybrid^{B}_{i,q+1}$ is indistinguishable from $Hybrid^{B}_{i+1,0}$*

Proof of Claim 5 From Claims 4, 3, and 2, we can see that Claim 1 is true. Now from Claim 1, the fact that $Hybrid^{\mathsf{c}}_{0,0} = Hybrid^{\mathsf{c}}_{q+1,q+1}$ (shown in the proof of chirp privacy), $Hybrid^{\mathsf{c}}_{q+1,q+1} = Hybrid^{B}_{0,0}$ as well as the fact that $Hybrid^{B}_{i,q+1} = Hybrid^{B}_{i+1,0}$, we find that $Hybrid^{\mathsf{c}}_{0,0}$ is indistinguishable from $Hybrid^{B}_{q+1,q+1}$. Noticing that $Hybrid^{\mathsf{c}}_{0,0}$ is the real game and $Hybrid^{B}_{q+1,q+1}$ is the ideal game proves Theorem 7.

D.2 Proofs for CoECs

Theorem 9 *The commitment scheme in Definition 27 is binding as defined by Definition 8 under the discrete logarithm problem.*

Proof of Theorem 9 Breaking this definition means this adversary outputs (C, C', M, O, M', O') where $M \neq M'$, $[C]_{\mathcal{R}} = [C']_{\mathcal{R}}$ and both C, M, O and C', M', O' are valid openings. Let's first look at the scenario where there's only one pair of bases, $B_{0,0}, B_{0,1}$ ($s = 1$). Because these were valid openings, we know that: $C = (\alpha B_{0,0}, \alpha m B_{0,1})$ $C' = (\beta B_{0,0}, \beta m' B_{0,1})$ where $O = \alpha$ and $O' = \beta$. If $[C]_{\mathcal{R}} = [C']_{\mathcal{R}}$, then $C_1/C_0 = C'_1/C'_0 = \frac{b_{0,1}}{b_{0,0}} m = \frac{b_{0,1}}{b_{0,0}} m'$ where $b_{i,i}$ is

[9] To ensure that no $t = \mathsf{Epoch}(t)$, we simply need to make the output of Epoch distinct from its input. Imagining the t values as a Unix timestamp, we can possibly make all outputs of Epoch negative or multiply the output by 2^{64} or similar.

$B_{i,i} = b_{i,i}P$. Thus, $m = m'$ so we have a contradiction. This implies that for one attribute, this scheme is perfectly binding. This changes to a computational hardness when we increase s.

Now we will prove this holds for $s > 1$. Because $M \neq M'$, we know $\exists j$ such that $m_j \neq m'_j$. Because $[C]_{\mathcal{R}} = [C']_{\mathcal{R}}$, we know that:
$C_1/C_0 = C'_1/C'_0$
$C_1 * C'_0 = C'_1 * C_0$
Thus

$$\alpha(\sum m_i B_{i,1}) * \beta(\sum B_{i,0}) = \alpha(\sum m'_i B_{i,1}) * \beta(\sum B_{i,0})$$
$$\alpha\beta(\sum m_i B_{i,1}) * (\sum B_{i,0}) = \alpha\beta(\sum m'_i B_{i,1}) * (\sum B_{i,0})$$
$$(\sum m_i B_{i,1}) = (\sum m'_i B_{i,1})$$
$$0 = (\sum m'_i B_{i,1}) - (\sum m_i B_{i,1})$$
$$m_j B_{j,1} - m'_j B_{j,1} = (\sum_{i \in [s]\setminus\{j\}} m'_i B_{i,1}) - (\sum_{i \in [s]\setminus\{j\}} m_i B_{i,1})$$
$$(m_j - m'_j) B_{j,1} = (\sum_{i \in [s]\setminus\{j\}} m'_i B_{i,1}) - (\sum_{i \in [s]\setminus\{j\}} m_i B_{i,1}) \tag{16}$$

Because we know $m_j - m'_j$ is non-zero, the left side of Equation 16 must also be non-zero. Thus, the right side must be non-zero.

Knowing that an adversary that can break the binding of our commitment scheme produces a set with the property described in Equation 16, we'll create a reduction that solves the discrete log problem with this.

We now describe a reduction to the discrete log problem: our reduction receives $P, P' = aP$ and is tasked with recovering a. The reduction then creates a commitment scheme with any number of random bases (s) where we know all the discrete logs over P, i.e., we know all $b_{i,e}$ such that: ($b_{i,e} = B_{i,e}/P$). Except one base, $B_{j,1}$, where the reductions instead replaces put in our challenge from the discrete logarithm game, $B_{j,1} = P'$, for a random $j \in [s]$. The reduction now gives $pp = \{B_{i,0}, B_{i,1}\}_{i \in [s]}$ to the adversary. Observing Equation 16, we can see that we retrieve some C, C', M, M', O, O' such that the adversary can compute:

$$(m_k - m'_k) B_{0,1} = \sum_{i \in [s]\setminus\{k\}} (m'_i - m_i) B_{i,1}$$

Where k is the index of the attribute that the adversary has changed (which must exist due to this being a double opening). We rerun this experiment until we happen to pick $j = k$. This will happen with non-negligible probability since the j base is a randomly sampled element just like the other bases we've made: $\{aP : a \xleftarrow{\$} \mathbb{Z}_p^*\} \approx \{Q : Q \xleftarrow{\$} \mathbb{G}\}$. So for the rest of the proof, set $j = k$. Because

we know the discrete logs of all the other bases, we can find d such that:

$$(m_j - m'_j)B_{j,1} = \sum_{i \in [s] \setminus \{j\}} (m'_i - m_i) b_i P$$

$$B_{j,1} = \sum_{i \in [s] \setminus \{j\}} b_i (m'_i - m_i)/(m_j - m'_j) P$$

We can divide by $(m_j - m'_j)$ if these values are distinct, which is true because we know this is where the message sets, M and M' differ. Thus, because we set $B_{j,1} = P'$, we can recompute $a = \sum_{i \in [s] \setminus \{j\}} b_i (m'_i - m_i)/(m_0 - m'_0)$ since we know all the $b_{i,e}$ values aside from $b_{j,1}$. Thus, we find the discrete log: $a = \frac{P'}{P}$.

Theorem 10 *The commitment scheme in Definition 27 is hiding as defined by Definition 9 as long as mercurial signatures are message class-hiding as described in Definition 20 using the equivalence classes in Definition 18.*

Our reduction to the class-hiding game for mercurial signatures accepts M_1, M_2^b and computes $pp_1 = \{a_{i,0} M_1, a_{i,1} M_1\}_{i \in [s]}$ where $a_{i,0}, a_{i,1} \leftarrow_\$ \mathbb{Z}_p^*$. The reduction gives this to the adversary and retrieves M, M'. The reduction then runs the commitment scheme using $pp_2 = \{a_{i,0} M_1, a_{i,2^b} M_2^b\}_{i \in [s]}$. The reduction now computes: $C^0 = \mathsf{Commit}(pp_2, M)$ and $C^1 = \mathsf{Commit}(pp_2, M')$ and returns $C^{b'}$ to the adversary where $b' \xleftarrow{\$} \{0,1\}$. The adversary then make their guess, b^*, and if $b^* = b'$, the reduction outputs $b^\dagger = 0$ as its guess. If $b^* \neq b'$, then the reduction outputs $b^\dagger = 1$ instead.

Analysis If the secret bit for the public-key hiding challenger, b is zero, we can see that our commitment C^b is a valid commitment to M or M' and thus, the game looks identical. If $b = 1$ though, this becomes a commitment to a random set of messages, M^r, and so the adversary has no advantage in distinguishing b'. To see why this second case ($b = 1$) results in a commitment to a random message, let's first label $d_1 = M_{1,1}/M_{1,0}$ and $d_2 = M_{2,1}^1/M_{2,0}^1$. We can now see that the discrete log, C_1^1/C_0^1 is:

$$d_2(\Sigma a_{i,0})/(\Sigma m'_i a_{i,1})$$

Thus, the discrete log could be any value, depending entirely on d_2. I.e., for any set of messages, M^r, we can find a d_2 such that when $b = 1$, the commitment given to the commitment hiding adversary is a commitment to that set of messages: $\mathsf{Commit}(pp_2, M^r) = C^2$. Thus, this adversary cannot have any advantage in guessing b' when $b = 1$, but still has an identical view to the real game when $b = 0$. Thus, the reduction can defeat public-key hiding by outputting a guess, $b^\dagger = 0$ when the guess given by the commitment hiding adversary, b^* equals the reduction's secret bit, b', and outputting $b^\dagger = 1$ otherwise.

Theorem 11 *The commitment scheme in Definition 27 is class-hiding as defined by Definition 10 as long as mercurial signatures are message class-hiding as described in Definition 20 using the equivalence classes in Definition 18.*

Proof of Theorem 9 Again, our reduction takes M_1, M_2^b from the challenger and computes $pp_1 = \{a_{i,0}M_1,\ a_{i,1}M_1\}_{i\in[s]}$. The reduction gives this to the adversary and retrieves C, O, M, C', O', M'. We then generate new commitments C_0 and C_1 to M and M' (respectively) and one of these randomly to the adversary. If $M = M'$, then, because randomizing a commitment perfectly chooses a random commitment in the same equivalence class, C and C' are indistinguishable. If $M \neq M'$, then we can instead reduce using our previous reduction from the proof of Theorem 10. This is done passing the adversary's M, M' to that reduction to mercurial message class-hiding and returning the given $C^{b'}$ to the class-hiding adversary and having the reduction return $b^\dagger$ dependent on whether the adversary guesses correctly just like the proof of Theorem 10. Again, because randomizing a commitment perfectly chooses another representative in the equivalence class, being able to specify the opening (O, O') does not help the adversary distinguish this reduction since the commitments returns to the adversary will always be entirely independent of O, O'.

D.3 Proofs for zk-SPoCs

Proof of Theorem . Let $\mathcal{S}^{\mathsf{ProveSubset}}$ be a simulator as defined below:

Definition 34 ($\mathcal{S}^{\mathsf{ProveSubset}}(\{A_i\}_{i\in[m]}, \{B_i\}_{i\in[n]})$) *First, create random commitments, messages, and openings: A'_i, O'_i, L'_i and B'_i, P'_i, M'_i and simulate the proof that these are equivalent to the concatenation of the original messages. Next, create random elements, $\{W_i\}_{i\in[m]}, \{C_i\}_{i\in[n]}$ and simulate the proof that $e(C_{i-1}, B'_i) \cong e(C_i, \hat{P})$ and $e(W_i, A'_i) \cong e(C_n, \hat{P})$.*

Let's define a number of hybrid games (labeled: $Game_j^{\mathsf{ProveSubset}}$) where $Game_j^{\mathsf{ProveSubset}}$ will act like the simulator for commitments: $[1...j-1]$ then replace the j-th with elements from a FHS19 hiding challenger (shown in Definition 26) and acts like the real function for elements $[j+1...m]$ or $[j+1...n]$. Because we replace multiple elements, between $Game_j$ and $Game_{j+1}$ we'll need to also construct intermediate hybrid games: $Game_{j,A}$, $Game_{j,A,B}$, $Game_{j,A,B,C}$, and $Game_{j,A,B,C,W}$

Definition 35 ($Game_j^{\mathsf{ProveSubset}}$) *Takes $\{A_i, O_i, L_i\}_{i\in[m]}, \{B_i, P_i, M_i\}_{i\in[n]})$ from the adversary. First, create random commitments, messages, and openings: A'_i, O'_i, L'_i and B'_i, P'_i, M'_i up to $i = j-1$, then for $i = j$, use a commitment from the FHS19 challenger, supplying either $c(L_i), c(M_i)$ or the random elements: L'_i, M'_i. For $i = j+1$ and on, use the real elements $L'_i = c(L_i), M'_i = c(M_i)$. Next, create random elements, W_i, C_i up to $i = j-1$ and for $i = j$, use an FHS19 scheme on $\mathbb{G}_1$ instead of $\mathbb{G}_2$ and commit to the correct sets needed for these witnesses. For $i = j+1$ and on, generate W_i, C_i as the real function would. If "$W''_j \in S$, then W_j is taken from the challenger.*

Definition 36 ($Game_{j,A}^{\mathsf{ProveSubset}}$) *Acts like $Game_{j-1}$ but replaces A_j like $Game_j$ would.*

Definition 37 ($Game^{\mathsf{ProveSubset}}_{j,A,B}$) *Acts like $Game_{j-1}$ but replaces A_j and B_j like $Game_j$ would.*

Definition 38 ($Game^{\mathsf{ProveSubset}}_{j,A,B,C}$) *Acts like $Game_{j-1}$ but replaces A_j, B_j and C_j like $Game_j$ would.*

Definition 39 ($Game^{\mathsf{ProveSubset}}_{j,A,B,C,W}$) *Acts like $Game_{j-1}$ but replaces A_j, B_j, C_j, and W_j like $Game_j$ would.*

Let's define a number of hybrid schemes (labeled: $Hybrid_{j,A}$) as well (defined analogously s.t. $Hybrid_{j,A}$ replaces A_j with real elements instead of giving A_j to a challenger). We can see that $Game^{\mathsf{ProveSubset}}_{j,A}$ appears as $Hybrid_{j-1}$ when the challengers bit is $b = 1$ (indicating random elements are used for A_j) and is exactly like $Hybrid_{j,A}$ when $b = 0$ (indicating real elements are used for A_j). $Game^{\mathsf{ProveSubset}}_{j,A,B}$ appears as $Hybrid_{j,A}$ when the challengers bit is $b = 1$ (indicating random elements are used for B_j) and is exactly like $Hybrid_{j,A,B}$ when $b = 0$. $Game^{\mathsf{ProveSubset}}_{j,A,B,C}$ appears as $Hybrid_{j,A,B}$ when the challengers bit is $b = 1$ and is exactly like $Hybrid_{j,A,B,C}$ when $b = 0$. $Game^{\mathsf{ProveSubset}}_{j,A,B,C,W}$ appears as $Hybrid_{j,A,B,C}$ when the challengers bit is $b = 1$ and is exactly like $Hybrid_{j,A,B,C,W}$ when $b = 0$. Then, we can see that $Hybrid_{j,A,B,C,W}$ is equivalent to $Hybrid_{j+1}$, thus, allowing us to repeat the previous intuition to reach any polynomial value for j such as n or m. We can see that $Hybrid_0$ is the same as the real $\mathsf{ProveSubset}$ and $Hybrid_n$ (or $Hybrid_m$ if $m > n$) is equivalent to $\mathcal{S}^{\mathsf{ProveSubset}}$. Since all the steps reduce to the hiding of the underlying commitment scheme, $\mathcal{S}^{\mathsf{ProveSubset}}$ is indistinguishable from $\mathsf{ProveSubset}$ and thus is a valid simulator proving that our zk-SPoC construction is zero-knowledge.

Proof of Theorem 4. If an adversary can produce a W that violates soundness, we can violate the binding of FHS19 set commitments by using the NIZKs to extract the openings of A'_i and W_i. We can then use these openings to derandomize the values such that $e(W^*_i, (a - L'_i)\hat{P}) = e(C^*_n, \hat{P})$. We can do the same to extract the openings of B'_i, finding $e(C^*_{i-1}, (a - M'_i)\hat{P}) = e(C^*_i, \hat{P})$. If $\{L'_i\} \not\subseteq \{M'_i\}$, we'll be able to create a witness, W^*_i, that each A_i is committed to a message included in C^*_n and create witnesses that each successive C^*_i includes another message committed to by B. But, because this is not a correct proof, this means that $\exists j : L'_j \notin \{M'_i\}$. This means that either (1) $e(W^*_j, (a - L'_j)\hat{P}) = e(C^*_n, \hat{P})$ and we can open C^*_n to $\{M'_i\}$ which actually doesn't contain L'_j, or, (2) the adversary was able to sneak in an extra value into C^*_n. In the case of (1) we immediately have a forgery. For (2), after de randomizing, know that $C^*_n = \prod(a - M_i)\hat{P}$. This means that we can open C^*_n up to $\{M_i\}$ and produce a witness that L_j is committed to by C^*_n, which is a violation of subset soundness.

E Details

E.1 Splitting uploads across batches

In the PACIFIC scheme defined in Definition 1, if a user needs to upload shortly after a change in epoch such that they have relevant contacts from the last epoch, this scheme does not clearly support this. There are a few ways of supporting this: We can have the adversary upload two batches, one for the previous period and another from the new period. This method leaks information as users who check the database could learn which batch and thus which epoch they had an interaction with the user during. Another method would be to run two versions of the scheme simultaneously. These schemes would define epochs that are double the length of an infection window, then offset them by half their length. Thus, the user never needs to split their notifications across two batches as one of the two schemes will contain all the chirps they need. This doubles the computation and bandwidth for chirps though as chirps for both schemes need to be broadcasted. The last method is to allow batches to be verified for two epochs (using NIZKs). This retains the privacy, integrity, and complexity, but requires a relatively complicated definition and solution, so we leave this to future work.

E.2 Necessity of VRF in chirp

If we don't compute VRFs on the time in chirps, an adversary could potentially de-anonymize honest users in our scheme by chirping some prime number of times and using a different prime number in each location. Thus, if the number of interactions does not exceed the lowest of these prime numbers (and adversaries are never in two different locations), the adversary can divide the number of contacts they had with the user by each prime to determine in which location they had their interaction. If we decide that this attack is not practical, we can remove the NIZK computation from chirps, removing the text in blue from the construction. This attack would require an adversary to broadcast a number of times relative to than the maximum number of times two users could potentially interact, which could be a lot of communication within a small time span. We could also reduce this NIZK to some extra group elements as was originally done to compute proofs of VRFs [DY05], but we leave this as future work.

E.3 Preventing de-anonymization with nonces

Nonces can lead to a problem where a malicious user choses a nonce from another nearby honest user to distinguish uploaders (making two VRFs on the same nonce appear) or suppress honest users' chirps if listeners discard duplicate nonces. This can be prevented by ensuring the nonce is a hash ($H : (\mathbb{G}_2)^\ell \to \mathbb{Z}_p$) of the chirper's randomized public key, $PK_{\mathcal{U}}{}'$.