zk-Cookies: Continuous Anonymous Authentication for the Web

Alexander Frolov[*]
University of Maryland
sfrolov@umd.edu

Hal Triedman[*]
Cornell Tech
triedman@cs.cornell.edu

Ian Miers
University of Maryland
imiers@umd.edu

Abstract

We are now entering an era where the large-scale deployment of anonymous credentials seems inevitable, driven both by legislation requiring age verification and the desire to distinguish humans from bots in the face of the proliferation of AI-generated content. However, the widespread deployment of anonymous credentials faces the same security and fraud concerns as existing credentials, but without the established techniques for securing them. For non-anonymous credentials on the web today, authentication is a continuous process in which servers collect large volumes of behavioral data to protect account holders (e.g., by detecting account compromise) or to combat fraudulent behavior.

In this paper, we propose Continuous Anonymous Authentication Schemes (CAASs) and give a concrete construction and applictions for preventing credential sharing and theft. CAASs allow us to move the server-side collection, storage, and processing of these behavioral signals to the client while maintaining privacy and integrity. CAASs support, on the client side, a number of common behavioral analysis tests and analytics both for determining fraudulent behavior and updating security policies. We implement a prototype, zk-Cookies, which runs in the browser, and supports common behavioral signals such as IP address and geolocation history, browser fingerprinting, and page view history. Using this, we build a prototype application for age verification based on legacy credentials (like passports). We implement these checks efficiently in zk-SNARKs, and also show how to securely implement differentially private behavioral analytics in a zk-SNARK. The simplest version of our construction can perform the computation for an update in under 200 ms.

1 Introduction

Anonymous credentials have long been an important area of cryptographic research [14, 15, 25, 27, 28, 31, 40, 57, 61], but we are now entering an era where their large-scale deployment seems both inevitable and critically important. This momentum is driven by two major factors. First, new legislation, such as age verification laws, requires that users identify themselves for online activities, creating a strong need for privacy-preserving ways to meet these requirements [3–5, 7, 45]. Second, the proliferation of AI-generated content and activity on the web is creating an urgent need for robust, verifiable "human" identifiers to distinguish people from bots. The recent deployment of anonymous credentials by Google for age verification on Android devices is a clear signal that this technology is moving from theory to practice [11, 39]. However, while a vast amount of research has been put into the cryptography of anonymous credentials, existing systems are largely not deployable for serious, large-scale applications. This is because they lack mechanisms to prevent cloning, credential sharing, and other forms of fraud and abuse that are rampant on the web. In an anonymous credential scheme, users can prove they meet criteria—e.g., they are over 18—without revealing their identity or linking their actions over time. But this very unlinkability means that if such a credential were deployed for a real-world application, we would have none of the usual tools to detect a shared account, prompt a potentially compromised user for additional authentication, or identify a bot generating fraudulent ad views.[1]

In applications where user privacy is not paramount, this problem is well understood and managed through *continuous authentication* and monitoring. Securing modern web services requires servers to observe, analyze, and act on client metadata in a constant feedback loop [54–56]. Servers therefore track clients during and between sessions and log behavioral data. A rapid change in IP address location might indicate a compromised account; an anomalous sequence of page views may suggest a bot. This process is not static, and in fact requires some mechanism to observe normal and suspected attacker behavior and adapt defenses to changing attacker

[*]Both authors contributed equally

[1]Google's current approach relies on trusted hardware to prevent credential cloning. This approach is highly fragile, and has raised questions about access and the open web.

behavior.

These techniques, which are essential for trust and safety, seemingly depend on the provider having full visibility into user actions, often using persistent identifiers like cookies to track users and log data.[2] Making any future privacy-preserving identity system viable requires solving this core challenge: enabling fraud and abuse detection *without* compromising the very privacy the credentials aim to provide.

Continuous Anonymous Authentication. In this paper, we address this challenge by introducing the idea of *Continuous Anonymous Authentication*. A Continuous Anonymous Authentication Scheme (CAAS) reconstructs the functionality of today's fraud prevention systems while preserving user privacy. Rather than a server collecting user data for analysis, a CAAS moves this functionality to the client while ensuring they collect accurate measurements, log them, analyze them, and report them to the server (in a privacy preserving manner).

Assume, for a moment, that we can force even dishonest clients to faithfully execute stateful programs, and consider two concrete examples, common in the non-private web today, that a server may wish to enforce:

- *Anomalous login detection:* A service may want to prevent account sharing or theft by ensuring a user isn't accessing a resource from multiple geographically disparate locations within a short time period. With a CAAS, the credential privately stores an authenticated history of recent login locations (e.g., geolocations derived from the server's observed IP addresses).[3] The credential system itself can then run policy-specific checks—like "verify that the user's last 5 logins have all occurred within a t-mile radius" for some threshold t or "verify that the last 5 browser fingerprints are relatively similar" using some distance measurement—without revealing the underlying data. Such tests can be used to detect anomalous location changes and prompt re-authentication or credential revocation.

- *Bot detection:* Social media platforms and ad networks often collect a range of behavioral signals to distinguish genuine user engagement from automated fraud including click fraud and "sockpuppeting". These can include which ads were viewed, what pages were visited, the order they were visited in, the time spent on each page, and browser fingerprints. With a CAAS, the logging of these behavioral signals is moved to the client. The credential accumulates these actions within its private state as the user navigates. To get credit for a click, the user does not reveal this raw history. Instead it asserts their accumulated signals satisfy the network's public fraud detection policy.

These applications, along with others, are enabled by a core set of programmable operations on the credential's private state. Rather than storing private data server-side, the client becomes a programmatically editable and cryptographically verifiable database. We build on recent work [59] for stateful anonymous credentials [32], combining zk-SNARKs over signed state commitments with a mechanism to prevent replays and forking. However, continuous anonymous authentication requires more than stateful anonymous credential. On an ongoing basis, the client must: 1) write authenticated values (measured by the server and third parties) to its state, 2) compute updates over this state, and 3) prove both that these values satisfy an authentication predicate and, optionally, produce a privacy-preserving measurement of their state.. For this paper, we assume the predicate test is public and leave the question of private (from the client) tests to future work.

Privacy-Preserving Measurement. A critical function of continuous authentication protocols is that they do not just enforce current policies, but that they allow defenders to analyze aggregate data in order to discover new attack patterns and evolve defenses. Making behavioral data fully private breaks this feedback loop. CAASs must address this by including a mechanism for privacy-preserving measurement, designed to serve as a partial replacement for server-side analytics cookies. Clients can generate reports on their private data (e.g., the distance between successive login locations) that are cryptographically bound to their credential usage. To ensure these reports do not compromise user anonymity, the reported values can be made private using techniques like differential privacy [34, 37].

Implementation Challenges. Realizing Continuous Anonymous Authentication poses a series of challenges. First, we must define a full end-to-end authentication process, supporting an ongoing authentication process between clients, servers, and third parties who measure behavioral signals. Second, we need to realize efficient examples of common applications.

Consider the problem of privacy-preserving measurement, where clients need to report on the distance between logins for honest users. Looking ahead, we choose RAPPOR [37] because of its simplicity and the fact that it does not use with complex distributions that are costly to work with in a zero-knowledge proof. Despite this simplification, we run into subtle problems generating randomness. To prevent biased reports, we need to ensure clients cannot pick any part of their data after they know the randomness, we cannot naively rely on server-provided values to produce randomness because of the risk of replay attacks from the server, and we need to minimize the number of interactions.

[2]The Wikimedia Foundation, for example, recently deployed an authenticated cookie encoding information including the date of cookie issuance and number of weeks that a user has been seen on the platform, which they intend to use for DDoS prevention [2].

[3]Here the server would still observe the client IP, but not log it. Looking ahead, we also cover the case where the IP address is hidden from the server and only visible to the entry node for, e.g., Tor or Apple Private Relay.

Implementing distance-based fraud checks also poses a challenge. Common calculations on latitude and longitude rely on floating-point operations, which are notoriously expensive to implement in circuits for zero-knowledge proofs. To build a practical system, we required an alternative approach. We solve this by mapping geographic coordinates to geohashes [9], a representation of geolocation that can be efficiently compared within the arithmetic circuits of a zk-SNARK. This allows for efficient proofs of proximity without resorting to costly floating-point logic.

Our Contributions. In this paper we propose a new type of anonymous credential, a Continuous Anonymous Authentication Scheme, designed to support programmable, privacy-preserving fraud and abuse prevention. We give an efficient construction for CAASs using zk-SNARKs [44] and prove it secure. We develop efficient new cryptographic techniques for stateful checks including geolocation proximity, browser fingerprint similarity, and verified content history. We also integrate a system for local differentially private reporting to enable server-side analytics. Finally, we build and test a prototype, zk-Cookies, demonstrating that our approach is practical. Our browser-based implementation can execute a credential update in under 1.5 seconds in client-side JavaScript, and a native implementation runs in under 200 ms (more details in Section 6 and in our example web app).[4] Finally, although we test our Continuous Anonymous Authentication implementation on typical laptops, we note that this entire system can be run on trusted hardware, which would provide further security guarantees.

2 Threat Model And Definitions

In this section, we discuss our threat model and setting, define the syntax of our scheme and its customizable application logic, and then introduce formal security definitions and discuss how they realize the threat model.

System model. There are three distinct parties within a Continuous Anonymous Authentication Scheme: clients who hold credentials; a server which manages the service, validates updates to client credentials, and verifies credential presentation; and measurement parties who measure aspects of the client that are used for updates.

As an example, consider an anonymous age verification system where the credential is augmented with additional data to detect credential theft. A client's credential would store both their date of birth and *locally* store IP addresses of each credential use. The client would use their credential both to prove their age to the server and attest that the IP address associated with their current credential use is within some geographic range of past uses.[5]

[4] https://zk-cookies.vercel.app/

[5] A failure to pass these geolocation checks does not necessarily mean the credential is stolen; it could prompt multi-factor authentication.

In our model, *clients* are trusted to maintain their own anonymity, but are not trusted to maintain the integrity of their own stateful anonymous credential. They may benefit from manipulating their credential or corrupting the measurement process (exceeding rate limits or avoiding detection of a stolen or shared credential).

Servers are trusted to maintain the integrity of the credentials that they issue to clients and correctly verify client proof validity, but are not trusted to maintain client anonymity. Similar to servers, they may benefit from breaking client anonymity (tracking for advertising purposes).

Measurement parties are trusted exclusively to conduct *accurate and unique* measurements of *already visible* client attributes for some scenario. For example, if a client connects to a server over Tor or Apply Private Relay, the entry node observes the client's IP and reports on it. Or, in other scenarios, the server may observe the IP address directly. Our scheme allows this observed IP to be stored in the credential without *additional* privacy leakage: even if the client sends its IP to the server for each use of the credential—e.g., if not used with Tor—the credential will not link those uses together.

This distinction is subtle, but important. In many real-world applications, clients do not use anonymity networks, but nonetheless may have policy or (in certain jurisdictions) legal protections from being tracked. In such a setting, the server learns the IP address of each client session, but is assumed not log or link together IP addresses across sessions. Our model allows clients to retain whatever level of privacy they achieve in this setting. We emphasize that our scheme supports both this model and a stronger one where clients connect over an anonymity service like Tor or Apple Private Relay, without additional leakage.

Responses are values the client sends to the server for aggregate statistics. zk-Cookies aims to serve as a partial replacement for server-side cookies, where a key function is analytics. For example, the server may want to collect the average distance between geolocated logins to dynamically set a threshold for detecting stolen or shared credentials. For integrity, these response values need to be authenticated as correct computations on authentic data. To ensure client privacy, we ensure these response are differentially private [34, 37].

For the sake of presentation clarity, the system model above is slightly simplified—in practice, one could have a separate *relying party* that verifies credentials independently of the server that manages and updates them.

Cryptographic syntax and definitions. The starting point for our definition is the definition of stateful anonymous credentials from Coul et al. [32]. This is the closest related work. We adapt the definition to a setting where we support arbitrary state transition functions, replacing Coul et al.'s "policy graph" with a tuple of labeled functions for state transitions encoding the application logic. We also assume that these algorithms perform all relevant communication over a secure and anonymous channel such as Tor or Apple Private Relay.[6]

For application logic, we define policies : $(\mathsf{InitialState}, \mathsf{UpdatePolicy}, \mathsf{ValidPolicy}, \mathsf{ResponsePolicy})$ to be a tuple of functions computable in probabilistic polynomial time which determine the valid operations on a stateful credential. Specifically: 1) $\mathsf{InitialState} : I \rightarrow F$ encodes how to produce a starting state based on initial user information I, 2) $\mathsf{UpdatePolicy} : (F, T_{\mathcal{M}}, T_{\mathcal{C}}) \rightarrow F$ determines how a credential is updated given an authenticated measurement (in $T_{\mathcal{M}}$) and new information from the client (in $T_{\mathcal{C}}$), 3) $\mathsf{ValidPolicy} : F \rightarrow \{0,1\}$ determines whether a credential is valid according to the current policy, and 4) $\mathsf{ResponsePolicy} : (F, \{0,1\}^*) \rightarrow \mathsf{A}$ produces anonymized analytics based on a credential's state and some randomness.

Definition. *A Continuous Anonymous Authentication Scheme is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{ObtainCred}, \mathsf{UpdateCred}, \mathsf{ProveCred})$ *representing protocols between a client $\mathcal{C}$, measuring party $\mathcal{M}$ and a credential provider $\mathcal{S}$ (the server):*

- $\mathsf{Setup}(\mathcal{C}(1^k), \mathcal{S}(1^k, \mathsf{fields}, \mathsf{policies}), \mathcal{M}(1^\lambda))$ *generates public parameters for all required cryptographic primitives in the scheme, such that they encode the fields* fields *and policies* policies.
- $\mathsf{ObtainCred}(\mathcal{C}(pk_{\mathcal{S}}, d, \mathsf{Cred}), \mathcal{S}(sk_{\mathcal{S}}, D))$*: $\mathcal{C}$ identifies themselves to $\mathcal{S}$ to obtain a credential* Cred *which is initialized to a valid starting state based on* policies *and initial user data d. The server also maintains a database D to ensure users in the system have unique keys.*
- $\mathsf{UpdateCred}(\mathcal{C}(pk_{\mathcal{S}}, I_{\mathcal{C}}, \mathsf{Cred}, \mathsf{Cred}'), \mathcal{S}(sk_{\mathcal{S}}, D, \mathsf{A}), \mathcal{M}())$*: $\mathcal{C}$ interacts with $\mathcal{M}$ to obtain a new measurement, and then $\mathcal{C}$ and $\mathcal{S}$ interact to update $\mathcal{C}$'s credential* Cred *to* Cred′ *in a way that is consistent with* policies*, the user's input $I_{\mathcal{C}}$, and the new measurement. This leaks no information about $\mathcal{C}$'s identity or state besides the analytics string* A. *The server also maintains a database D to prevent $\mathcal{C}$ from replaying prior credential states.*
- $\mathsf{ProveCred}(\mathcal{C}(pk_{\mathcal{S}}, \mathsf{Cred}), \mathcal{S}(pk_{\mathcal{S}}))$*: $\mathcal{C}$ proves posession of a valid credential according to* policies *to $\mathcal{S}$, or another relying party.*

Looking ahead, when presenting these protocols we will write them as algorithms for the client (suffixed with Req), and server (suffixed with Verify). (e.g., $\mathsf{UpdateCredReq} \leftrightarrow \mathsf{UpdateCredVerify}$). For simplicity, we will view measurement as a subprotocol to authenticate (e.g., sign) a measurement and a subroutine for verifying the measurement. This avoids the complexity of presenting three-party interactions in our concrete protocol and allows for a modular use of different types of measurement authentication.

Security definitions. As in [32], this scheme should satisfy *Provider Security* (that a malicious user should not be able to prove possession of a credential without obtaining the credential directly or via an allowed set of updates) and *User Security* (that a malicious server should not be able to learn information about users beyond what the scheme allows via the analytics string A). Here of course, the security is for a family of state-transition functions. We formalize this by defining an ideal functionality $\mathcal{F}_{\mathsf{caa}}$ for a Continuous Anonymous Authentication Scheme, which we define in Figure 1.

[6]Unless, of course the application specifically needs to track IPs.

Subtleties of the threat model and ideal functionality. Our threat model introduces a few subtleties, covered by our ideal functionality. First, we note that the ideal functionality allows each client to have multiple credentials (modeled by the second table index j)

Second, we need to ensure measured values, like IPs measured by a Tor or Apple Private Relay entry node, are authenticated (signed) and bound to a session. In particular, we need to ensure that a single measurement by an honest measuring party cannot be reused across multiple credential updates, or reused by multiple users, while maintaining unlinkability of a client's measurement requests. We model this, in the ideal functionality, as a single-use message from the measuring party that is available as part of a credential update.

Third, our ideal functionality models differentially private statistics—e.g., average geographic distance between logins across all clients—by providing randomness that cannot be biased by any involved party. In the ideal functionality, we also limit the client to computing responses over its previous credential state, to prevent the client from adaptively choosing its new state given the DP randomness (see Section 5.3).

A note on fingerprinting as a measurement. In one of our applications, we use browser fingerprinting, which is a common technique used by traditional websites to (nearly) uniquely identify users' browsers. Generally speaking, fingerprints are measured client-side, using obfuscated JavaScript, and reported to the server. As a result, their security is hard to model theoretically, and they rely on dubious notions of security through obscurity. However, fingerprinting schemes *are* in fact widely used across the web for client tracking and defending against bots, although they are heuristic. They offer a best-effort defense that is dependent on obfuscation and the ability to update fingerprinting techniques as attackers adapt.

How do we authenticate measured fingerprints given the heuristic nature of obfuscated code? We could assume that the obfuscated code contains a signing key, but this makes additional assumptions of the obfuscation. Instead, we assume that the obfuscated code has a tamper-resistant communication channel with the server or measuring party, which is closer to practice— in real deployments servers collect fingerprints — but more unusual cryptographically. In this model, the obfuscated code can send a commitment to the client's measured browser fingerprint to the server, and the client must show that they used a browser fingerprint consistent with this commitment when updating their credential.

$\mathcal{F}_{\mathsf{caa}}$

Setup :

Functionality initializes empty stateful credential table tbl and a broadcast channel, then fixes fields fields and policies $\mathsf{policies} = (\mathsf{InitialState}, \mathsf{UpdatePolicy}, \mathsf{ValidPolicy}, \mathsf{ResponsePolicy})$.

ObtainCred:

1. User $\mathcal{C}_i$ requests their jth credential, providing initial information d
2. If $\mathsf{tbl}[i][j] \neq \emptyset$ abort.
3. Otherwise, compute $\mathsf{Cred}_{init} := \mathsf{InitialState}(d)$ and set $\mathsf{tbl}[i][j] := \mathsf{Cred}_{init}$ in the credential table.
4. Broadcast "Registered" to all parties and $\mathcal{A}$.

UpdateCred:

1. Receive input $\mathsf{input}_{\mathcal{C}} = I_{\mathcal{C}} \in T_{\mathcal{C}}$ and credential number j from user $\mathcal{C}_i$.
2. Receive input (a measurement) $\mathsf{input}_{\mathcal{M}} = m \in T_{\mathcal{M}}$ from $\mathcal{M}$.
3. Retrieve $\mathsf{Cred} = \mathsf{tbl}[i][j]$ and abort if the credential is not in the table.
4. Sample $r \leftarrow\$ \{0,1\}^m$
5. Compute $\mathsf{A} := \mathsf{ResponsePolicy}(\mathsf{Cred}, r)$
6. Compute $\mathsf{Cred}' := \mathsf{UpdatePolicy}(\mathsf{Cred}, m, I_{\mathcal{C}})$ and update the credentials table as $\mathsf{tbl}[i][j] := \mathsf{Cred}'$.
7. Compute $b := \mathsf{ValidPolicy}(\mathsf{Cred})$ and abort if $b \neq 1$.
8. Send A to $\mathcal{S}$, and "Executed" to $\mathcal{A}$ and $\mathcal{C}_i$.

ProveCred:

1. Receive $\mathsf{input}_{\mathcal{C}}$ from user $\mathcal{C}_i$ and credential number j.
2. Retrieve $\mathsf{Cred} = \mathsf{tbl}[i][j]$, and abort if not found.
3. Compute $b := \mathsf{ValidPolicy}(\mathsf{Cred})$.
4. Send b to $\mathcal{S}$ and "Executed" to $\mathcal{A}$ and $\mathcal{C}_i$.

Figure 1: Simple ideal functionality $\mathcal{F}_{\mathsf{caa}}$. $\mathcal{A}$ is the adversary. This captures provider security, since users can only obtain a credential by ObtainCred and a series of UpdateCred operations based on valid measurements, and user security, since a malicious server only learns the outputs allowed by ResponsePolicy. Note that the ideal functionality does not require the to client give a differentially private report. It simply provides the appropriate randomness, and the security of the reporting function can be analyzed separately.

3 Background and Notation

Our constructions use the following standard cryptographic primitives. We use a signature scheme $(\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ that we assume to be existentially unforgeable under a chosen message attack (EUF-CMA). We use a secure pseudorandom function (PRF), denoted $\mathsf{PRF}(k,x)$, and a computationally hiding and binding commitment scheme, denoted $(\mathsf{Comm.Commit}, \mathsf{Comm.Open})$. Finally, we use a zk-SNARK $(\mathsf{SNARK.Setup}, \mathsf{SNARK.Prove}, \mathsf{SNARK.Verify})$ for arithmetic circuits over a field $\mathbb{F}$. We assume the zk-SNARK is complete, knowledge-sound, and zero-knowledge. We further require that the scheme supports simulation-extraction. The Groth16 proof system [13, 44, 57], used in our work, satisfies all these properties. We make use of accumulators with non-membership proofs [52]. Specifically, for a database D stored in an accumulator, we call the digest for its state $D.\mathsf{dig}$ and a non-membership proof $\overline{\pi}_D$.

We also use the concept of *nullifiers* [36,46]. Zerocash [16] uses replay nonces, which have come to be called nullifiers, that are commonly used to support state in anonymous systems based on zero-knowledge proofs. In Zerocash, users reveal a pseudorandom value (the nullifier) based on their private state when they spend coins. Transactions must have unique nullifiers to be valid. This nullifier uniqueness check ensures that users cannot spend the same coins twice, even though the coins being spent are kept private. In our construction, we use this same basic approach of revealing nullifiers to prevent users from repeating a credential state or registering multiple accounts with the same key.

3.1 Local differential privacy and RAPPOR

A differentially private (DP) mechanism can either be "central" (a dataset curator collects a sensitive dataset and calculates statistical queries using a randomized algorithm) or "local" (each data subject individually noises her data *prior* to sending it to a curator for aggregation). We utilize RAPPOR [37] to collect statistics from clients. [34, 48, 63] provide formal definitions of DP and local DP; we omit them for brevity.

The simplest example of a local DP algorithm for generating a privatized single bit is *randomized response*. If a data curator would like to collect a single bit of data d from respondents, each respondent can implement the randomized response algorithm:

```
1 : RandomizedResponse($d : \{0,1\}$) :
2 : $b_0 \leftarrow\$ \{0,1\}$
3 : if $b_0$ is 0, then return $d$
4 : Otherwise, sample $b_1 \leftarrow\$ \{0,1\}$
5 : return $b_1$
```

Looking ahead, we implement the RAPPOR mechanism [37], which generalizes 1-bit randomized response. It allows users to respond with sets of strings (using Bloom filters [19]), and is secure when users respond with repeated, correlated measurements over time. This makes it an appropriate algorithm to use for an anonymous service provider collecting analytics, like metrics related to geolocation and browser fingerprints, from anonymous clients.

RAPPOR takes the respondent's true value v, as well as the server-set parameters k, h, p, and q, and is executed locally by a respondent:

```
1 : BloomFilterRandomizedResponse($v$) :
2 : $B \leftarrow$ Hash $v$ onto Bloom filter of size $k$ using $h$ hash functions
3 : Create bit array $S$ of size $k$
4 : for $i = 1$ to $k$ do
5 :     Sample $S_i$ with $\Pr[S_i = 1] = p$ if $B_i = 1$ else $q$
6 : return $S$
```

In short, a respondent plays a randomized response game on the bits of a Bloom filter of encoding their value v to prevent individual user tracking based on the true value of B. The original implementation of RAPPOR does another randomized response game with parameter f on these bits to provide a longitudinal privacy guarantee; this is not applicable to our setting since clients are already anonymous.

3.2 Discrete Global Grid Systems and Geohashing

We briefly describe the Geohash standard [9], a type of Discrete Global Grid System (DGGS). A discrete global grid is a way to partition the surface of the Earth into a set of *cells*. A DGGS is a hierarchy of discrete global grids, such that there are multiple partitions depending on the size of cell desired.

Geohashing [9] is a particularly simple DGGS. A geohash is an (up to 60 bit) identifier where the even bits encode the longitude of a cell's location, and the odd bits encode the latitude of a cell's location. To encode a latitude or longitude coordinate in bits, the geohash format uses a "repeated halving" approach.

To encode the longitude of a cell into bits, geohashing sets the first even-indexed bit to 1 if the longitude is more than 0 degrees, and 0 otherwise. The second even bit is set to 1 if the longitude is in the upper half of the remaining range of bits, and 0 otherwise. This approach proceeds for the number of even bits in the geohash, where each bit halves the range of longitude that the geohash represents. This same approach is used to encode the latitude of a cell in the odd-indexed bits of a geohash, except for the starting range being -90 degrees to 90 degrees rather than 180 degrees. The bits of a geohash are often serialized in Base32 format for ease of reading. We give

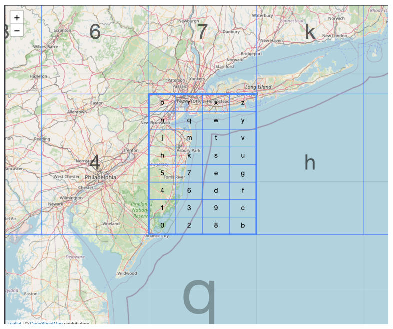


Figure 2: An example of the geohash cell dr5. Observe that the cell is split into 8 equal degree chunks length-wise, and 4 chunks width-wise because of 5 bit Base32 characters.

a visualization of the geohash cells surrounding New Jersey using the tool from [17] in Figure 2.

4 Continuous Anonymous Authentication Schemes

We now describe cryptographically instantiating a Continuous Anonymous Authentication Scheme. The scheme is parameterized by a set of functions $(\mathsf{InitialState}, \mathsf{UpdatePolicy}, \mathsf{ValidPolicy}, \mathsf{ResponsePolicy})$ that we will use to realize specific applications. We defer concrete examples of the functions $\mathsf{InitialState}, \mathsf{UpdatePolicy}, \mathsf{ValidPolicy}$, $\mathsf{ResponsePolicy}$ and discussing algorithmic design choices to the next section.

4.1 Cryptographic construction

The core of our construction is straightforward and uses standard techniques in the design of anonymous credentials built from zero-knowledge proofs. Clients compute SNARKs showing that they initialized, updated, or possess a credential in a given state. At each "update" step, the client computes a new commitment to their state based on some authenticated data, which the server signs after verifying a SNARK showing that the client performed a valid state update. To prevent a client from replaying old states, the client's proofs also produce *nullifiers*, which are pseudorandom strings generated by a PRF keyed by the user. Each time a client reveals a nullifier, the server checks that the nullifier has not been previously been revealed, which gives our system replay prevention in an anonymous way. In our pseudocode, all inputs to SNARK circuits are private unless otherwise stated. In our pseudocode, when working with a tuple like $\mathsf{Cred} = (k, s, \mathsf{counter})$, we access elements by their implied name (like $\mathsf{Cred}.k$) and by "unpacking" the tuple (like $(k, s, \mathsf{counter}) = \mathsf{Cred}$) depending on notational convenience. We defer discussion of how randomness is handled to Section 5.3.

Setup. As in the ideal functionality, the scheme requires a one-time setup for the primitives used. As part of Setup, the server $\mathcal{S}$ and the measuring party $\mathcal{M}$ are given a security parameter λ and specifications of $(\mathsf{InitialState}, \mathsf{UpdatePolicy}, \mathsf{ValidPolicy}, \mathsf{ResponsePolicy})$. In the setup phase, $\mathcal{S}$ and $\mathcal{M}$ generate key pairs $(\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}})$, $(\mathsf{pk}_{\mathcal{M}}, \mathsf{sk}_{\mathcal{M}})$ and broadcast them. $\mathcal{S}$ also computes $\mathsf{SNARK.Setup}$ for the circuits used in $\mathsf{ObtainCred}$, $\mathsf{UpdateCred}$, $\mathsf{ProveCred}$ and broadcasts pp for each circuit. $\mathcal{S}$ also initializes empty databases D_1, D_2 for storing PRF outputs (nullifiers). D_2 must support non-membership proofs and be able to provide digests of its state.[7]

ObtainCred. In $\mathsf{ObtainCred}$ (first column of Figure 4), the client produces a proof that they initialized their state correctly based on client-provided initial information d. The proof also produces a nullifier to ensure that multiple clients do not register with the same user key, and a commitment to the new state. The server checks the client's proof and nullifier, and signs a commitment to the client's initialized state if all checks pass. A client's state Cred is a tuple of the credential fields, a PRF key, and a counter (initialized to 1). The client uses a signed commitment to its state c to prove that their state is valid in subsequent interactions. By checking that $\mathsf{PRF}(k, 0)$ is not already present in D_1, $\mathcal{S}$ ensures that user keys are unique, which allows for our scheme to bind measurements to one specific user.

Measure. We split out the protocol for $\mathcal{C}_i$ obtaining an authenticated measurement from $\mathcal{M}$ (described Figure 3). Where $\mathcal{C}_i$'s credential has key k and counter counter, $\mathcal{C}_i$ computes a measurement binding tag $t = \mathsf{PRF}(k, \mathsf{counter})$, and sends t as part of a measurement request to $\mathcal{M}$. $\mathcal{M}$ signs t and the measurement. When using the measurement, $\mathcal{C}_i$ must show that $k, \mathsf{counter}$ in their credential satisfy $t = \mathsf{PRF}(k, \mathsf{counter})$. This is described in $\mathsf{VerifyMeasure}$. Because users have unique keys, and the credential's counter increments for each credential state, this binds each measurement to one user and credential state.

UpdateCred. In $\mathsf{UpdateCred}$ (second column of Figure 4), the client first requests a new measurement (an IP address, geolocation, and nonce for randomness generation in our system) using the $\mathsf{Measure}$ subprotocol. The client then produces a proof that they correctly updated their state via $\mathsf{UpdatePolicy}$ using these inputs. To verify that the state was updated correctly, the proof checks that the client's prior state is valid via a signature/commitment check, and by verifying that the measurement m is valid. The proof produces a commitment, a nullifier and an analytics string as public output. This is defined in $\mathsf{UpdateCredReq}$. For randomness generation, we assume $\mathcal{M}$ provides a new random nonce nonce in its response m. The server then verifies the client's proof/nullifier, and signs a commitment to the client's state if all these checks pass. We discuss in-circuit randomness generation for private analytics in greater detail in Section 5.3. Note that because of the incrementing counter, the values of $\mathsf{nullifier}_2$ are guaranteed to be different in each invocation of $\mathsf{UpdateCred}$.

[7]This can be implemented using a Merkle Patricia Trie, for example

```
Measure:
    1. $\mathcal{C}_i$: $t = \mathsf{PRF}(\mathsf{Cred}.k, \mathsf{Cred.counter})$.
    2. $\mathcal{C}_i \to \mathcal{M}$: t
    3. $\mathcal{M}$: Compute a measurement $m \in T_{\mathcal{M}}$ for $\mathcal{C}_i$ and
       $\sigma = \mathsf{Sig.Sign}(\mathsf{sk}_{\mathcal{M}}, (m,t))$.
    4. $\mathcal{M} \to \mathcal{C}_i$: $(m,t,\sigma)$
VerifyMeasure($\mathsf{Cred}, (m,t,\sigma)$):
    1. Check that $t = \mathsf{PRF}(\mathsf{Cred}.k, \mathsf{Cred.counter})$
    2. Check that $\mathsf{Sig.Verify}(\mathsf{pk}_{\mathcal{M}}, (m,t), \sigma) = 1$.
    3. If both checks pass, return 1. Otherwise, return 0.
```

Figure 3: Measure is a subprotocol for UpdateCred where $\mathcal{C}_i$ obtains an authenticated measurement. $\mathcal{C}_i$ has a credential Cred with a counter counter and a key k. $\mathcal{C}_i$ shows that a measurement is bound to their credential via VerifyMeasure.

ProveCred. In ProveCred, the prover shows that they have a credential in a state that is valid according to ValidPolicy without changing their credential's state. The specific policy encoded by our implementation of ProveCred is that the user is presenting a valid credential, it is their most recent valid credential, and they can show a credential arbitrarily many times. Furthermore, our design allows calls to ProveCred to be unlinkable. To ensure users do not use an old credential in ProveCred, we make sure the server periodically posts the database D_2 of used nullifiers for used credential states to a public bulletin board. We expect this database to be stored in a Merkle Patricia Trie [53] or another accumulator construction supporting non-membership proofs so users can show that the associated nullifier for their credential is not in the public list.

We show pseudocode for the interactions in ProveCred in Figure 4. In an interaction, a client generates a proof that they have a valid, signed credential according to ValidPolicy, and a verifier (not necessarily the issuing server) verifies the proof and that they provided a non-membership proof for a valid recent version of the database. We describe how to implement ProveCred with lower synchronization requirements using the technique of [26] in Appendix D.

4.2 Security argument

This construction can be shown to instantiate the ideal functionality $\mathcal{F}_{\mathsf{caa}}$ by using the underlying security properties of the PRF, commitment scheme, signature scheme, and zk-SNARK used. For space reasons, we defer a sketch of the security proof to Appendix F.

5 Continuous Anonymous Authentication features and implementations

In this section, we develop concrete example applications of zk-Cookies, inspired by common trust and safety problems on the web, like detecting account compromise or click fraud, and explain the algorithmic design choices necessary to securely and efficiently realize them.

5.1 Geohashing for efficient ZK geolocation

One common behavioral signal in online applications is changes in user location, where the server needs to test whether a user's current login location is within some distance of past logins. Moving this test to be run client-side requires the client to prove operations on location data inside of a zk-SNARK circuit. However, approaches like directly operating on the longitude/latitude coordinates of a user's location or using a complicated grid system like Uber's H3 (implemented in SNARKs in [38]) often require performing floating-point and trigonometric operations inside of a circuit, which is generally expensive and, looking ahead, potentially cost prohibitive for proofs generated in-browser.

Recall geohashing (Section 3.2), which partitions the surface of the earth into regular regions. It is commonly used in database applications, like Twitter/X's recommendation algorithm [65], for its efficiency. While geohashing's cell sizes and, consequently, distance computation accuracy vary with latitude, it offers a significant advantage for SNARKs. Specifically, it minimizes or even eliminates the need for in-circuit trigonometry or floating-point operations, making it potentially more SNARK-friendly than systems like H3 or raw longitude and latitude calculations. To leverage this efficiency, we must efficiently represent geohashes within a circuit and encode common location-based operations.

Geohashes can be represented in a single field element in many commonly used fields for SNARKs. Geohashing defines grid cells at different *precisions*. The bits in a geohash determines its precision (i.e. how many times the represented grid cell is split in half vertically or horizontally). The highest commonly supported resolution is 60 bits, which fits into a 64-bit word. Notably, this also fits into a single field element in many currently-used SNARKs. With careful design, we can realize associated operations, like calculating neighboring cells or using fixed-point arithmetic for converting between longitude-latitude pairs and geohashes efficiently in a circuit.

Given this representation, some location checks can be performed very cheaply with geohashes. Checking whether a higher-precision geohash cell is contained inside of a lower-precision geohash cell requires checking that the lower precision geohash's bits are a prefix of the higher precision geohash's. This approach is often used at scale in database systems because it allows for some location-related operations to be expressed as string operations. It is straightforward to

```
// Interactions for ObtainCred.
ObtainCredReq($d$):
  1. $k \leftarrow\!\!\$ \{0,1\}^\lambda$ $r \leftarrow\!\!\$ \{0,1\}^\lambda$
  2. Compute a proof $\pi$ with $k, r, d$ as private inputs showing:
     • $s = \mathsf{InitialState}(d)$
     • $\mathsf{Cred} = (k, s, 1)$
     • $c = \mathsf{Comm.Commit}(\mathsf{Cred}, r)$
     • $\mathsf{nullifier}_1 = \mathsf{PRF}(k, 0)$
     • The public outputs are $(c, \mathsf{nullifier}_1)$
  3. Send $\pi, c, \mathsf{nullifier}_1$ to the server $\mathcal{S}$.
ObtainCredVerify($\pi, c, \mathsf{nullifier}_1, \mathsf{pp}, D_1, \mathsf{sk}_{\mathcal{S}}$):
  1. Check that $\mathsf{SNARK.Verify}(\mathsf{pp}, \pi) = 1$.
  2. Check that $\mathsf{nullifier}_1 \notin D_1$ and add $\mathsf{nullifier}_1$ to $D_1$.
  3. Compute $\sigma' = \mathsf{Sig.Sign}(\mathsf{sk}_{\mathcal{S}}, c)$ and send $\sigma'$ to $\mathcal{C}_i$.
$\mathcal{C}_i$ stores state $\mathsf{Cred} = (k, s, 1)$ with auxiliary data $c, \sigma', r$.

// Function encapsulating the state transition and analytics logic in UpdateCred.
$\mathcal{F}_{\mathsf{Trans}}(\mathsf{Cred}, m, I_C)$:
  1. $(k, s, \mathsf{counter}) = \mathsf{Cred}$
  2. $s' = \mathsf{UpdatePolicy}(s, m, I_C)$
  3. Assert that $\mathsf{ValidPolicy}(s') = 1$.
  4. $r = \mathsf{PRF}(k, m.\mathsf{nonce} || \mathsf{counter})$
  5. $\mathsf{A} = \mathsf{ResponsePolicy}(s, r)$
  6. Return $(k, s', \mathsf{counter} + 1), \mathsf{A}$
```

```
// Interactions for UpdateCred.
UpdateCredReq($\mathsf{Cred}, c, r, \sigma, I_C, \mathsf{Meas}$):
  1. Sample $r' \leftarrow\!\!\$ \{0,1\}^\lambda$
  2. Compute a proof $\pi$ with private inputs $\mathsf{Cred}, c, r, \sigma, I_C, \mathsf{Meas}, r'$ showing:
     • $\mathsf{Sig.Verify}(\mathsf{pk}_{\mathcal{S}}, c, \sigma) = 1$
     • $\mathsf{Comm.Open}(c, \mathsf{Cred}, r) = 1$
     • $\mathsf{VerifyMeasure}(\mathsf{Cred}, \mathsf{Meas}) = 1$
     • $\mathsf{Cred}', \mathsf{A} = \mathcal{F}_{\mathsf{trans}}(\mathsf{Cred}, \mathsf{Meas}.m, I_C)$
     • $\mathsf{nullifier}_2 = \mathsf{PRF}(\mathsf{Cred}.k, \mathsf{Cred})$.
     • $c' = \mathsf{Comm.Commit}(\mathsf{Cred}', r')$
     • Public outputs are $(c', \mathsf{A}, \mathsf{nullifier}_2)$
  3. Send $(\pi, c', \mathsf{A}, \mathsf{nullifier}_2)$ to the state server $\mathcal{S}$.

UpdateCredVerify($\pi, c', \mathsf{A}, \mathsf{nullifier}_2, D_2, \mathsf{sk}_{\mathcal{S}}$):
  1. Check that $\mathsf{SNARK.Verify}(\mathsf{pp}, \pi) = 1$.
  2. Check that $\mathsf{nullifier}_2 \notin D_2$ and add $\mathsf{nullifier}_2$ to $D_2$.
  3. Store $\mathsf{A}$ for analytics.
  4. $\sigma' = \mathsf{Sig.Sign}(\mathsf{sk}_{\mathcal{S}}, c')$
  5. Send $\sigma'$ to $\mathcal{C}_i$.
$\mathcal{C}_i$ stores state $\mathsf{Cred} = (k, s', \mathsf{counter} + 1)$ with auxiliary data $c', \sigma', r'$.
```

```
// Interactions for ProveCred.
ProveCredReq($\pi, \mathsf{Cred}, c, r, \sigma, \overline{\pi}_{D_2}$):
  1. Compute a SNARK $\pi$ with private inputs $\mathsf{Cred}, c, r, \sigma$ and $\overline{\pi}_{D_2}$ (a non-membership proof for $D_2$) showing:
     • $\mathsf{Sig.Verify}(\mathsf{pk}_{\mathcal{S}}, c, \sigma) = 1$
     • $\mathsf{Comm.Open}(c, \mathsf{Cred}, r) = 1$
     • $\mathsf{nullifier} = \mathsf{PRF}(\mathsf{Cred}.k, \mathsf{Cred})$
     • Verify nullifier $\notin D_2$ via $\overline{\pi}_{D_2}$
     • $\mathsf{ValidPolicy}(\mathsf{Cred}.s) = 1$
     • The public output is $D_2$.dig, a digest for the state of $D_2$ used in $\overline{\pi}_{D_2}$.
  2. Send $\pi, D_2$.dig to the state server $\mathcal{S}$ (or another party being authenticated to).

ProveCredVerify($\pi, D_2.\mathsf{dig}, \mathsf{pp}$):
  1. Accept if $\mathsf{SNARK.Verify}(\mathsf{pp}, \pi) = 1$ and $D_2$.dig is a recent valid digest for $D_2$.
```

Figure 4: Interactions for ObtainCred, UpdateCred and ProveCred. Values labeled Cred represent credentials and are tuples $(k, s, \mathsf{counter})$ of key, credential state and counter. Values labeled M represent authenticated measurements from Measure and are labeled tuples (m, t, σ) of measurement, binding tag and signature. InitialState, UpdatePolicy, ResponsePolicy, ValidPolicy are functions parametrizing the scheme. In ObtainCred, the client runs ObtainCredReq with initial user data d, while the server verifies the operation with ObtainCredVerify. In UpdateCred, the client runs UpdateCredReq with its previous state, new user input I_C, and an authenticated measurement Meas obtained via Measure. In ProveCred, the client runs ProveCredReq to prove they have a valid credential, and the server (or other relying party) verifies this with ProveCred.

do in-circuit, requiring only a bit decomposition as opposed to complex floating point operations.

Computing the neighboring cell of a geohash or finding cells at other simple offsets can also be performed very efficiently. The bit representation of a geohash contains the latitude and longitude of a location as fixed-point integers, shifted so the values are always positive. The neighboring cells of a geohash have latitude and longitude values at some fixed offset from that cell's value. Computing a neighboring geohash involves "deinterleaving" the bits of the geohash to find the latitude and longitude values, adding an offset, and recombining the geohash. This is fairly cheap; we give the costs for implementing these operations in Section 6.

Converting latitude/longitude coordinates to and from geohashes can also be done efficiently. As specified in CTA-5009 [9], a geohash can be encoded/decoded by dividing the latitude and longitude of a location by a fixed value that depends on the precision. This fixed-precision computation can be encoded in fixed-point division, which is more efficient in SNARKs than floating-point division. A drawback of geohashes is that the relevant unit of distance is degrees of latitude and longitude rather than meters, which means that the actual distance between cells can vary depending on their latitude. Our system accepts this variance in the sizes of cells. This problem can also be solved by approximating trigonometric operations to vary the size of a geohash cell based on its latitude. We hardcode our example circuits to check if two geohashes share a prefix of 20 bits (a box of size 39.1 $\times$ 19.5 km at the equator, and a maximum allowed distance of about 44 km). The number of shared prefix bits is a parameter than can be tuned by the server.

5.2 RAPPOR implementation considerations

To realize anonymized random responses, we implement the RAPPOR protocol as described in Section 3.1. When implementing the RAPPOR protocol, we choose parameters to be consistent with Google's reference implementation [42]. Specifically, we use $h = 2$ hash functions and $k = 16$ bits for the Bloom filter parameters, and $p = \frac{1}{2}$ and $q = \frac{3}{4}$. However, we set $f = 0$ for the DP parameters. For demonstration purposes, we used 1 cohort.

We chose $f = 0$ (a non-standard parameter choice) due to the anonymous credentials context. Users' responses cannot be correlated to each other due to the anonymous credentials setting, and our implementation produces analytics that change between each response. This means there is no need to add longitudinal privacy (called a permanent randomized response) to user's responses by setting $f > 0$. The authors of [37] prove that RAPPOR is $h\log(\frac{q(1-p)}{p(1-q)})$-differentially private. Our settings yield a final DP guarantee of $\varepsilon \approx 0.954$.

To convert the pseudorandom bits output by a PRF to bits that are 1 with some probability, we grouped the PRF bits into blocks (8 bits at a time in our implementation, consistent with the Google reference implementation), and returned 0 or 1 depending on whether the bits interpreted as an integer were greater or less than some threshold.

5.3 Secure randomness for Local DP

One subtle detail of implementing DP analytics inside of a zk-SNARK is choosing the randomness used to randomize responses. The approach presented in this section is generic, and applies to other local DP algorithms (and ones in the shuffle model [20]) besides RAPPOR. Clients must not have control over the randomness used to ensure that they can not bias the analytics, while the server cannot have knowledge of the randomness used to ensure that the server cannot de-anonymize the analytics. We explain these issues, and our solution for generating the randomness to use for RAPPOR with the simple 1 bit randomized response protocol of Section 3.1 and by going through a series of "strawman" solutions.

First, we observe that clients cannot be trusted to produce their own randomness in our security model. If trusted to produce their own coin flips, a malicious client could simply claim that they sampled a 1 (and thus are responding with a random response) and then provide an arbitrary bit to the server collecting DP responses. This could be used to arbitrarily bias the responses collected by the server. A naïve approach to solving this problem would be computing a client's randomness in a "Fiat-Shamir" manner by hashing all relevant data in the SNARK circuit. This approach has problems with *grinding attacks*.

In our simple randomized response protocol, the probability of giving a random response is $\frac{1}{2}$, and the probability of doing the same in RAPPOR is similar. Thus, under the reasonable assumption that the user could control some inputs into the hash function for generating a random challenge, a malicious user would only have to try a few possible seeds to the hash function before finding a seed that generates a pseudorandom value that allows them to provide the desired randomized response. This is similar to output manipulation in [49]. For instance, when the randomized response probability is 0.5, a malicious user could expect to have to try two seeds before finding one that allows them to respond with a random bit, rather than their actual bit. Though the analysis is slightly more complicated, a user would not have to sample many possible seeds before finding a random input to RAPPOR that would allow them to report a result that does not contain their actual information.

Second, the server alone cannot be trusted to dictate the randomness used for randomized response to clients (in our architecture, the server could dictate the randomness to use in the next randomized response when responding with a signed state commitment). Again, consider the situation where the server assigns random bits 01 to the client in the simple randomized response protocol. Since the server is aware that they provided the random bits 01 to the client, they know that the client responded honestly in the randomized response protocol, thus leaking the client's actual data.[8][9]

One natural solution for producing randomness that is *grinding resistant* and secret to the server is for the client to store a key k as part of their state. For each application of randomized response, the client computes their randomness as $\mathsf{PRF}(k,n)$ where n is a server-provided nonce. But here the server can repeat a previous nonce provided to the same client, leading to repeated randomness and likely an information leak. To prevent this adversarial server behavior, we have the client store a state counter counter, and compute their randomness as $\mathsf{PRF}(k,n||\mathsf{counter})$. This approach resembles a standard 2-party coin flipping protocol, but, unlike [41], forgoes the need for an extra interaction per response.

In more detail, during ObtainCred, the client sends a commitment to a random PRF key k, and the client's state counter is initialized to 1. UpdateCred guarantees that the client's key and counter are continuously stored in the credential. During each execution of UpdateCred, the server (or measurement party) provides a signed nonce n, and the client computes $\mathsf{PRF}(k,n||\mathsf{counter})$ to generate randomness. In our protocol, by attaching the nonce to the measuring party's response, we do not incur an additional round of communication.

One remaining subtlety in realizing the ideal functionality of our system is ensuring that clients cannot adaptively bias their input data because they already know the randomness that will be used for their randomized response. We have UpdateCred compute analytics on the previous state of a user's credential, rather than the updated state. This approach has the effect of a user committing to the values that they will respond with before producing a randomized response. Thanks to the standard design approach used in our scheme (that users frequently compute commitments to their state), we have users commit to their response values before producing a randomized response at no additional cost—in particular, we do not need an extra round round of communication to commit to inputs—though collection of responses is delayed by one round of UpdateCred.

This set of techniques may also be useful for other contexts where secure unbiased randomness is required inside of a zk-SNARK. Differential privacy is a different application from many use cases for randomized algorithms in zk-SNARKs, like permutation checking or RAM techniques [68], where their random challenges can be biased by grinding, but this is acceptable because the algorithms have low enough failure probabilities that a polynomial-time adversary cannot meaningfully change the output of the algorithm. Algorithms like randomized response rely on sampling bits from a uniform distribution, which is a different setting from traditional randomized algorithms in SNARKs.

[8]In an anonymous credential setting, the server may not be able to link randomness provided at, e.g., credential issuance, to a specific client's subsequent response, but they could still supply the same randomness to all clients to learn clients' actual responses.

[9]These issues also apply to approaches where the server has partial information about the randomness a client will used for a randomized response.

5.4 Fuzzy fingerprint hashing in zk-SNARKs

Browser fingerprinting is a commonly used technique in practice for identifying transient or ephemeral users of a web service, wherein an internet service provider records detailed browser information about clients to track their browsing behavior [23, 35]. These fingerprints are commonly used for targeted advertising and product suggestions, but are also commonly used in commercial applications for fraud and abuse prevention [24, 54, 56]. We are particularly interested in them for their ability to identify spammy, fraudulent, or otherwise atypical browsing behaviors—for example, a user logging in from the Brave browser on a Linux machine when they normally log in using Safari on a Mac [67].

Unfortunately, perhaps to avoid aiding attackers, is little public discussion of details of these techniques, nor a readily available standard open-source implementation. Inspired by the CreepJS project's fuzzy fingerprint functionality [1] (and finding a dearth of functional open-source fingerprinting software), we implemented our own locality sensitive hash function for browser fingerprints in JavaScript. We base our fuzzy hash implementation on the MinHash [22] and SimHash [29] algorithms, though in principle any function that approximates a nearest neighbor search will do. To implement our algorithm, the browser first collects a variety of data points from the client's device, including browser; screen size; time zone data; and WebGL, audio, and canvas fingerprints. Then, all collected data is converted to a JSON string, ordered alphabetically by key for consistency, and hashed with the djb2 hash algorithm so that each data point becomes a 32-bit integer value. Next, we assign each data point a weight (based on the feature entropy). We then create a 500-bit output feature vector. We iterate through each data point and each bit of the output feature vector. If the inspected bit of the data point is 1, we add weight to that index of the output feature vector; otherwise we subtract weight from that index of the output feature vector. Next, we normalize the output feature vector based on the largest absolute value (so each index is a real-valued number between -1 and 1). Finally, we cast indices with negative values to 0 and indices with positive values to 1, yielding a binary bit string that serves as our fuzzy hash.

To compare two bit string fingerprints $\mathsf{F}_1, \mathsf{F}_2$ with $|\mathsf{F}_1| = |\mathsf{F}_2| = n$, we calculate the Hamming distance between the two strings and divide it by the length of the string ($\frac{\mathsf{XOR}(\mathsf{F}_1,\mathsf{F}_2)}{n}$), yielding a similarity score between 0 and 1. This similarity score can be tuned by the anonymous service provider to set thresholds on the minimum allowable fingerprint similarity between a given request and the fingerprint history. We chose 500-bit hashes because this almost fully uses 2 field elements in the SNARK we used (Groth16 over the BLS12-381 curve).

In the context of our implementations of Continuous

Anonymous Authentication Schemes, we utilize fuzzy fingerprints by implementing a client-side script which collects a fuzzy fingerprint and reports it as a hash-based commitment for the server to sign. Inside the credential, we have a circuit for comparing the similarity between current fingerprint and the previous valid fingerprint. In a real deployment of a fingerprinting system like this, the JavaScript for measuring fingerprints would likely be obfuscated. Because our system design enables measurements from non-server third parties (e.g. Tor nodes), we could make this process more cryptographically secure by having obfuscated JavaScript either (a) sign the measurement itself or (b) return a commitment to the measurement that the server then signs. We did not implement this for our prototype, as it does not change the core zk-SNARK functionality.

5.5 Geolocation streaming in zk-SNARKS

Keeping a list of previously valid client geolocations and checking the client's current location against them is the simplest solution for checking location similarity. However, the size of the circuits for these checks in a Continuous Anonymous Authentication scheme increases linearly with the size of the history stored, and this potentially leads to efficiency problems when storing long histories. To demonstrate how a Continuous Anonymous Authentication implementer might maintain similar functionality with a constant constraint count, we prototype a naive geolocation "streaming" algorithm that is evaluated in the circuit without requiring the Continuous Anonymous Authentication Scheme to store location history.

We define $(La_1, Lo_1), (La_2, Lo_2), \ldots, (La_n, Lo_n)$ to be the sequence of latitude and longitude values observed so far. $(S^{La}_{n-1}, S^{Lo}_{n-1})$ are the previous latitude and longitude sums ($S^{La}_{n-1} = \sum_{i=1}^{n-1} La_i$, same for S^{Lo}_{n-1}). $(A^{La}_{n-1}, A^{Lo}_{n-1})$ denote the average latitude and longitude. When a new geolocation (La_n, Lo_n) is added, we update the sums and the averages:

$$(S^{La}_n, S^{Lo}_n) = (S^{La}_{n-1} + La_n, S^{Lo}_{n-1} + Lo_n)$$
$$(A^{La}_n, A^{Lo}_n) = \left(\left\lfloor \frac{S^{La}_n}{n} \right\rfloor, \left\lfloor \frac{S^{Lo}_n}{n} \right\rfloor\right)$$

A^{Lat}_n and A^{Lo}_n are computed using standard techniques for integer division in SNARK circuits. Similarly to the non-streaming version of the algorithm, we can ensure that the new geolocation (La_n, Lo_n) is close to the running average by converting the average position and current locations both into geohashes and comparing their first k bits.

5.6 Ad attribution and verified browser history checks

Ad attribution (part of online advertising infrastructure where a server checks whether a client has seen a given advertisement when the client performs an action like buying a product [18]) is a key feature of commerce on the internet. This is a clear area of interest; a variety of efforts have been made over the past few years to make ad attribution more privacy-preserving [6, 8, 12]. Verified browser history checks refer to when a client takes a risky action (for example, attempting to reset a password in a user's admin panel), and a server may want to be able to verify that the client has navigated through a given preceding set of pages (e.g., "Settings" → "Security" → "Reset password"). This is a similar mechanism to partially blind signatures [58] and allowed referrer lists [24].

We can similarly model both of these desired features for implementation in a zk-SNARK. Say that we have a sequence of pages or advertisement IDs stored in the credential history $(P_1, P_2, \ldots, P_n)$. For all $i \in [n]$, $P_i = \texttt{hash(URL)}$. Hashing the strings allows us to store each arbitrary length URL as a single field element, improving circuit efficiency.[10] Given this stored data, we model ad attribution as the client receiving an advertising ID (or a list of IDs) from the server and iterating through the attribution history stored in its credential, checking to see if any of the stored advertising IDs match the server's provided ID(s). Similarly, we implement verified browser history checks as the server providing a sequence of k page IDs and the client searching its history for that sequence (or even just considering the last k entries of the history).

5.7 Legacy credential checks in ObtainCred

We also consider parameterizing InitialState with other checks. Concretely, we consider having InitialState verify a signature on a user's passport and validate the user's age. This simulates credential systems like Google's [11], where credential issuance requires a user to have a valid government issued credential. In the context of Persistent Anonymous Credentials, expensive checks like verifying a legacy signature on a passport can be performed once per user when doing ObtainCred, after which the validity of the passport and relevant fields can be stored in the credential for further uses.

6 Evaluation

In this section, we evaluate zk-Cookies, a prototype implementation of a Continuous Anonymous Authentication Scheme.

6.1 Prototype Implementation

For our prototype of zk-Cookies, we fix a specific application—an anonymous credential for age verification—and implement several scenarios on top of it that explore the design space. In each scenario, the credential stores the user's year of birth (verified via digitally signed passport data in

[10]In many uses, where the web server knows the ad or page currently being viewed, the initial hash can safely be computed outside the circuit as an authenticated input.

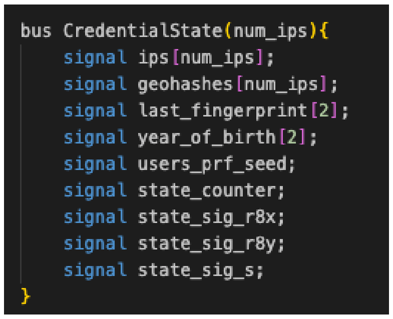

```
bus CredentialState(num_ips){
    signal ips[num_ips];
    signal geohashes[num_ips];
    signal last_fingerprint[2];
    signal year_of_birth[2];
    signal users_prf_seed;
    signal state_counter;
    signal state_sig_r8x;
    signal state_sig_r8y;
    signal state_sig_s;
}
```

Figure 5: The Circom bus with the fields in our credential's state. `state_sig_{r8x,r8y,s}` are components of an Ed-DSA signature on the state.

ObtainCred) and additional application-specific history data, e.g., geolocation history, ad attribution history, etc.

Each scenario follows the same pattern: a user proves they meet a specific policy (e.g., they are over 18), and simultaneously updates their credential with a new measurement and reports a DP value to the server. Specifically, we implement:

- A "basic" version that implements geohash and fingerprint closeness checks to detect credential theft/sharing, parameterized by geolocation and IP history length.
- A "streaming" version that maintains a single user average geohash instead of maintaining a history list, offering increased performance but more limited fraud checks.
- An "attribution history" variant of the basic version, which implements additional checks for the presence of a single page history UUID (simulating ZK ad attribution) or a set of consecutive page history UUIDs (simulating verified browser history checks), parameterized by attribution history size. We hold the credential history length constant at five IPs and geohashes.

We implement and test our prototypes in the Circom programming language for zk-SNARKs, as ∼1,500 lines of Circom code. For testing, we used an optimized prover and verifier called RapidSnark [10]; for online demonstration purposes, we use Circom's built-in WASM compilation.

Figure 5 shows a Circom struct with the fields used in the "basic" prototype which stores recent IPs and geolocations, the last browser fingerprint, year of birth (extracted from a passport presented during registration), a seed for a PRF, a counter (used for nullifiers and generating pseudorandom values for randomized response), and a state signature. In Figure 7 in Appendix E, we give pseudocode for the functions (InitialState, UpdatePolicy, ValidPolicy, ResponsePolicy) for the basic application to show the connection between our ideal functionality and implemented application.

6.2 Experiments

In our scheme, the primary computational cost is client-side proving time, which is a function of circuit size. In contrast, server-side verification is a fixed cost because Groth16 has constant verification time in our setting.

Methodology. For each circuit configuration, we ran 100 trials of the prove and verify portions of the circuit and recorded the average proving and verifying times (we include 95% confidence intervals in our figures but they are too small to be seen in some cases). Proofs and verifications were run using RapidSnark. Our experiments do not measure the time to send proofs over the network. However, since Groth16 proofs are constant size, as are our commitments, message sizes were less than 1 kB for all our tested scenarios.

Hardware and Testing Environment. All trials were run on a 2019 Macbook Pro running Sequoia 15.6, with a 2.4 GHz, eight-core Intel i9 CPU and 32GB of RAM. We used Node.js version 23.11.0 and RapidSnark version 0.0.7, which were current at the time of testing.

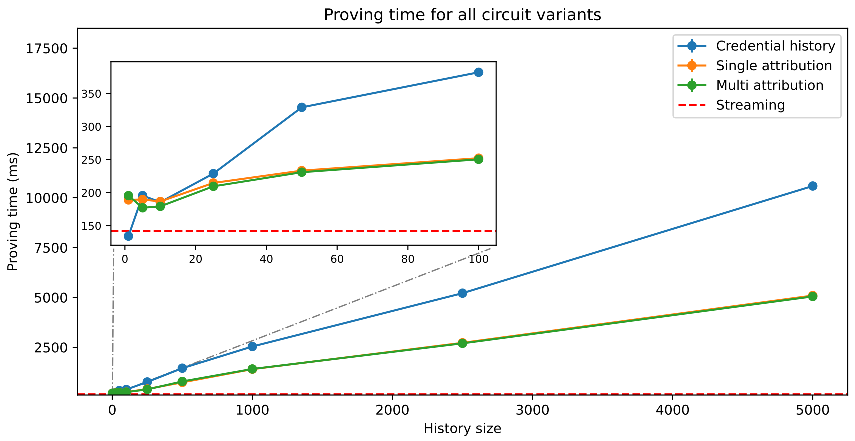
6.3 Results

Proving times are given in Figure 6a. The corresponding circuit sizes are in Table 1 in Appendix C. We stopped testing at a credential history size of 5,000 because the Circom JavaScript compiler started to run out of memory beyond this history size.[11] This is not a performance limit on the client—better tooling would let us increase our credential history length.

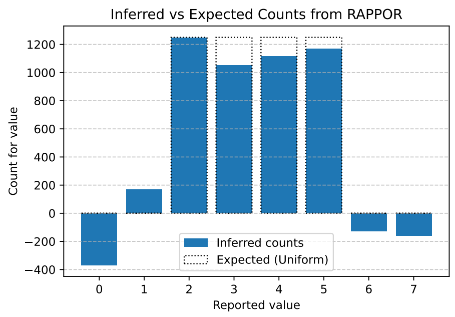
All circuits with a credential history size under 100 run in under 400 ms, on average. This includes the streaming algorithm (141.92 ms [95% CI ± 1.00 ms]), which performs distance checks based on a running average and can in principle take in an arbitrary number of past geolocations. All attribution history circuits with a history size under 100 run in under 250 ms, on average. For shorter history lengths, the number of constraints per history value saved to the credential is dominated by the "fixed overheads" from verifying signatures on the client's previous state and the new measurement to add to the credential. Thus, proving times for short history lengths are similar. After that, proving time increases linearly. A credential history length of 50, for example, yields an average proving time of 329.12 ms (95% CI ± 5.57 ms).

In-browser performance. We manually tested performance of a Continuous Anonymous Authentication circuit published as a web app in the browser. We accessed the web app using a browser running Chromium version 137, and the circuit itself was running in WebAssembly (the "basic" circuit with a credential history length of five). Across 20 trials, the proving time averaged 1473.24 ms (95% CI ± 313.97 ms). This does not use RapidSnark, resulting in slower performance.

[11] Since RapidSnark uses the same frontend, this applies to it as well.



(a) Proving times for all circuit variant types (basic and streaming, multi attribution, and single attribution), parameterized by credential history size. The streaming scheme is faster (plotted as a horizontal line) than all circuits other than that with a history size of one.



(b) Histogram with the results of simple RAPPOR decoding code for 5000 reports on a simple uniform distribution.

Figure 6: Proving times (left) and RAPPOR decoding histogram (right).

Verification performance. We also benchmarked the server-side verification operations (including nullifier checks and signing newly validated outputs). We implemented a basic server in Go using LevelDB. In this (toy) setting, verification of a circuit with history length five takes 6.02 ms (95% CI ± 0.15 ms), checking for nullifiers in a server database takes 7.86 μs (95% CI ± 5.82 μs), and signing the output takes 125.18 μs (95% CI ± 80.00 μs). Server-side performance for nullifier checks, however, depends on the actual database and cloud architecture. We tested against LevelDB, a persistent database for single-server applications

ObtainCred performance. We implemented a circuit for ObtainCred that produced a commitment to an initial state, as well as verifying a passport to extract a year of birth for the credential. Since the focus of this work is not validating legacy signatures, we used the existing implementation from self.xyz [62], which implements Circom circuits for this purpose. Across 100 trials, the ObtainCred circuit has a mean run time of 3,358.19 ms (95% CI ± 13.49 ms). This code is run once per user enrollment and we note that recent work offers new proof techniques with sub-second performance on legacy mobile devices [39].

Randomized Response. To verify that meaningful analytics could be extracted from our RAPPOR implementation, we implemented a simplified version of the decoding algorithm from RAPPOR [37] for determining the true distribution of values represented by a set of reports. In Figure 6b, we show the results of determining the distribution represented by 5,000 RAPPOR responses generated by our RAPPOR circuit, with encoded values drawn from a uniform distribution. The accuracy of this histogram is expected to improve with more data points and a more advanced approach to decoding.

7 Related work

Here we discuss related work on anonymous credential schemes and account security.

Anonymous Credentials. There has been a large amount of work on anonymous credentials. [14, 15, 25, 27, 28, 31, 40, 57, 61]. The most directly relevant here is Stateful Anonymous Credentials [32], which provides a very similar construction to our scheme, albeit for a different application, with different definitional requirements, and without the benefits of modern zero-knowledge proofs. To the best of our knowledge, there has been little follow-up on this scheme, though work on Decentralized Anonymous Payments [16] and Private Smart Contracts [21, 50] for blockchains offer an analogous form of privacy-preserving state manipulation (in a different context).

Credentials for fraud prevention. For credentials designed explicitly for fraud prevention, there is, of course, Privacy Pass [33]. Facebook has proposed Blind Tokens for fraud prevention [47]. The work in [51] introduced private tokens with hidden metadata—a concept now in commercial usage—with subsequent work including [30, 60]. The goal of these schemes is to avoid a pitfall of Privacy Pass. In Privacy Pass, when server-side logic detects a client is fraudulent, the client is not issued a token and can adapt to avoid future detection. In private metadata bit constructions, the client is still issued a token, but it contains a single bit that the client cannot observe which marks the client as fraudulent.

Other related works. [20] is a concurrent work that presents a similar scheme to ours for producing trusted randomness for local DP in zero-knowledge proofs. The DP portion of our protocol generalizes theirs. [20] assumes that the measurements being reported in the local DP scheme come from trusted hardware, while our protocol supports measurements from untrusted users. Our protocol does not assume the presence of timestamps and we bind measurements to specific users differently.

8 Conclusion

In this work, we introduced Continuous Anonymous Authentication, a framework designed to shift sensitive user data collection from servers to clients while preserving privacy. Continuous Anonymous Authentication Schemes enable a variety of on-device security and functionality checks that have traditionally required extensive server-side data collection, from validating location history to verifying ad attribution.

Our approach is modular and can be instantiated with different cryptographic primitives. It can, for instance, readily incorporate improved zk-SNARK schemes, directly benefiting from the consistent performance gains the field has seen over the last decade.Similarly, this approach can readily be instantiated with post-quantum primtives.

Looking ahead, technologies like ZKVMs—which execute assembly from standard languages such as Rust—represent a promising path toward greater programmability. This would empower developers and system administrators to dynamically adapt security policies as threats evolve. A compelling alternative development avenue is to replace zero-knowledge proofs entirely with local trusted hardware operating in an analogous role [64]. This would result in a lightweight and highly efficient protocol a that mobile platform owners could easily deploy. Crucially, client privacy would not depend on TEE security.

Acknowledgments This paper would not be where it is today without a number of helpful comments in refining the concept. Thanks to Bobby Bhattacharjee for originally pointing out we could store IP addresses in the credential and kicking off the work. Thanks to Matt Green for his comments; in particular those insisting this was not simply a stateful anonymous credential and his ironic but accurate insistence that "no one was going to read *your* previous stateful anonymous credential paper" [32] (it was his stateful anonymous credential paper, not ours, and one you should read). Thanks to Christina Garman and David Adrian for helpful comments, and to Tudor Dumitras for suggesting the name in a conversation over the espresso machine he generously made available to the security group.

References

[1] CreepJS. https://abrahamjuliot.github.io/creepjs/.

[2] Edge uniques - Wikitech. https://wikitech.wikimedia.org/wiki/Edge_uniques.

[3] Georgia SB351. https://www.legis.ga.gov/legislation/66023.

[4] Louisiana Laws - Louisiana State Legislature. https://legis.la.gov/legis/Law.aspx?d=1337818.

[5] Nebraska LB383. https://nebraskalegislature.gov/bills/view_bill.php?DocumentID=59569&docnum=LB383&leg=109.

[6] Overview of Attribution Reporting API. https://privacysandbox.google.com/private-advertising/attribution-reporting.

[7] UK Online Safety Act. https://www.legislation.gov.uk/ukpga/2023/50.

[8] Privacy Preserving Ad Click Attribution For the Web, May 2019. Section: Privacy.

[9] Cta-5009-a: Fast and readable geographical hashing, June 2024.

[10] iden3/rapidsnark, August 2025. original-date: 2023-11-23T09:09:39Z.

[11] It's now easier to prove age and identity with Google Wallet. https://blog.google/products/google-pay/google-wallet-age-identity-verifications/, April 2025.

[12] Ralph Ankele, Sofia Celi, Ralph Giles, and Hamed Haddadi. The Boomerang protocol: A Decentralised Privacy-Preserving Verifiable Incentive Protocol, October 2024. arXiv:2401.01353 [cs].

[13] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-SNARK. Cryptology ePrint Archive, Paper 2020/811, 2020.

[14] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous Credentials Light, 2012. Publication info: Published elsewhere. Unknown where it was published.

[15] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and Noninteractive Anonymous Credentials. In Ran Canetti, editor, *Theory of Cryptography*, pages 356–374, Berlin, Heidelberg, 2008. Springer.

[16] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

[17] Alexander Berezovsky. Geohash Explorer — geohash.softeng.co. `https://geohash.softeng.co/`.

[18] Ron Berman. Beyond the Last Touch: Attribution in Online Advertising. *Marketing Science*, 37(5):771–792, September 2018. Publisher: INFORMS.

[19] Burton Bloom. Space/time trade-offs in hash coding with allowable errors | Communications of the ACM, July 1970.

[20] Tariq Bontekoe, Hassan Jameel Asghar, and Fatih Turkmen. Efficient verifiable differential privacy with input authenticity in the local and shuffle model. PETS 2025, 2024.

[21] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

[22] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 327–336, New York, NY, USA, May 1998. Association for Computing Machinery.

[23] Tomasz Bujlow, Valentín Carela-Español, Josep Solé-Pareta, and Pere Barlet-Ros. A Survey on Web Tracking: Mechanisms, Implications, and Defenses. *Proceedings of the IEEE*, 105(8):1476–1510, August 2017.

[24] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Comput. Surv.*, 50(1):13:1–13:34, March 2017.

[25] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable & Modular Anonymous Credentials: Definitions and Practical Constructions, 2015. Publication info: Preprint. MINOR revision.

[26] 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.

[27] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 268–289, Berlin, Heidelberg, 2003. Springer.

[28] Jan Camenisch and Anna Lysyanskaya. Signature Schemes and Anonymous Credentials from Bilinear Maps. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Matt Franklin, editors, *Advances in Cryptology – CRYPTO 2004*, volume 3152, pages 56–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.

[29] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388, New York, NY, USA, 2002. Association for Computing Machinery.

[30] Melissa Chase, F. Betül Durak, and Serge Vaudenay. Anonymous tokens with stronger metadata bit hiding from algebraic MACs. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 418–449. Springer, Cham, August 2023.

[31] David Chaum. Security without identification: transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, October 1985.

[32] Scott E. Coull, Matthew Green, and Susan Hohenberger. Access controls for oblivious and anonymous systems. *ACM Trans. Inf. Syst. Secur.*, 14(1), June 2011.

[33] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3):164–180, July 2018.

[34] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3-4):211–407, 2013.

[35] Peter Eckersley. How Unique Is Your Web Browser? In Mikhail J. Atallah and Nicholas J. Hopper, editors, *Privacy Enhancing Technologies*, pages 1–18, Berlin, Heidelberg, 2010. Springer.

[36] Electric Coin Company / Zcash Foundation. Nullifiers — the orchard book, 2025.

[37] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1054–1067, New York, NY, USA, November 2014. Association for Computing Machinery.

[38] Jens Ernstberger, Chengru Zhang, Luca Ciprian, Philipp Jovanovic, and Sebastian Steinhorst. Zero-knowledge location privacy via accurate floating-point SNARKs. Cryptology ePrint Archive, Paper 2024/1842, 2024.

[39] Matteo Frigo and abhi shelat. Anonymous credentials from ECDSA. Cryptology ePrint Archive, Paper 2024/2010, 2024.

[40] Christina Garman, Matthew Green, and Ian Miers. Decentralized Anonymous Credentials, 2013. Publication info: Preprint.

[41] Gonzalo Munilla Garrido, Matthias Babel, and Johannes Sedlmeir. Towards Verifiable Differentially-Private Polling, June 2022. arXiv:2206.07220 [cs].

[42] Google. Rappor: Privacy-preserving reporting algorithms. https://github.com/google/rappor, 2014.

[43] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019.

[44] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016.

[45] Michael Hoffman. House Bill 3: Florida residents will have to verify their age to access adult sites starting Jan. 1, 2025, December 2024. Section: State.

[46] Daira Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification, Version 2023.4.0 [NU5].

[47] Subodh Iyengar. siyengar/private-fraud-prevention, March 2025. original-date: 2019-08-06T20:08:52Z.

[48] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. What Can We Learn Privately?, February 2010. arXiv:0803.0924 [cs].

[49] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. Preventing Manipulation Attack in Local Differential Privacy using Verifiable Randomization Mechanism, June 2021. arXiv:2104.06569 [cs].

[50] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 839–858, 2016.

[51] Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous tokens with private metadata bit. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 308–336. Springer, Cham, August 2020.

[52] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[53] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. Cryptology ePrint Archive, Paper 2014/1004, 2014.

[54] Theresa Nguyen. What is Browser Fingerprinting? https://www.experian.com/blogs/insights/browser-fingerprinting/, November 2024.

[55] OWASP. Credential Stuffing Prevention - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html#device-fingerprinting.

[56] Plaid. How device fingerprinting improves fraud prevention. https://plaid.com/resources/identity/device-fingerprinting/.

[57] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 790–808, 2023.

[58] Ben Savage and Subodh Iyengar. Fighting fraud using partially blind signatures, October 2019.

[59] Maurice Shih, Michael Rosenberg, Hari Kailad, and Ian Miers. zk-promises: Anonymous Moderation, Reputation, and Blocking from Anonymous Credentials with Callbacks, 2024. Publication info: Published elsewhere. Major revision. Usenix Security 2025.

[60] Tjerand Silde and Martin Strand. Anonymous tokens with public metadata and applications to private contact tracing. In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 179–199. Springer, Cham, May 2022.

[61] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers, March 2020. arXiv:1802.07344 [cs].

[62] Florent Tarvenier. self: Identity wallet using nfc-based passport authentication and zk-snark proofs. https://github.com/selfxyz/self, 2025.

[63] Apple Differential Privacy Team. Learning With Privacy At Scale, December 2017.

[64] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 19–34, April 2017.

[65] Twitter. The algorithm: Open-source twitter recommendation algorithm. https://github.com/twitter/the-algorithm, 2023. Accessed: 2025-05-07.

[66] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve. Technical report, iden3 / Ethereum Foundation / Universitat Pompeu Fabra, 2019.

[67] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martın Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications.

[68] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. Cryptology ePrint Archive, Paper 2017/1145, 2017.

A Ethical considerations

This research develops cryptographic constructions and, as such, side-steps the standard ethical issues involving human subjects, personal data, or vulnerability disclosure outlined in the Menlo report.

But the means do not justify the ends in terms of ethics: just because research was done with ethical steps in a process does not mean it is ethical. This is, unfortunately, under-addressed by the current ethics guidelines for Usenix Security. The biggest question for zk-Cookies is how future implementations may be used. Our sample application, age verification, is itself controversial. Even with privacy issues addressed, there are questions of the appropriateness of controlling access to content on the internet, the scope of content covered by those controls, and the very real risk these tools may be used by oppressive regimes (or future governments of democratic countries) for malicious purposes. These questions are not unique to age verification. Access control is, after all, a form of control, whether it is cryptographically secure and private or not.

In lieu of a further discussion, we simply state there are applications of zk-Cookies that are beneficial and ones that are not and utility must be evaluated independently of a system's cryptographic properties; just because a construction is secure and private, does not make it a good idea. And just because a system provides cryptographic security guarantees now does not mean that its algorithms and software cannot be later modified to remove those protections. Removing privacy protections, for example for identity on the internet, may be considerably easier than forcing the deployment of a completely non-privacy preserving system to start.

B Open science

Code for this paper is available at: https://github.com/htried/zkp-ldp. As mentioned previously, we deployed a web app with zk-Cookies, our implementation of a Continuous Anonymous Authentication Scheme at: https://zk-cookies.vercel.app/.

C Circuit constraints

In Table 1a, we give the constraint counts for various circuits for UpdateCred.

D Decreasing synchronization requirements for ProveCred

The approach for implementing ObtainCred presented in Section 4.1 has potentially onerous requirements for synchronizing the database D_2. We can use the approach of [26] to implement ObtainCred in a way that lowers these requirements. At a high level, [26] requires a user to output part of an n of n secret share of their secret key (or some other sensitive information) every time they show a credential. Users are punished for using their credential more than n times by having sensitive information revealed. This approach no longer requires D_2 to be synchronized, just for all revealed secret shares to be posted. Implemented with this approach (for a credential which can be shown once), ObtainCred would output a random point (based on a challenge from $\mathcal{S}$ or hashing appropriate session-specific data)) on a line encoding the client's key k, and a nullifier $\mathsf{PRF}(k, \mathsf{Cred}||1)$. $\mathcal{S}$ would then add this share and nullifier to a public database D_3, which auditors would check for matching nullifiers, and then punish and ban users by reconstructing keys with the same nullifier.

History Size / Circuit Type	Constraints
1	30,317
5	35,165
10	35,995
25	46,853
50	67,739
100	101,143
250	209,723
500	393,479
1,000	752,623
2,500	1,838,423
5,000	3,650,879
Streaming	24,685
Cold Start	1,526,683

(a) Number of constraints for different credential history sizes, the streaming algorithm, and the cold start functionality.

Attribution History Size	Number of Constraints	
	Single	Multi
5	35,315	35,311
10	35,391	35,391
25	39,579	39,815
50	48,047	48,583
100	60,799	61,935
250	103,239	106,175
500	175,367	181,303
1000	315,439	327,375
2500	739,839	769,775
5000	1,448,567	1,508,503

(b) Number of constraints for different attribution history sizes, comparing the single attribution method (for ad attribution) with the multi attribution method (for verified browser history checks). All circuits have credential history size = 5 and produce a RAPPOR report.

Table 1: A side-by-side comparison of the total number of circuit constraints across all zk-Cookies configurations, across a variety of history size parameters.

```
// The fields of the stateful credential
struct F:
  ip_history[n] // n most recent IPs
  geohashes[n] // geolocation of n most recent logins
  last_fingerprint // fingerprint of most recent login
  year_of_birth // Year of birth of the user.

// Information required to initialize a credential
struct I:
  ip // user's initial IP
  geohash // user's initial login location
  fingerprint // user's initial browser fingerprint
  passport // string of user's passport data.

InitialState(init_info: I):
  Verify that I.passport is a valid passport.
  return F with:
    ip_history set to [I.ip, 0, 0, ...]
    geohashes set to [I.geohash, 0, 0, ...]
    last_fingerprint set to I.fingerprint
    year_of_birth set to the year of birth in I.passport

// Signed information provided from the server to user
struct $T_S$:
  new_ip // IP the server sees for a given request
  new_location // Location for user's IP

struct $T_C$:
  new_fingerprint // The new browser fingerprint

UpdatePolicy(prev_state: F, server_info: $T_S$, user_info: $T_C$,
    r: $\{0,1\}^*$):
  // fingerprint_dist and geohash_dist are thresholds for
  // the maximum allowed distances between measurements.
  Verify that F.last_fingerprint and user_info.
      new_fingerprint are closer than threshold
      fingerprint_dist.
  Verify that server_info.new_geohash is within distance
      geohash_dist a geohash in F.geohashes
  Update F as follows:
    Prepend server_info.new_ip to F.ip_history, and
        remove the last element of F.ips
    Prepend server_info.new_location to F.geohashes, and
        remove the last element of F.geohashes.
    Set F.last_fingerprint to user_info.new_fingerprint.
  return updated F

ValidPolicy(state: F):
  // In our instantiation, validation of credentials
  // primarily happens at the cryptographic level (i.e.
  // verifying SNARKs and associated signatures)
  Verify that year_of_birth is more than 18 years ago.
  return 1

ResponsePolicy(state: F, r: $\{0,1\}^*$):
  Encode analytics, like the number of non-null IPs in
      the history or the average distances between
      geohashes in the location history in a RAPPOR
      response A using randomness r.
  Return A
```

Figure 7: Pseudocode and labeled structs for input/output types for our prototype zk-cookie application (InitialState, UpdatePolicy, ValidPolicy, ResponsePolicy)

E Pseudocode for instantiating InitialState, UpdatePolicy, ValidPolicy and ResponsePolicy

We give pseudocode for functions and structs for instantiating the ideal functionality that perform the same checks as in the implemented zk-Cookies system in Figure 7.

F Security proof sketch for Continuous Anonymous Authentication construction

We now present a proof sketch that our construction realizes the ideal functionality $\mathcal{F}_{\mathsf{caa}}$ in Figure 1 against computationally bounded adversaries assuming: 1) only static corruption of parties. 2) the underlying cryptographic primitives are correct. 3) all parties, in the real protocol, operate over anonymous connections (e.g., Tor).

Specifically, we assume the existence of a secure and collision-resistant pseudorandom function PRF (we use key-prefixed Poseidon [43] in our implementation), a secure commitment scheme Comm (instantiated with Poseidon), a secure signature scheme Sig (instantiated with ECDSA over the Baby Jubjub curve [66]), a cryptographic accumulator supporting nonmembership proofs (using a Merkle Tree as in [59]), and a secure zk-SNARK (we use the Groth16 proof system over the BLS12-381 ellptic curve). We also assume that all parties have the same view of the bulletin boards used for the nullifier databases.

To prove correctness, we describe a simulator Sim for our real protocol in the ideal world and argue it is correct. Our proof follows the basic structure of [59].

Sim runs Setup to produce trapdoors for simulation and extraction of zero-knowledge proofs for the circuits used in the scheme, as well as public parameters for the zero-knowledge proofs. Sim also sets up a table that maps the states of credentials in the simulated protocol to those in the ideal protocol $\mathcal{F}_{\mathsf{caa}}$, as well as empty nullifier databases D_1, D_2. When creating or updating the state of a credential Cred, Sim updates the mapping accordingly.

In ObtainCred,UpdateCred, and ProveCred, messages from client $\mathcal{C}_i$ to the server $\mathcal{S}$ include an associated proof. For adversarially generated messages, Sim can extract the relevant values (measurements and inputs from the client), and then send the relevant request to the ideal functionality. For honest interactions with the ideal functionality, Sim can model $\mathcal{S}$'s view of the protocol by simulating proofs with random inputs, random commitments and random PRF values as outputs. By the soundness of the SNARK, clients are bound to honestly compute the circuits in each SNARK.

Now we sketch the details of Sim simulating each subprotocol in our Continuous Anonymous Authentication construction. We focus on specific details of simulating each protocol, as we have described the commonalities of proving each protocol secure in the previous paragraph.

Simulating ObtainCred. $\mathcal{S}$'s view in ObtainCred can be simulated via the high-level approach described above: by simulating $\mathcal{C}_i$'s proofs with random inputs; and random commitments/PRF outputs. More interestingly, because the PRF is secure and collision-resistant, and the bulletin board is synchronized, clients are guaranteed to have unique keys and are guaranteed to only be able to call ObtainCred once per key except with negligible probability,

Simulating UpdateCred. Since $\mathcal{M}$ only ever sees t, a PRF output, Sim can simulate $\mathcal{M}$'s view by sending a uniformly random value t. Similarly, $\mathcal{S}$'s view in the protocol can be simulated by the high-level approach described previously, as well as by producing a simulated analytics string A.

By an argument similar to what is outlined in Section 4.1, the construction for Measure can be shown to bind each measurement to one client and credential state. As justified previously in Section 5.3, our construction produces randomness with the desired properties for ResponsePolicy. By the security of Comm and Sig, $\mathcal{C}_i$ can only use valid credentials obtained from previous executions of UpdateCred or ObtainCred as inputs to UpdateCred. By the security of PRF and the bulletin board D_2 being synchronized, $\mathcal{C}_i$ can only use a signed commitment to a previous state once except with negligible probability. Additionally, by the collision-resistance of PRF and the aforementioned security properties, users cannot interrupt other users' state transitions, except with negligible probability. Consequently, $\mathcal{C}_i$'s behavior matches the behavior of the ideal functionality except with negligible probability.

Simulating ProveCred. Lastly, $\mathcal{S}$'s view in ObtainCred can be simulated in the same manner as the other methods. The argument for $\mathcal{C}_i$'s security is similar to the other methods, and follows from the nullifier database being synchronized and the cryptographic security of the relevant primitives.