Divisible E-Cash for Billing in Private Ad Retargeting

Kevin Liao
MIT and Harvard Law School

Henry Corrigan-Gibbs
MIT

Dan Boneh
Stanford University

ABSTRACT

This paper presents new techniques for *private billing* in systems for privacy-preserving online advertising. In particular, we show how an ad exchange can use an e-cash scheme to bill advertisers for ad impressions without learning which client saw which ad: The exchange issues electronic coins to advertisers, advertisers pay publishers (via clients) for ad impressions, and publishers unlinkably redeem coins with the exchange. To implement this proposal, we design a new *divisible* e-cash scheme that uses modern zero-knowledge proofs to reduce the ad exchange's computational costs by roughly 250× compared to the previous state-of-the-art. With our new e-cash scheme, our private-billing infrastructure adds little overhead to existing private ad-retargeting systems: less than 63 ms of latency, negligible client computation, less than 3.2 KB of client communication, and a combined server operating cost (advertisers, publishers, and exchange) of less than 1% of ad spend, an over 5× savings compared to the previous state-of-the-art.

1 INTRODUCTION

Retargeted ads are the ads that "follow you around the web." Say you visit the website of a shoe retailer, add a pair of shoes to your shopping cart, and then browse away without making the purchase. Over the next week, as you browse through your favorite news sites and blogs, you see ads for that particular pair of shoes.

Behind the scenes, an ad exchange has been tracking your web-browsing activity using third-party cookies [26], creating a profile of your demographics and interests based on the websites you have visited. When an opportunity arises to show you an ad, the exchange coordinates with a number of advertisers, and other intermediaries, to auction off the chance to show an ad someone with your exact demographics and your exact interests to the highest bidder. In the time you wait for a webpage to load, a multitude of advertisers and intermediaries have gained access to your preferences and online behavior. Finally, the exchange charges advertisers and pays publishers based on your ad impressions (views) and clicks.

Privacy-conscious users have long recognized the invasive privacy practices of targeted advertising [43, 44, 48, 56, 57, 76]. To address these issues, over the past two decades, academic researchers have proposed a handful of ideas for *private ad targeting* [16, 46, 66, 71, 74, 78, 88, 96, 103]. More recently, with major browser vendors deprecating third-party cookies [44, 77, 94, 100], a new wave of industry proposals has cropped up, blending new and existing ideas to support various kinds of targeted advertising in a third-party-cookieless world [3, 24, 54, 55, 98, 99].

Proceedings on Privacy Enhancing Technologies 2024(4), 5–23

https://doi.org/10.56553/popets-2024-0104

Recent private ad-retargeting proposals [3, 54, 99], such as Google's Protected Audience API [3] (formerly known as FLEDGE [54]), shift the retargeting pipeline, traditionally done by the exchange, to the client's web browser: the client's browser tracks retargeting opportunities, runs ad auctions with advertisers, and serves ads on publisher websites, all without involvement from the exchange. This not only grants clients privacy, but also transparency into and control over retargeting.

These proposals, however, do not address privacy of the billing process: how the exchange will correctly charge advertisers and pay publishers for ad impressions, without requiring the exchange to collect sensitive client data.

Our approach. In this paper, we propose that the exchange use an *e-cash* scheme [40] to privately bill for ad impressions without needing to learn clients' web-browsing activity: The exchange issues electronic coins to advertisers, advertisers pay publishers (with messages proxied through the client's browser) for ad impressions, and publishers redeem coins with the exchange. E-cash payments are *unlinkable*, meaning that an adversarial exchange cannot determine which advertiser paid which publisher—which potentially leaks sensitive information about clients' web-browsing behavior. All the exchange learns is what it can infer from advertiser expenditures and publisher earnings. By combining our private-billing scheme with a Protected-Audience-like retargeting system, we construct a system for online advertising with end-to-end privacy guarantees.

One practical barrier is that existing e-cash schemes may be too computationally expensive to deploy for advertising at web scale. A long line of work on divisible e-cash [15, 22, 34–37, 82, 84] has yielded state-of-the-art schemes with very low asymptotic costs [22, 36, 37, 84]. Yet, these schemes remain concretely expensive—they require two pairing operations per unit spent (e.g., per $0.01 spent). We estimate that the computational cost of e-cash validation alone would require an ad exchange to spend *5% of its total ad revenue* (Section 6). In contrast, our scheme requires a number of pairings that is only *logarithmic* (instead of linear) in the maximum spend value.

To circumvent this barrier, we design a new divisible e-cash scheme with reduced concrete costs. Most divisible e-cash schemes, including ours, rely on constrained pseudorandom functions ("constrained PRFs") [21]. Prior e-cash schemes [22, 36, 37, 84] are based on costly pairing-based constrained PRFs, since these schemes compose naturally with the Groth-Sahai zero-knowledge proofs [69, 70] that these constructions use.

In contrast, in our construction, we do not need to prove statements in zero-knowledge about the constrained PRF, so we can rely on simple and fast symmetric-key-based constrained PRFs [63]. Our construction does require the coin spender (i.e., advertiser) to generate a zero-knowledge proof of inclusion in a Merkle tree.

To do so, we use modern succinct zero-knowledge proofs (zk-SNARKs) [68, 89], which are heavily optimized for exactly this type of statement.

Still, applying zk-SNARKs directly does not yield a scheme that meets our efficiency targets out of the box. To start, zk-SNARKs have relatively high proof-generation costs, which can be detrimental to latency. Fortunately, in our application setting, we can move proof generation completely offline to eliminate their online latency cost. In addition, we devise a further latency optimization by relying on a new (but entirely reasonable) collision-resistance assumption of the pseudorandom generator underlying our constrained PRF.

Experimental results. We implement our divisible e-cash scheme (and auxiliary schemes) in 3.4k lines of Rust code. In our experimental evaluation, we measure the overhead our scheme would add to Protected-Audience-like systems. First, we find that our scheme adds less than 63 ms of latency. For our target of placing ads on websites in under 150 ms, this leaves ample time for other parts of the private ad-retargeting process, such as ad auctioning and ad serving. For clients, our scheme adds negligible computation and less than 3.2 KB of communication, just over 0.1% of the average page weight of 2.2 MB [92]. For servers, our scheme adds a combined server operating cost (advertiser, publisher, and exchange) of less than 1% of ad spend (vs. 5% of ad spend for double-spend detection alone using existing schemes), comprised primarily of proof precomputation by advertisers. If we isolate the exchange's computational costs, then our scheme achieves a roughly 250× reduction compared to existing schemes.

Limitations. While achieving a combined server operating cost of less than 1% of ad spend is a significant improvement, Meta's private ad measurement system (an orthogonal problem to private ad targeting) has a target (though not yet achieved) operating cost of 0.1% of ad spend [85]. Still, our 250× reduction in the exchange's computational costs is favorable, since costs are spread across multiple parties rather than concentrated at the exchange. A fruitful direction for future work is to further reduce operating costs while maintaining comparable latency and client costs, and the same privacy guarantees as our divisible e-cash scheme.

Furthermore, we have not yet integrated our divisible e-cash scheme for testing into the Protected Audience API, since it has been under development in the course of this research. As API features become available, integrating our scheme with the API and collecting end-to-end benchmarks will shed light on what a realistic latency overhead for our scheme should be, though we expect our current latency overhead to be sufficiently low.

Finally, we do not present new solutions for ad-fraud detection or ad-exchange auditability. As is the case in today's advertising ecosystem, our system does not guarantee that, in the presence of malicious clients, advertiser payments necessarily lead to client ad impressions. Moreover, we do not address head-on the issue of malicious ad exchanges overcharging advertisers or underpaying publishers, though our system may obviate much of the need for auditing at the exchange, since auctions are delegated to clients.

Contributions. The main contributions of this paper are:

- the identification of private billing as a missing piece in recent private ad-retargeting systems and a proposed fixed using e-cash (Section 2),
- a new divisible e-cash scheme with optimizations that lead to low latency, low client cost, and low server operating cost payments (Section 4), and
- an open-source implementation (Section 5), which is publicly available at https://github.com/kev-liao/private-billing, and experimental evaluation (Section 6) of our scheme.

2 MOTIVATION AND GOALS

In this section, we overview the current, non-private ad-retargeting ecosystem, how the Protected Audience API [3] aims to address privacy issues in today's ecosystem, and gaps in the API that this research aims to fill.

2.1 Ad retargeting today

The web-advertising ecosystem is vast and complex [47]. In the context of ad retargeting, we focus on the primary participants: *clients* browse the web, visiting advertiser and publisher websites; *advertisers* pay to retarget ads to clients who have previously visited their websites; *publishers* monetize their websites by showing ads to clients; and the *exchange* acts as broker, matching advertisers to publishers.

Today, the ad-retargeting pipeline is centered on the exchange. The exchange tracks clients' web-browsing activity via third-party cookies [26] and identifies retargeting opportunities when clients visit advertiser websites. When a client visits a publisher's website, the exchange initiates an auction with the client's visited advertisers. The advertisers submit sealed bids to the exchange for the price they are willing to pay for ad interactions (impressions and clicks), the auction determines the winning advertiser and the *clearing prices* for the interactions, and the winning advertiser serves an ad to the client on the publisher's website. Finally, the exchange charges advertisers and pays publishers for interactions, and shares measurement data with advertisers to help them optimize their campaigns. The central role of the exchange gives it unchecked access to clients' web-browsing activity.

2.2 Private ad retargeting with the Protected Audience API

Recent private ad-retargeting proposals [3, 54, 99] shift the retargeting pipeline, traditionally done by the exchange, to the client's web browser. In this paper, we focus on Google's Protected Audience API [3], which is currently under testing and development in Chrome.[1]

The unhighlighted parts of Figure 1 describe the Protected Audience API at an abstract level; the highlighted parts describe our private-billing extension, which we describe shortly. At a high-level, the client's browser tracks the advertisers they visit to later set up retargeting opportunities (step 0-B). Upon visiting a publisher website that wants to serve a retargeted ad (step 1), the browser runs a local auction with its visited advertisers to determine a winning

[1]Much of this research was conducted when the Protected Audience API was known as FLEDGE [54]. Parts of the API still refer to its former name [91].

0-A: Advertisers obtain coins from exchange. Advertisers exchange hard currency (e.g., U.S. dollars) for electronic coins to pay publishers for ad impressions.

0-B: Client visits advertiser websites. When a client visits an advertiser's website, the advertiser invokes a JavaScript API to store a callback URL on the client's browser. The browser uses this callback to later run ad auctions.

1: Client visits publisher website. When the client visits a publisher's website, the publisher invokes a JavaScript API to indicate its desire to serve a retargeted ad on its page.

2: Client runs ad auction. The client's browser initiates an ad auction (e.g., a first-price, open-bid auction) by calling the stored callback URLs for the advertisers it has visited. Each callback returns the corresponding advertiser's bid for the impression. The browser runs the auction and notifies the winning advertiser of the clearing price for the impression.

3: Winning advertiser sends ad and coins to client. The winning advertiser sends the URL of its ad creative to the client's browser along with coins amounting to the clearing price.

4: Client and publisher forward coins to exchange. The client's browser forwards the coins to the publisher, who forwards them to the exchange.

5: Exchange checks validity of coins. If the coins are valid and have not been double-spent, then the exchange accepts, sends the publisher hard currency for the value of the redeemed coins, and notifies the client of acceptance through the publisher. Otherwise, the exchange rejects and the client's browser retries with the next winner of the auction.

6: Client displays ad. If the exchange accepts, then the client's browser displays the ad on the publisher's website in an isolated *fenced frame* [59], which prevents the publisher from learning who the winning advertiser is.

Figure 1: Combined workflow of private ad retargeting with the Protected Audience API [3] and private billing with e-cash (this work). The parts corresponding to private billing are highlighted; everything else is part of Protected Audience.

advertiser (step 2). The winning advertiser sends its ad to the client (step 3) and the browser displays the ad in a *fenced frame* [59] (step 6), which prevents the publisher from seeing the displayed ad and inferring the winning advertiser.

The Protected Audience API seems to obviate the need for a traditionally privacy-invasive exchange in the ad-retargeting process. Moreover, clients themselves serve as proxies between advertisers and publishers, so an advertiser cannot link a publisher's identity to a client (and vice versa). This not only improves client privacy, but also grants clients transparency into and control over the retargeting process. However, the Protected Audience API postpones the question of *billing* and how it will be achieved privately. That is, absent an exchange, it is unclear how advertisers will be correctly charged and publishers correctly paid. Presumably, an exchange exists out-of-band from the ad-retargeting protocol to perform billing, but it is further unclear how this would be achieved without the sensitive client data it collects today.

To sharpen the point: The Protected Audience API trivially supports payments but not privacy for ad *clicks*. Ad clicks inherently reveal client and publisher identities (through client IPs and Referer headers [9], respectively) to the advertiser and the exchange, which allows the exchange to tally clicks. However, the Protected Audience API supports privacy but not payments for ad *impressions*. Ad impressions alone hide client and publisher identities from the advertiser and the exchange by design through the use of fenced frames. In the absence of clicks, the exchange does not learn which ads were served on which publisher websites, so it cannot easily tally impressions. Yet, private billing is a necessary piece of a functioning private-advertising system.

2.3 Adding private billing to the Protected Audience API

To accommodate private billing, we propose to reinstate a limited exchange that uses an e-cash scheme [40] to privately bill for ad impressions without needing to learn clients' web-browsing activity: The exchange issues electronic coins to advertisers, advertisers pay publishers (with messages proxied through the client's browser) for ad impressions, and publishers redeem coins with the exchange. Importantly, e-cash payments are *unlinkable*, meaning that a coin's redemption cannot be traced back to its respective issuance. Therefore, the exchange learns nothing about clients, except what it can infer from advertiser expenditures and publisher earnings.

The highlighted parts of Figure 1 describe how to integrate private billing with e-cash into the Protected Audience API. Steps 0-A and 0-B are "offline," meaning they run in the background, while steps 1–6 are "online," meaning they are latency-critical. Offline, advertisers pay the exchange in hard currency for electronic coins, which will later be used to pay publishers for ad impressions (step 0-A). As in the original Protected Audience API, the client's browser tracks retargeting opportunities and initiates a local auction upon visiting a publisher's website (steps 0-B, 1, 2). After determining the winning advertiser and the clearing price for the impression, the winning advertiser sends to the browser its ad along with coins amounting to the clearing price (step 3). The browser forwards these coins to the publisher, who then forwards them to the exchange (step 4). The exchange checks that the coins are valid and have not been double-spent (step 5). If the exchange accepts, then it pays the publisher in hard currency for the value of the redeemed coins (this can delayed to the end of a billing cycle). Otherwise, the exchange rejects and the browser retries with the next winner of the auction. Finally, the browser displays the ad only after the exchange accepts (step 6).

The end result is that the advertiser does not learn what website the client visited, and the publisher does not learn what ad the client's browser displayed on its page. Yet, the parties are properly billed and compensated.

2.4 Goals

To this end, we require the e-cash scheme to meet several properties. For intuition, we present them here informally as the system-level properties they would achieve.

- *Correctness.* If all parties are honest, then the exchange correctly charges advertisers and pays publishers.
- *Unforgeability.* When an honest exchange interacts with a coalition of adversarial advertisers and publishers, the exchange's total revenue is always non-negative. That is, a coalition of adversarial advertisers cannot extract more hard currency from the exchange than they put in.
- *Unlinkability.* An attacker who controls the exchange learns nothing about the client's web-browsing activity, except what the exchange can infer from advertiser expenditures and publisher earnings.

To scale e-cash to the volume of traffic present in today's web-advertising ecosystem while preserving (or even improving) user web-browsing experience, the scheme should allow the Protected Audience API to meet the following efficiency targets:

- *Low latency.* Ads should be placed on publisher websites in under 150 ms [47].
- *Low client costs.* Client computation and communication costs should be less than 1% of average page load time and average page weight, respectively.
- *Low server operating costs.* The combined server operating cost for advertisers, publishers, and the exchange should be less than 1% of ad spend.

There are several important non-goals for private billing. The first is specific to this work while the remaining three are inherited from the Protected Audience API. Private billing need not provide:

- *Malicious client protection.* We do not prevent a malicious client from stealing the publisher's share of the coins. This is to some degree inevitable and is true of ad retargeting today, since a client can always set up its own website as a publisher and retarget ads to itself. See Section 7 for more discussion.
- *Fraud protection.* We do not guarantee that ad impressions are the result of legitimate traffic. For example, they may be the result of a click bot [10].
- *Bid privacy.* We do not hide advertiser bids from other parties (i.e., other advertisers, clients, publishers, or the exchange).
- *Privacy after ad click.* We aim to protect client privacy only up to the point that a client clicks on an advertiser's ad. After that point, we allow the advertiser to learn the identity of the referring publisher.

3 BACKGROUND

This section summarizes the cryptographic primitives we will use to build up to our divisible e-cash scheme.

Notation. For $a, b \in \mathbb{N}$, the notation $[a]$ refers to the set $\{1, 2, \ldots, a\}$, and $[a, b]$ or $[a, b)$ is standard interval notation. For a finite set S, the notation $r \xleftarrow{\$} S$ refers to choosing r independently and uniformly at random from S. For strings $a, b \in \{0,1\}^*$, the notation $a \| b$ refers to string concatenation.

Pseudorandom function (PRF). A pseudorandom function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a function that can be computed by a deterministic, polynomial-time algorithm such that on input $(k, x) \in \mathcal{K} \times \mathcal{X}$ the algorithm outputs $y \leftarrow F(k, x) \in \mathcal{Y}$ [20]. Security for a PRF says that for a randomly chosen key k, the function $F(k, \cdot)$ is indistinguishable from a random function from $\mathcal{X}$ to $\mathcal{Y}$.

Concretely, we use the VOPRF of Jarecki et al. [73], which we recall here. Let $\mathbb{G}$ be a group of prime order q with generator g, let $H : \{0,1\}^* \to \mathbb{G}$ be a hash function modeled as a random oracle, and let $F(\mathsf{sk}, x) \triangleq H(x)^{\mathsf{sk}}$, where $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and $\mathsf{pk} = g^{\mathsf{sk}}$. To obliviously evaluate $F(\mathsf{sk}, x)$, the client sends the blinded input $H(x)^r$ for a random $r \in \mathbb{Z}_q$, the server raises the blinded input to the secret key sk, and the client unblinds the signed, blinded input by raising it to $1/r$, recovering $H(x)^{\mathsf{sk}}$. To verify $F(\mathsf{sk}, x) = H(x)^{\mathsf{sk}}$, the server also returns a discrete log equality proof $\mathsf{DLEQ}(g, \mathsf{pk}, H(x)^r, H(x)^{r \cdot \mathsf{sk}})$, which proves for a tuple $(g, h, u, v) \in \mathbb{G}^4$ that there exists $\alpha \in \mathbb{Z}_q$ such that $h = g^\alpha$ and $v = u^\alpha$ [42].

Verifiable oblivious pseudorandom function (VOPRF). A verifiable oblivious pseudorandom function (VOPRF) is a protocol that allows a client holding secret input x and public key pk to evaluate a PRF $F(\mathsf{sk}, x)$ with a server holding secret key sk [73]. Security for a VOPRF says that the evaluation should be *verifiable* (the client can verify its correctness with pk) and *oblivious* (the server learns nothing about x).

Constrained pseudorandom function (CPRF). A constrained PRF (CPRF) is a PRF that additionally allows deriving, from a principal key $k \in \mathcal{K}$, a constrained key $k_{\mathcal{S}}$ for a subset $\mathcal{S} \subseteq \mathcal{X}$ [21]. This constrained key allows evaluating $F(k, x)$ for every $x \in \mathcal{S}$, but reveals nothing about the value of the function at points outside $\mathcal{S}$.

$\mathsf{CPRF.Constrain}(k, \mathcal{S}) \to k_{\mathcal{S}}$**.** Given principal key $k \in \mathcal{K}$ and subset $\mathcal{S} \subseteq \mathcal{X}$, output constrained key $k_{\mathcal{S}}$.

$\mathsf{CPRF.Eval}(k_{\mathcal{S}}, x) \to y$**.** Given constrained key $k_{\mathcal{S}}$ and input $x \in \mathcal{S} \subseteq \mathcal{X}$, output $y \in \mathcal{Y}$.

Security for a CPRF is analogous to security for a PRF, but applies to both principal keys and constrained keys. In addition, a constrained key $k_{\mathcal{S}}$ must reveal nothing about the PRF values at points outside of $\mathcal{S}$. A CPRF is *key-pseudorandom* if a sequence of constrained keys for pairwise disjoint sets is indistinguishable from a sequence of random strings.

Goldreich-Goldwasser-Micali (GGM) PRF. The GGM PRF [63] allows constrained evaluation on sets of bitstrings $\mathcal{S}$ whose members begin with a common prefix. Let $G : \{0,1\}^s \to \{0,1\}^{2s}$ be a length-doubling pseudorandom generator (PRG) [20]. Let $G_0(x)$ (or $G_1(x)$) denote the s most (or least) significant bits of $G(x)$, so that $G(x) = G_0(x) \| G_1(x)$. The GGM PRF on key $k \in \{0,1\}^s$ and input $x \in \{0,1\}^n$ is defined as $F(k, x) = G_{x_n}(G_{x_{n-1}}(\ldots(G_{x_2}(G_{x_1}(k)))))$, where $x_1 \| \cdots \| x_n$ is the binary representation of x.

Intuitively, the GGM PRF defines a binary tree on the input domain, where each leaf has label $x \in \{0,1\}^n$ and value $F(k,x) \in \{0,1\}^s$, and each internal node has label $p \in \{0,1\}^{m \le n}$ and value $F(k,p) \in \{0,1\}^s$. The internal node $F(k,p)$ is a constrained key for the domain $\mathcal{S}_p \subseteq \mathcal{X}$ of inputs prefixed by p. For convenience, we define an additional algorithm $\mathsf{CPRF.Expand}(k_{\mathcal{S}}, n) \rightarrow \mathbf{y}$, which takes constrained key $k_{\mathcal{S}}$ and $n \in \mathbb{N}$, and computes the n-level GGM-expansion of $k_{\mathcal{S}}$ into the size-2^n vector $\mathbf{y}$ of length-s bitstrings.

Merkle tree. A Merkle tree built out of a collision-resistant hash function allows one to compute a hash r (called a "Merkle root" or simply "root") of a vector of n elements $\mathbf{x} = (x_1, \ldots, x_n)$. Later, a prover can generate a short proof π validating that an element x is in the vector with root r. The verifier, knowing r, can check the proof efficiently. Formally, a Merkle tree is defined by the following algorithms:

$\mathsf{Merkle.Hash}(\mathbf{x}) \rightarrow r$. Given vector $\mathbf{x}$, output Merkle root r.

$\mathsf{Merkle.Prove}(x, \mathbf{x}) \rightarrow \pi$. Given element x and vector $\mathbf{x}$, output proof π that $x \in \mathbf{x}$.

$\mathsf{Merkle.Verify}(x, r, \pi) \rightarrow b$. Given element x, Merkle root r, and proof π, output $b = 1$ if π is valid and $b = 0$ otherwise.

Security for a Merkle tree says that an adversary cannot output a Merkle root r and then fool the verifier into accepting a proof for an element x not in the vector with root r. We also require a statistical assumption about the underlying hash function H, namely that it is also a hiding commitment: If r is uniform and unknown to the adversary and $h = H(x, r)$, then (x, h) is indistinguishable from (y, h), where y is uniform.

Commitment scheme. A commitment scheme allows a sender to commit to a secret message and later verifiably reveal the message to a receiver. Formally, a commitment scheme is defined by the following algorithms:

$\mathsf{Commit}(m) \rightarrow (\mathsf{com}, \mathsf{open})$. Given message m, output commitment-opening pair $(\mathsf{com}, \mathsf{open})$.

$\mathsf{VerifyCom}(m, \mathsf{com}, \mathsf{open}) \rightarrow b$. Given message m, commitment com, and opening open, output $b = 1$ if the opening is valid and $b = 0$ otherwise.

Security for a commitment scheme says that the commitment should be *hiding*, i.e., the commitment to a message m reveals no information about m, and *binding*, i.e., the sender cannot cheat by opening a commitment to message m to a different message m'.

Signature scheme. A signature scheme allows a signer to authenticate a message that a receiver can later verify. Formally, a signature scheme is defined by the following algorithms:

$\mathsf{Sig.KeyGen}(1^\lambda) \rightarrow (\mathsf{sk}, \mathsf{pk})$. Given security parameter 1^λ, output signing-verification key pair $(\mathsf{sk}, \mathsf{pk})$.

$\mathsf{Sig.Sign}(\mathsf{sk}, m) \rightarrow \sigma$. Given signing key sk and message m, output signature σ.

$\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, m) \rightarrow b$. Given verification key pk, signature σ, and message m, output $b = 1$ if σ is a valid signature of m under pk and $b = 0$ otherwise.

Security for a signature scheme (known as existentially unforgeable under chosen message attacks (EUF-CMA) [64]) says that an adversary cannot create a valid signature for a previously unsigned message.

Succinct non-interactive argument of knowledge (SNARK). An arithmetic circuit $C : \mathbb{F}^l \times \mathbb{F}^m \rightarrow \mathbb{F}^n$ over a finite field $\mathbb{F}$ takes a statement-witness pair $(\phi, w) \in \mathbb{F}^l \times \mathbb{F}^m$ as input and produces $C(\phi, w) \in \mathbb{F}^n$ as output. The arithmetic circuit satisfiability problem of C is captured by the relation $\mathcal{R}_C = \{(\phi, w) \in \mathbb{F}^l \times \mathbb{F}^m : C(\phi, w) = 0^n\}$ and has language $\mathcal{L}_C = \{\phi \in \mathbb{F}^l : \exists w \in \mathbb{F}^m \text{ s.t. } C(\phi, w) = 0^n\}$. A succinct non-interactive argument of knowledge (SNARK) for arithmetic circuit satisfiability is defined by the following algorithms:

$\mathsf{SNARK.KeyGen}(1^\lambda, C) \rightarrow (\mathsf{pk}, \mathsf{vk})$. Given security parameter 1^λ and arithmetic circuit C, output proving key pk and verification key vk.

$\mathsf{SNARK.Prove}(\mathsf{pk}, \phi, w) \rightarrow \pi$. Given proving key pk, statement ϕ, and witness w such that $(\phi, w) \in \mathcal{R}_C$, output proof π for $\phi \in \mathcal{L}_C$.

$\mathsf{SNARK.Verify}(\mathsf{vk}, \phi, \pi) \rightarrow b$. Given verification key vk, statement ϕ, and proof π, output $b = 1$ if π is valid and $b = 0$ otherwise.

Security for a zk-SNARK says that the system should be *complete* (honest prover convinces honest verifier), *knowledge-sound* (an extractor can extract a valid witness from a convincing prover), and *zero-knowledge* (verifier learns nothing) [62].

4 E-CASH FOR PRIVATE BILLING

In this section, we first define e-cash and the relevant properties for our private-billing application (Section 4.1). As a warm up, we then describe a simple e-cash scheme that is efficient, but, owing to a small amount of leakage (i.e., the denominations of coins), does not quite achieve the unlinkability property we are after (Section 4.2). Finally, we describe a *divisible* e-cash scheme that fixes this leakage (Section 4.3) and crucial optimizations (Section 4.4).

4.1 Definition

We define e-cash as a two-party protocol involving a *customer* and a *bank*. The bank *issues* electronic coins to the customer in exchange for hard currency, and the customer *redeems* coins with the bank to reclaim hard currency. Importantly, e-cash payments are *unlinkable*, meaning the bank cannot link a coin's redemption with its respective issuance.

Formally, e-cash is defined by the following algorithms:

$\mathsf{Bank.Setup}(1^\lambda) \rightarrow (\mathsf{st}, \mathsf{pp})$. Given security parameter 1^λ, the bank outputs initial state st, which can include secret keys and a list of serial numbers of redeemed coins to detect double-spending, and public parameters pp, which can include public keys and other cryptographic parameters. For notational simplicity, the remaining algorithms implicitly take in pp.

$\mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}, n) \rightarrow (\mathsf{st}, \mathsf{rsp})$. Given state st, issue request req from the customer, and issue amount $n \in \mathbb{N}$, the bank outputs updated state st and issue response rsp.

$\mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, n) \rightarrow (\mathsf{st}, b)$. Given state st, coins cs from the customer, and redemption amount $n \in \mathbb{N}$, the bank outputs updated state st, which may include new serial numbers, and bit b for whether or not the redemption was successful.

$\mathsf{Cust.Setup}(1^\lambda) \rightarrow \mathsf{wlt}$. Given security parameter 1^λ, the customer outputs initial wallet wlt.

$\mathsf{Cust.IssueRequest}(\mathsf{wlt}, n) \longrightarrow (\mathsf{wlt}, \mathsf{req})$. Given wallet wlt and issue amount $n \in \mathbb{N}$, the customer outputs updated wallet wlt and issue request req.

$\mathsf{Cust.IssueProcess}(\mathsf{wlt}, \mathsf{rsp}) \longrightarrow \mathsf{wlt}$. Given wallet wlt and issue response rsp from the bank, the customer outputs updated wallet wlt, which may include newly issued coins.

$\mathsf{Cust.RedeemRequest}(\mathsf{wlt}, n) \longrightarrow (\mathsf{wlt}, \mathsf{cs})$. Given wallet wlt and redemption amount $n \in \mathbb{N}$, the customer outputs updated wallet wlt and coins cs. The output cs will be the empty set $\varnothing$ if there are insufficient funds in the wallet.

For private billing in Protected Audience, the bank corresponds to the exchange and customers correspond to advertisers and publishers. To make a payment, the advertiser sends a redemption request (coins) through the client's browser to the publisher, who redeems the coins with the exchange. For more details on how these algorithms fit into Protected Audience, see Appendix D.

We require the e-cash scheme to satisfy several properties. An e-cash scheme is *correct* if, when honest customers interact with an honest bank, issuances add the correct values to customer balances and redemptions (with sufficient funds) subtract the correct values from customer balances.

Definition 4.1 (Correctness). We say that an e-cash scheme running with C customers is *correct* if, for all efficient adversaries $\mathcal{A}$, the adversary wins the following game with negligible probability in the security parameter λ.

- The challenger runs
$$(\mathsf{st}, \mathsf{pp}) \leftarrow \mathsf{Bank.Setup}(1^\lambda),$$
$$\mathsf{wlt}_i \leftarrow \mathsf{Cust.Setup}(1^\lambda) \text{ for } i \in [C],$$
sets $\mathsf{wlt}_i \leftarrow 0$ for $i \in [C]$ and $b \leftarrow 1$, and sends pp to $\mathcal{A}$.
- The adversary $\mathcal{A}$ can send polynomially many issue and redeem requests to the challenger, each time specifying a customer $i \in [C]$:
  - On issue request of value n, the challenger runs
  $$(\mathsf{wlt}_i, \mathsf{req}) \leftarrow \mathsf{Cust.IssueRequest}(\mathsf{wlt}_i, n),$$
  $$(\mathsf{st}, \mathsf{rsp}) \leftarrow \mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}, n),$$
  $$\mathsf{wlt}_i \leftarrow \mathsf{Cust.IssueProcess}(\mathsf{wlt}_i, \mathsf{rsp}),$$
  and sets $\mathsf{bal}_i \leftarrow \mathsf{bal}_i + n$.
  - On redeem request of value $n \in [0, \mathsf{bal}_i]$, the challenger runs
  $$(\mathsf{wlt}_i, \mathsf{cs}) \leftarrow \mathsf{Cust.RedeemRequest}(\mathsf{wlt}_i, n),$$
  $$(\mathsf{st}, b) \leftarrow \mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, n),$$
  and sets $\mathsf{bal}_i \leftarrow \mathsf{bal}_i - n$.

We say that $\mathcal{A}$ wins the game if $b = 0$.

An e-cash scheme is *unforgeable* if malicious customers cannot redeem coins for more value than was issued to them.

Definition 4.2 (Unforgeability). We say that an e-cash scheme is *unforgeable* if, for all efficient adversaries $\mathcal{A}$, the adversary wins the following game with negligible probability in the security parameter λ.

- The challenger runs $(\mathsf{st}, \mathsf{pp}) \leftarrow \mathsf{Bank.Setup}(1^\lambda)$, sends pp to $\mathcal{A}$, and sets $\mathsf{bal} \leftarrow 0$.
- The adversary $\mathcal{A}$ can send polynomially many issue and redeem requests to the challenger:
  - On issue request req of value n, the challenger runs $(\mathsf{st}, \mathsf{rsp}) \leftarrow \mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}, n)$, sends rsp to $\mathcal{A}$, and sets $\mathsf{bal} \leftarrow \mathsf{bal} + n$.
  - On redeem request cs of value n, the challenger runs $(\mathsf{st}, b) \leftarrow \mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, n)$, sends b to $\mathcal{A}$, and sets $\mathsf{bal} \leftarrow \mathsf{bal} - n$ if $b = 1$.

We say that $\mathcal{A}$ wins the game if $\mathsf{bal} < 0$.

An e-cash scheme is *unlinkable* if a malicious bank cannot link redemptions to issuances or link redemptions to the same wallet as long as customers have sufficient funds. (A malicious bank can, however, learn when customers run out of funds, which is inherent.)

Definition 4.3 (Unlinkability). We say that an e-cash scheme is *unlinkable* if, for all efficient adversaries $\mathcal{A}$, the following game is won with negligible advantage in the security parameter λ.

- The challenger runs
$$b \xleftarrow{\$} \{0, 1\},$$
$$(\mathsf{st}, \mathsf{pp}) \leftarrow \mathsf{Bank.Setup}(1^\lambda),$$
$$\mathsf{wlt}_i \leftarrow \mathsf{Cust.Setup}(1^\lambda) \text{ for } i \in \{0, 1\},$$
and sends $(\mathsf{st}, \mathsf{pp})$ to $\mathcal{A}$.
- The adversary $\mathcal{A}$ can send polynomially many issue and redeem requests, and a challenge request to the challenger:
  - On issue request of value n for customer $i \in \{0, 1\}$:
    * The challenger runs
    $$(\mathsf{wlt}_i, \mathsf{req}) \leftarrow \mathsf{Cust.IssueRequest}(\mathsf{wlt}_i, n)$$
    and sends req to $\mathcal{A}$.
    * The adversary $\mathcal{A}$ sends rsp to the challenger.
    * The challenger runs
    $$\mathsf{wlt}_i \leftarrow \mathsf{Cust.IssueProcess}(\mathsf{wlt}_i, \mathsf{rsp}).$$
  - On redeem request of value n for customer $i \in \{0, 1\}$:
    * The challenger runs
    $$(\mathsf{wlt}_i, \mathsf{cs}) \leftarrow \mathsf{Cust.RedeemRequest}(\mathsf{wlt}_i, n)$$
    and sends cs to $\mathcal{A}$.
  - On challenge request of value n:
    * The challenger runs
    $$(\mathsf{wlt}_i, \mathsf{cs}_i) \leftarrow \mathsf{Cust.RedeemRequest}(\mathsf{wlt}_i, n)$$
    for $i \in \{0, 1\}$, and sends cs_b to $\mathcal{A}$.
    * If $\mathsf{cs}_i = \varnothing$ for any $i \in \{0, 1\}$, then the game terminates and $\mathcal{A}$ loses.
- $\mathcal{A}$ outputs b'.

We say that $\mathcal{A}$ wins the game if $b = b'$.

E-cash schemes sometimes require additional properties, such as traceability (the identity of double-spenders will be revealed) [41], but these turn out to be superfluous in our application setting, so we do not require them (see Section 7 for more discussion).

4.2 A simple e-cash scheme

As a warm up, we begin with a simple e-cash scheme that is efficient, but has a small amount of leakage. Although this leakage may be tolerable in practice, the simple scheme ends up satisfying a weaker notion of unlinkability than the one we set out to achieve.

Our construction. As a starting point, the VOPRF of Jarecki et al. [73] directly yields a simple, single-denomination e-cash scheme. To start, the bank generates and holds a signing key $\mathsf{sk} \in \mathbb{Z}_q$. To request a coin, the customer samples a random serial number $x \in \{0,1\}^\lambda$, blinds it as $H(x)^r \in \mathbb{G}$ for $r \xleftarrow{\$} \mathbb{Z}_q$, obtains the VOPRF output $F(\mathsf{sk}, H(x)^r) = H(x)^{r \cdot \mathsf{sk}}$, and unblinds it to obtain $c \leftarrow H(x)^{\mathsf{sk}}$, a verifiably signed coin with serial number x. To redeem

c, the customer sends (x, c) to the bank, which checks that $c = F(\mathsf{sk}, H(x))$ and that x has not been previously spent. The customer repeats this protocol as needed for larger payments.

Correctness, unforgeability, and unlinkability of the single-denomination scheme is straightforward. However, since there is only a single denomination, the scheme works poorly when customers want to make payments for vastly different amounts. In the ads context, prices vary widely: The average cost per thousand impressions can range from \$3 to almost \$40 [1]. Moreover, bids should be sufficiently granular to allow for bid micro-optimization. As a result, payments can require many coins.

To fix the inefficiencies of the single-denomination scheme, our simple e-cash scheme supports coins of different denominations—namely power-of-two-valued denominations, $2^0, 2^1, \ldots, 2^\ell$, for some parameter $\ell \in \mathbb{N}$—by running $\ell + 1$ single-denomination schemes concurrently (each with a different signing key). The full details of our simple scheme are presented in Construction A of Appendix A.

Efficiency. The efficiency benefits of the simple scheme are two-sided. For coin issuance, a v-valued coin, for $v \in \{2^0, 2^1, \ldots, 2^\ell\}$, reduces customer/bank computation/communication and customer storage by a factor of v over the single-denomination scheme. For coin redemption, customers can redeem value $n \in [0, 2^{\ell+1})$ using a number of coins equal to the Hamming weight of n, resulting in $O(\ell)$ customer/bank computation/communication costs.

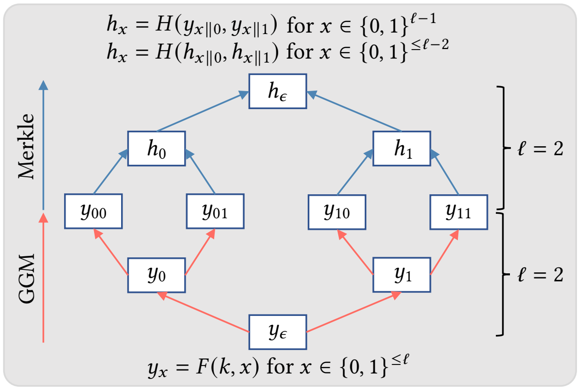
Security. Unfortunately, the efficiency of the simple scheme comes with a small loss in security. Unlinkability only holds for coins of the same denomination, since the bank learns the denominations of requested coins. Therefore, if only a few issuances of coins with value, say, 2^ℓ, occur, then redemption of these coins can be linked back to the small set of issuances, which is a rather weak privacy guarantee. In practice, this small amount of leakage may be tolerable, since rare denominations may be unlikely. Still, a bank that learns coin issuance patterns over time may be able to further de-anonymize redemptions, so we believe it prudent to patch this leakage.

4.3 A divisible e-cash scheme

To recover the security of the single-denomination scheme while retaining the efficiency of the simple scheme, we construct a *divisible* e-cash scheme, drawing from an existing line of work [15, 22, 34–37, 82, 84]. The main idea of divisible e-cash is to allow customers to efficiently request a fixed number of unit-valued coins, which can then be efficiently and unlinkably redeemed in batches of varying size.

While state-of-the-art divisible e-cash schemes [22, 36, 37, 84] rely on complicated Groth-Sahai non-interactive zero-knowledge proofs [69, 70] and concretely expensive pairing-based CPRFs, our scheme instead takes advantage of more modern zk-SNARKS [68, 89], which allows us to use a symmetric-key-based, GGM-based CPRF while maintaining good concrete efficiency.

Our construction. We present the full details of our divisible e-cash scheme in Construction 1 and explain the high-level ideas here. To start, the bank runs Bank.Setup, which generates key pairs for signatures and zk-SNARKs, initializes its double-spend list to keep



$h_x = H(y_{x\|0}, y_{x\|1})$ for $x \in \{0,1\}^{\ell-1}$
$h_x = H(h_{x\|0}, h_{x\|1})$ for $x \in \{0,1\}^{\leq \ell-2}$
Merkle
GGM
h_ϵ
h_0
h_1
y_{00}
y_{01}
y_{10}
y_{11}
y_0
y_1
y_ϵ
$\ell = 2$
$\ell = 2$
$y_x = F(k, x)$ for $x \in \{0,1\}^{\leq \ell}$

Figure 2: Divisible e-cash tree of height $\ell = 2$. The leaves of the Merkle tree are coin serial numbers generated by GGM-expanding a principal key k. Each node of the Merkle tree represents a batch of coins with value equal to the number of leaf nodes under it.

track of the serial numbers of spent coins, and publishes public parameters. The customer runs Cust.Setup, which initializes its wallet state to empty.

To begin the issuance protocol, the customer runs Cust.IssueRequest for amount $n = 2^\ell$, which is fixed by parameter $\ell \in \mathbb{N}$. In this algorithm, the customer generates a principal GGM CPRF key and computes its ℓ-level GGM-expansion into 2^ℓ leaves. Each leaf is a serial number corresponding to a unit-valued coin. The customer also computes a Merkle root over these leaves.

Intuitively, each Merkle tree node succinctly describes a batch of coins with value equal to the number of leaves underneath. Figure 2 shows a Merkle tree of height $\ell = 2$ with four leaves $y_{00}, y_{01}, y_{10}, y_{11}$, each of unit value. The internal node h_0 has value two, covering leaves y_{00}, y_{01}. The corresponding constrained key y_0 succinctly represents and generates these leaves. For clarity, we henceforth refer to an entire Merkle tree as a *full tree* and its root as the *full root* to distinguish from a subtree and its root.

The customer then sends a commitment to the full root to the bank in an issuance request. Upon receiving the request, the bank runs Bank.Issue, which returns a signature of the commitment to the customer. Finally, the customer runs Cust.IssueProcess, which verifies the signature and populates its wallet with coins that can now be spent.

To begin the redemption protocol, the customer runs Cust.RedeemRequest for amount $n \in [0, 2^{\ell+1})$. At a high-level, the customer finds in its wallet a number of subtrees (equal to the Hamming weight of n) that cover n unspent leaves. For each subtree, the customer sends to the bank as part of its redemption request the constrained key covering the subtree leaves, the subtree root, and a zk-SNARK proof of the following statement: *"Given the root of a subtree* root *(the statement), I know a signature σ, commitment* com, *opening information* open, *full root* root*, *and Merkle proof* π_{path} *(the witness) such that:*

- *the signature is on the commitment to the full root:* $\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, \mathsf{com}) = 1$ *with bank public key* pk,

- *the commitment opens to the full root:* $\mathsf{VerifyCom}(\mathsf{root}^*, \mathsf{com}, \mathsf{open}) = 1$*, and*
- *the Merkle proof is a valid subtree membership proof:* $\mathsf{Merkle.Verify}(\mathsf{root}, \mathsf{root}^*, \pi_{\mathsf{path}}) = 1$."

Upon receiving the request, the bank runs Bank.Redeem. For each subtree in the request, the bank first recovers the subtree leaves by expanding the constrained key and then checks the following:

- no leaves have been double-spent,
- the subtree is *consistent*, i.e., the leaves hash to the root of the subtree (which is also the zk-SNARK statement), and
- the zk-SNARK proof is valid.

If all checks pass and a total of n coins are deemed valid, then the bank accepts. Otherwise, the bank rejects.

Efficiency. For coin issuance, the customer generates a request for 2^ℓ coins by constructing a height-ℓ full tree, which requires $O(2^\ell)$ PRG evaluations, $O(2^\ell)$ hashes, and a single commitment. The bank, on the other hand, issues these 2^ℓ coins by producing a single signature. For customer storage, there is a trade-off: Although storing the principal CPRF key used to generate the 2^ℓ leaves is sufficient, storing (parts of) the GGM tree and Merkle tree saves from needing to recompute subtrees when redeeming coins.

For coin redemption, the customer generates a request to redeem $n \in [0, 2^{\ell+1})$ coins using a number of subtrees equal to the Hamming weight of n, resulting in $O(\ell)$ customer computation/communication costs. Notably, this involves generating and sending $O(\ell)$ zk-SNARK proofs. The bank verifies $O(\ell)$ zk-SNARK proofs, checks consistency for $O(\ell)$ subtrees, which requires $O(2^\ell)$ PRG evaluations and $O(2^\ell)$ hashes, and checks double-spends for $O(2^\ell)$ leaves.

Security. Construction 1 meets a relaxed correctness property, which only requires correctness for redemptions with sufficient funds *and* sufficient unspent subtrees. That is, for a redemption of amount n, the customer can find a number of unspent subtrees equal to the Hamming weight of n that cover n leaves. We analyze the security of Construction 1 in Appendix B, where we prove the following properties:

LEMMA 4.4 (UNFORGEABILITY). *Construction 1 is an unforgeable e-cash scheme assuming the signature scheme is EUF-CMA-secure, the commitment scheme is binding, the zk-SNARK proof system is knowledge-sound, and the Merkle tree is built from a collision-resistant hash function.*

LEMMA 4.5 (UNLINKABILITY). *Construction 1 is an unlinkable e-cash scheme assuming the commitment scheme is hiding, the CPRF is secure and key-pseudorandom, the Merkle tree hash function is a hiding commitment, and the zk-SNARK proof system is zero-knowledge.*

4.4 Divisible e-cash optimizations

Our divisible e-cash scheme, as described in Construction 1, does not yet meet our efficiency requirements out of the box. In the context of private billing for Protected Audience, zk-SNARK proof generation and verification, which can be expensive, are currently on the online, latency-critical path. We therefore introduce several latency and operating-cost optimizations to rectify this.

zk-SNARK proof precomputation for lower latency. Our first task is to move the $O(\ell)$ zk-SNARK proof generations in Cust.RedeemRequest offline. We observe that the zk-SNARK proofs are not tied to any information unknown ahead of time, besides perhaps the value of the redemption request. Still, the customer can get around this by precomputing proofs for subtrees of each size in advance. For example, for full trees of height $\ell = 2$, customers prepare subtrees of heights 0, 1, and 2 ahead of time, and replenish whenever subtrees are used. As a result, there will always be a set of prepared subtrees covering a number of leaves that sum to any allowed redemption value. Moving proof generation offline substantially decreases latency and allows us to choose a zk-SNARK optimized along other measures, such as proof size, instead of prover time.

SNARK-friendly primitives for smaller circuits. Our choice of zk-SNARK is thus Groth16 [68], which enjoys the smallest proof size and fastest verifier time (both constant) among existing SNARKs [89]. Although Groth16's prover time is $O(n \log n)$, where n is the circuit size (in multiplication gates or equivalently R1CS constraints), we can reduce the size of our arithmetic circuits (and hence precomputation costs) by using SNARK-friendly primitives. For our Merkle tree, hash-based commitment, and Schnorr signature constructions, we instantiate all hashing with Poseidon [65], a SNARK-friendly hash function. This leads to circuits that are several orders of magnitude smaller than those using traditional hash functions—the circuit for proving leaf knowledge in a Merkle tree of 2^{30} elements is around 7k R1CS constraints using Poseidon, compared to 826k using SHA-256 or 630k using Blake2s [65].

Collision-resistant PRG for faster redemption. Although Poseidon is efficient to compute in an arithmetic circuit, it is slow to compute natively—0.03 ms of CPU time on our evaluation testbed (Section 6). In Construction 1, $O(2^\ell)$ native Poseidon hashes are required for the customer to construct a full tree and for the bank to check consistency of redeemed subtrees.

While these hashes are unavoidable on the customer, we find a way to optimize them out on the bank. Instead of the customer proving subtree membership and the bank checking subtree consistency, the customer proves membership of only the leftmost leaf of the subtree and the bank relies on an additional *collision-resistance* property of the PRG underlying GGM—an unusual requirement for a PRG—to guarantee subtree consistency.

For any general PRG, this optimization is insecure. Consider the following attack in reference to Figure 2: A malicious customer first redeems leaves y_{10} and y_{11}, proving membership of y_{10} with respect to root h_ϵ, and the bank stores y_{10} and y_{11} in its double-spend list. Next, the customer finds a different GGM key k' that expands into leaves $y_{00}, y'_{01}, y'_{10}, y'_{11}$, where $y_{10} \neq y'_{10}$ and $y_{11} \neq y'_{11}$. The customer redeems all of these leaves, proving membership of y_{00} with respect to the same root h_ϵ, and the bank will accept, since $y_{10} \neq y'_{10}$ and $y_{11} \neq y'_{11}$. Thus, the customer has successfully redeemed six leaves from a full tree with four leaves.

To prevent such an attack, we require the underlying PRG to be "left-collision-resistant." That is, given a length-doubling PRG $G(k) = G_0(k) \| G_1(k)$, it should be hard to find $k \neq k'$ such that $G_0(k) = G_0(k')$. (G_1 can also have the same property, though it is unnecessary.) The PRG $G(k) = G_0(k) \| G_1(k) = H(k\|0) \| H(k\|1)$ is

Construction 1. Divisible e-cash scheme. Let parameter $\ell \in \mathbb{N}$ be the height of a full tree.

$\underline{\mathsf{Bank.Setup}(1^\lambda) \to (\mathsf{st}, \mathsf{pp}).}$

- Generate signing key pair $(\mathsf{sk}_\Sigma, \mathsf{pk}_\Sigma) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$.
- Generate proving and verifying key $(\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{SNARK.KeyGen}(1^\lambda, C)$, where the coin validity circuit C is for the following relation: *"Given the root of a subtree* root *(the statement), I know a signature σ, commitment* com*, opening information* open*, full root* root**, and Merkle proof π_{path} (the witness) such that:*
  - *the signature is on the commitment to the full root:* $\mathsf{Sig.Verify}(\mathsf{pk}_\Sigma, \sigma, \mathsf{com}) = 1$,
  - *the commitment opens to the full root:* $\mathsf{VerifyCom}(\mathsf{root}^*, \mathsf{com}, \mathsf{open}) = 1$*, and*
  - *the Merkle proof is a valid subtree membership proof:* $\mathsf{Merkle.Verify}(\mathsf{root}, \mathsf{root}^*, \pi_{\mathsf{path}}) = 1$."
- Set double-spend list $\mathsf{dsl} \leftarrow \varnothing$.
- Set state $\mathsf{st} \leftarrow (\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Set public parameters $\mathsf{pp} \leftarrow (1^\lambda, \mathsf{pk}_\Sigma, \mathsf{pk}_\Pi, \mathsf{vk}_\Pi)$.
- Output $(\mathsf{st}, \mathsf{pp})$.

$\underline{\mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}, n) \to (\mathsf{st}, \mathsf{rsp}).}$

- If $n \neq 2^\ell$, then output $(\mathsf{st}, \perp)$.
- Parse st as $(\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Parse req as com.
- Compute signature $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_\Sigma, \mathsf{com})$.
- Set response $\mathsf{rsp} \leftarrow \sigma$.
- Output $(\mathsf{st}, \mathsf{rsp})$.

$\underline{\mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, n) \to (\mathsf{st}, b).}$

- If $n \notin [0, 2^{\ell+1})$, then output $(\mathsf{st}, 0)$.
- Parse state st as $(\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Set initial state $\mathsf{st}_0 \leftarrow \mathsf{st}$.
- Set amount $m \leftarrow 0$.
- For $(i, k_\mathcal{S}, \phi, \pi)$ in cs:
  - Compute $\mathbf{y} \leftarrow \mathsf{CPRF.Expand}(k_\mathcal{S}, \ell - i)$.
  - If $\mathbf{y} \cap \mathsf{dsl} \neq \varnothing$ or $\mathsf{Merkle.Hash}(\mathbf{y}) \neq \phi$ or $\mathsf{SNARK.Verify}(\mathsf{vk}_\pi, \phi, \pi) = 0$, then output $(\mathsf{st}_0, 0)$.
  - Add $\mathbf{y}$ to dsl in st and set $m \leftarrow m + 2^{\ell - i}$.
- If $n = m$, then output $(\mathsf{st}, 1)$. Otherwise, output $(\mathsf{st}_0, 0)$.

$\underline{\mathsf{Cust.Setup}(1^\lambda) \to \mathsf{wlt}.}$

- Set wallet $\mathsf{wlt} \leftarrow \varnothing$.
- Output wlt.

$\underline{\mathsf{Cust.IssueRequest}(\mathsf{wlt}, n) \to (\mathsf{wlt}, \mathsf{req}).}$

- Generate principal key $k \leftarrow \mathsf{CPRF.KeyGen}(1^\lambda)$.
- Compute full tree leaves $\mathbf{y} \leftarrow \mathsf{CPRF.Expand}(k, \ell)$.
- Compute full root $\mathsf{root}^* \leftarrow \mathsf{Merkle.Hash}(\mathbf{y})$ and store the full tree tree.
- Compute commitment $(\mathsf{com}, \mathsf{open}) \leftarrow \mathsf{Commit}(\mathsf{root}^*)$.
- Set request $\mathsf{req} \leftarrow \mathsf{com}$.
- Add $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \perp)$ to wlt.
- Output $(\mathsf{wlt}, \mathsf{req})$.

$\underline{\mathsf{Cust.IssueProcess}(\mathsf{wlt}, \mathsf{rsp}) \to \mathsf{wlt}.}$

- Parse response rsp as σ.
- Find $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \perp) \in \mathsf{wlt}$ such that $\mathsf{Sig.Verify}(\mathsf{pk}_\Sigma, \sigma, \mathsf{com}) = 1$ and update $\perp$ to σ.
- Output wlt.

$\underline{\mathsf{Cust.RedeemRequest}(\mathsf{wlt}, n) \to (\mathsf{wlt}, \mathsf{cs}).}$

- Set initial wallet $\mathsf{wlt}_0 \leftarrow \mathsf{wlt}$.
- Set coins $\mathsf{cs} \leftarrow \varnothing$.
- Let $n_\ell \| \cdots \| n_0$ be the binary representation of n.
- For $i \in [0, \ell]$ such that $n_i = 1$:
  - Find $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \sigma) \in \mathsf{wlt}$ such that tree contains an unredeemed subtree with root root covering 2^i leaves with labels $\mathcal{S}$ and mark the subtree as redeemed. If no such subtree exists, then output $(\mathsf{wlt}_0, \varnothing)$.
  - Compute constrained key $k_\mathcal{S} \leftarrow \mathsf{CPRF.Constrain}(k, \mathcal{S})$.
  - Build Merkle proof $\pi_{\mathsf{path}} \leftarrow \mathsf{Merkle.Prove}(\mathsf{root}, \mathbf{x})$, where $\mathbf{x}$ is the vector of depth-i nodes of tree.
  - Build zk-SNARK proof $\pi \leftarrow \mathsf{SNARK.Prove}(\mathsf{pk}_\Pi, \phi, w)$, where the statement is $\phi \leftarrow \mathsf{root}$ and the witness is $w \leftarrow (\sigma, \mathsf{root}^*, \mathsf{com}, \mathsf{open}, \pi_{\mathsf{path}})$.
  - Add $(i, k_\mathcal{S}, \phi, \pi)$ to cs.
- Output $(\mathsf{wlt}, \mathsf{cs})$.

left-collision-resistant in the random oracle model, since it is hard to find $k \neq k'$ such that $H(k\|0) = H(k'\|0)$. This effectively forces the customer to "commit" to all full tree leaves upon generating a principal CPRF key.

This optimization leads to a few simplifications to Construction 1 that we present in full in Construction B of Appendix C. At a high-level, the proof statement becomes: *"Given the leftmost leaf of a subtree* leaf *(the statement), I know a signature σ, commitment* com*, opening information* open*, full root* root**, and Merkle proof π_{path} (the witness) such that:*

- *the signature is on the commitment to the full root:* $\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, \mathsf{com}) = 1$ *with bank public key* pk,
- *the commitment opens to the full root:* $\mathsf{VerifyCom}(\mathsf{root}^*, \mathsf{com}, \mathsf{open}) = 1$*, and*
- *the Merkle proof is a valid leaf membership proof:* $\mathsf{Merkle.Verify}(\mathsf{leaf}, \mathsf{root}^*, \pi_{\mathsf{path}}) = 1$."

In Bank.Redeem, for each subtree, instead of checking consistency, the bank simply checks equality between the leftmost leaf of the subtree and the leaf used in the proof statement. We prove that this optimization is secure in Appendix C.

5 IMPLEMENTATION

We implement our simple and divisible e-cash schemes in approximately 3.4k lines of Rust code. The code is publicly available at https://github.com/kev-liao/private-billing.

For our simple e-cash scheme (Section 4.2), we adapt an existing Rust implementation of VOPRFs [7] defined over the Ristretto group [51] of Curve25519. The implementation supports batch DLEQ proofs [72], which cover multiple VOPRF evaluations with a single proof. This allows a customer (advertiser) to batch together issuance requests and the bank (exchange) to attest to the validity of the issued coins with a single proof, substantially reducing computation and communication costs.

For our divisible e-cash scheme (Section 4.3), we implement our zk-SNARKs using arkworks [14], a Rust ecosystem for zk-SNARK programming. We define the arithmetic circuit for coin validity over the scalar field of BLS12-381, a pairing-friendly elliptic curve [23]. To keep the circuit small, we use Poseidon [65] for all of our hashing purposes. Further, we implement Schnorr signatures over the JubJub curve [102], whose base field is the BLS12-381 scalar field, giving fast, in-circuit elliptic curve operations. Our arithmetic circuits comprise just under 10k R1CS constraints. We use arkwork's Groth16 [68] implementation for proof generation and verification. For a general GGM-based CPRF, we instantiate the length-doubling PRG with two AES calls: $G(k) = \mathsf{AES}(k, 0^\lambda) \| \mathsf{AES}(k, 1^\lambda)$. For the collision-resistant PRG optimization (Section 4.4), we use BLAKE3 [5] as the underlying hash function.

6 EVALUATION

In this section, we benchmark our simple e-cash (SEC) and divisible e-cash (DEC) schemes, analytically compare our divisible scheme with the state-of-the-art, and ask whether our schemes add acceptable latency, client costs, and operating costs to private ad-retargeting systems, such as Protected Audience. We run our experiments on an Amazon EC2 `c5.metal` instance using one core of an Intel Xeon Platinum 8275CL CPU clocked at 3.0 GHz. The machine runs Linux kernel version 5.10.149-133.644.amzn2.x86_64 and Rust version 1.65.0.

6.1 Benchmarks and comparisons

Figure 3 shows the costs for issuance. The top left plot varies the batch size for SEC (i.e., the number of coins per issuance request) and shows that customer/bank computation grows linearly. The top center plot varies the batch size for DEC (i.e., the number of unit-valued coins/tree leaves per issuance request) for DEC and shows that Cust.IssueRequest computation grows linearly, while other costs are constant and negligible (less than 1 ms). The top right plot also varies the batch size for DEC and shows that the cost for precomputing a zk-SNARK proof grows logarithmically. The bottom table shows that communication costs for SEC grow linearly with the batch size, while communication costs for DEC are independent of the batch size.

Figure 4 shows the costs for redemption. For our divisible e-cash scheme, we fix $\ell = 11$. We elide the computation costs for Cust.RedeemRequest, since they are constant and negligible (less than 1 ms) for both SEC and DEC with precomputation. The left plot varies the redeem value and shows that the communication costs of Cust.RedeemRequest for both schemes are proportional to the Hamming weight of the redeem value. The right plot varies the redeem value and shows: (1) the computation costs of Bank.Redeem for SEC, DEC, and unoptimized DEC without the collision-resistant PRG optimization (uDEC) are also proportional to the Hamming weight of the redeem value, and (2) the collision-resistant PRG optimization is significant.

To the best of our knowledge, no prior divisible e-cash schemes have been implemented and state-of-the-art schemes [22, 36, 37, 84] are particularly challenging to implement due to their use of Groth-Sahai proofs [69, 70], so Table 1 analytically compares our divisible e-cash scheme with the state-of-the-art. We report computation costs (in asymmetric operations) and communication costs for redemption, where n is the redeem value, $|C|$ is the circuit size, E_i is exponentiations in $\mathbb{G}_i$, and P is pairings. We elide issuance costs, since they are less critical for our application and all schemes are efficient in this regard. Zero-knowledge proof generation costs in Cust.RedeemRequest are higher in our scheme, but precomputation turns them into an offline cost. Asymptotic and concrete communication costs are sufficiently low across all schemes. Finally, our scheme reduces the number of pairings in Bank.Redeem from $O(n)$ to $O(\log n)$, which more than makes up for the higher proof generation costs, as we will show.

6.2 Main results

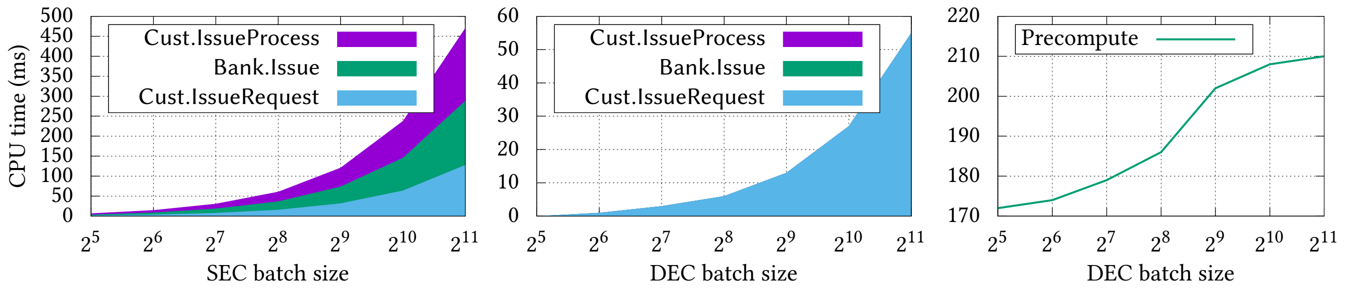
We now interpret our benchmarks in the context of private billing for Protected Audience.

Our simple e-cash scheme, though leaky, is practically efficient and still improves privacy. Support for coins of varying denominations allows advertisers to exchange large amounts of hard currency for electronic coins efficiently. Although the exchange learns the denominations of issued coins, unlinkability still holds for coins of the same denomination and this small amount of leakage may be tolerable in practice. Moreover, the use of the concretely efficient VOPRF of Jarecki et al. [73], which is standardized [49] and deployed in Privacy Pass [50], leads to essentially negligible overheads for latency, client costs, and server operating costs.

Our divisible e-cash scheme adds less than 63 ms of latency. With precomputation, Bank.Redeem becomes the only online cost and takes at most 63 ms for a spend of 4095 (Figure 4 right). Today's ad-retargeting pipeline aims to place ads on publisher websites in under 150 ms [47], so our scheme leaves ample time for other parts of the retargeting process, such as ad auctioning and ad serving. The collision-resistant PRG optimization is crucial, since latency would otherwise reach 172 ms. There are also other tricks to reduce latency, which we discuss in Section 7.

Our divisible e-cash scheme adds negligible computation and less than 3.2 KB of communication to web clients. A web client simply forwards coins to the publisher, which adds negligible computation and less than 3.2KB of communication (Figure 4 left)—just over 0.1% of the average page weight of 2.2 MB [92].

Our divisible e-cash scheme adds a combined server operating cost of less than 1% of ad spend, an over 5× savings compared to existing schemes. We use the following cost figures to calculate the combined server operating cost (advertiser, publisher, and exchange) of our scheme relative to ad spend: As of November 2022, a `c5.metal` instance with 96 vCPUs is priced



		SEC	DEC
Communication (bytes)	Cust.IssueRequest	52 · batchSize	40
	Bank.Issue	52 · batchSize	72

Figure 3: Issuance computation costs for SEC (top left), DEC (top center), and DEC coin precomputation (top right); issuance communication costs (bottom).

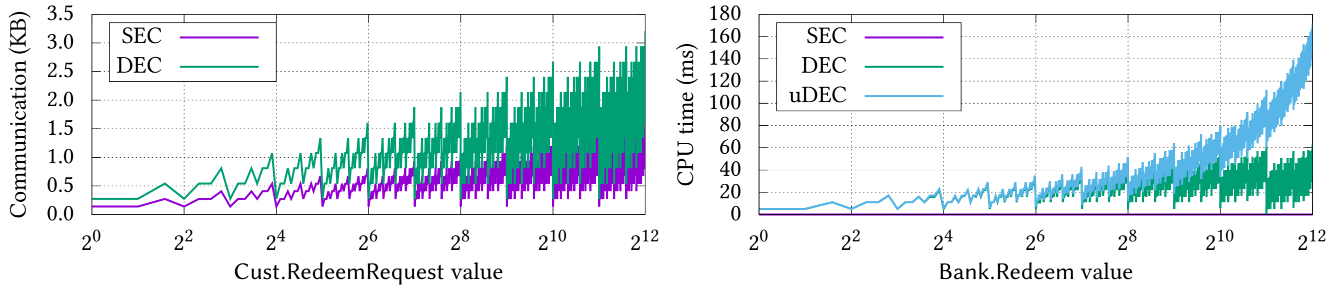


Figure 4: Redemption customer communication (left) and bank computation (right) costs for SEC, DEC, and unoptimized DEC without the collision-resistant PRG (uDEC). For DEC and uDEC, we fix $\ell = 11$ (i.e., batch size is 2^ℓ).

	CPST [36, 37], BPS [22]	PST [84]	This work (DEC)
Cust.RedeemRequest*	$O(\log n)\ E_1 + O(\log n)\ P$	$O(1)\ E_1 + O(1)\ P$	$O(\log n \cdot \lvert C\rvert)\ E_1 + O(\log n \cdot \lvert C\rvert)\ E_2$
Communication	$O(\log n)$	$O(1)$	$O(\log n)$
Bank.Redeem	$O(n)\ P$	$O(n)\ P$	$O(\log n)\ P$

Table 1: Comparison of computation costs (in asymmetric operations) and communication costs for redemption in state-of-the-art divisible e-cash schemes, where n is the redeem value, $|C|$ is the circuit size (in multiplication gates), E_i is exponentiations in $\mathbb{G}_i$, and P is pairings. *We include precomputation costs in Cust.RedeemRequest**.**

at \$4.08/hour or \$0.0425/hour/core [4]. We elide data transfer and storage costs, since they turn out to be negligible in our calculations. According to a 2021 survey [1], the average cost per thousand impressions (CPM) on the most popular ad platforms range from \$3.12 (Google Display Ads), which is the conservative cost figure we use, to \$38.40 (Google Search Ads).

In our experiments, by fixing $\ell = 11$, the most one can spend per payment is 4095 coins, which provides more-than-sufficient granularity for bidding. To spend this amount, the advertiser's non-negligible computation costs are $2 \times 55 = 111$ ms for requesting two batches of 2048 unit-valued coins (Figure 3 center) and $11 \times 210 = 2310$ ms for precomputing the eleven batches to-be-spent (Figure 3 right). The exchange's non-negligible computation cost is 63 ms for redemption (Figure 4 right). The publisher's computation costs are negligible. In total, each impression requires at most 2.48 seconds of CPU time, or 0.69 hours of CPU time per thousand impressions. At \$0.0425/hour/core and \$3.12 CPM, the combined server operating cost of our divisible e-cash scheme is less than \$0.03 CPM, which is less than 1% of ad spend.

On the other hand, state-of-the-art divisible e-cash schemes [22, 36, 37, 84] require two pairings per unit spent for double-spend detection, so 8190 pairings to spend 4095 coins. At 1.5 ms per pairing on our testbed, double-spend detection alone would require an operating cost of 5% of ad spend.

Our divisible e-cash scheme reduces exchange computational costs by roughly 250× compared to existing schemes. The cost of pairings for e-cash validation dominates the exchange's computational costs in both our scheme and in existing schemes. The 33 pairings required in our scheme (three pairings per Groth16 verification) compared to the 8190 pairings required in existing schemes presents a roughly 250× reduction in exchange computational costs. This is favorable, since costs are spread across multiple parties rather than concentrated at the exchange.

7 DISCUSSION

Can't malicious clients steal coins and redeem them for their own gain? Detecting ad fraud, such as botnet or click-farm-generated traffic and publishers displaying "hidden ads," is a challenging problem that exists both today and in our system [53]. Our proposal to route e-cash through clients seems to simplify ad fraud—a malicious client could collect coins from every bidding advertiser, and then either let the coins go to waste or redeem them for their own gain. Fortunately, there are ways to mitigate this.

First, we address the incentives a malicious client may have to redeem coins for themselves. We envision a system where publishers have commercial relationships with the exchange and have sole redemption capabilities. A fitting analogy is Apple's App Store, where developers must pay a fee to enroll in the program and submit apps [2]. In our system, publishers must enroll with the exchange to serve ads and receive payments. Of course, this does not prevent a malicious client from registering as a publisher and directing ads to themselves, a problem that persists in the current advertising ecosystem.

While this prevents a malicious client from redeeming coins (short of enrolling as a publisher), they could still let them expire, wasting advertisers' resources. To mitigate this, the exchange can issue an auction ID to the publisher, which is then bound to the winning advertiser's coins, ensuring only the designated publisher can redeem them. Additionally, the winning advertiser can attach an opaque payment ID to their coins upon payment. The exchange regularly publishes redeemed auction-payment ID pairs, which the advertiser can check against to verify that their payment actually went through. If there are discrepancies, the advertiser can blocklist the malicious client. More details about this mitigation can be found in Appendix D.1.

Although there are many tricks that can help mitigate ad fraud, it is difficult to eradicate completely. We posit that ad fraud in the Protected Audience API, incorporating our private billing solution, is no worse than existing forms of ad fraud. In fact, it may pave the way for innovative detection techniques, since advertisers can establish direct relationships with clients and retain control over their marketing targets.

Can't malicious exchanges overcharge advertisers and underpay publishers? Another concern in today's advertising ecosystem is that centralized ad exchanges can rig auctions to overcharge advertisers and underpay publishers [60]. Given the lack of transparency into today's exchanges, auditing may be the only way to hold them accountable, which can be costly and yet uncomprehensive. While we do not address this issue head-on, our system may ease the audit burden. Our system may obviate much of the need for auditing at the exchange, since auctions are delegated to clients (though this introduces other challenges as discussed). Instead, auditing at the exchange can ensure that advertisers and publishers receive fair exchange rates, i.e., similarly situated parties receive similar exchange rates. See related work (Section 8) for systems that aim to address this issue.

Why isn't there an end-to-end private billing implementation and evaluation on top of the Protected Audience API? Many features of the Protected Audience API are still under development as of this research. Moreover, given that the API is experimental, features are likely to change. Therefore, we have not yet integrated our divisible e-cash scheme for testing with the API. As the API matures, integrating our private billing solution with the API and collecting end-to-end benchmarks will be important future work to understand the end-to-end latency of the combined system.

Does the latency overhead of our divisible e-cash scheme meet the demands of real-world ad delivery? Our divisible e-cash system introduces at most 63 ms of latency to the ad delivery process, comfortably below the industry benchmark of 150 ms. This leaves 87 ms for additional steps (e.g., ad auctioning) outside the scope of our work, which unfortunately prove challenging to measure in practice. While it is difficult to reach a definitive answer without end-to-end benchmarks, we believe the latency overhead is sufficiently low and still there are further tricks to make latency essentially negligible.

For example, considering the high probability of legitimate advertiser payments, the exchange can employ an "optimistic" verification strategy to expedite the process. Rather than check zk-SNARK proofs and double-spends online, the exchange can immediately accept the coins, notify the client, and conduct the checks offline. Alternatively, the exchange could verify the leading-order coins and optimistically accept the remainder. Should any coins later be deemed invalid, the exchange can, for instance, inform the client (via the publisher) to blocklist the cheating advertiser. Although this approach might permit a small degree of double-spending, the trade-off for faster ad deliveries could prove worthwhile.

In any case, even if our system includes additional network trips, ad-tech companies optimize for exactly this by, e.g., moving data centers closer to clients and exchanges, or moving the media itself closer via CDNs [93].

Does our divisible e-cash scheme offer server operating costs low enough for practical deployment? Attaining a server operating cost below 1% of ad spend is a considerable improvement over the previous state-of-the-art. However, it is worth noting that Meta's private ad measurement system, which solves an orthogonal problem to private ad targeting, aims for an ambitious (albeit unrealized) operating cost of 0.1% of ad spend [85]. Given that online advertising is an over half-trillion dollar industry [52], important future work is to further reduce server operating costs while retaining the same privacy guarantees. Using an alternative zk-SNARK backend, such as Spartan [89], should significantly reduce server operating costs with only a modest increase in latency and communication. Moreover, as zk-SNARK technology continues to evolve, our e-cash scheme stands to reap the benefits of these advancements.

What are some ethical considerations of working on targeted advertising? Today's web advertising model is problematic: Studies suggest that ad targeting can fuel discrimination, negatively influence behavior, cause mental distress, and more [81, 86]. It may even bear responsibility for hateful speech and misinformation online [56, 58, 61]. Nevertheless, we feel compelled to develop techniques for private ad targeting for two reasons. First, advertising is what pays for most of the "free" Internet [57, 76], and will almost certainly remain essential to the Internet economy for the foreseeable future. Second, advertising today relies on the large-scale collection and analysis of user preferences. Therefore, this work aims to develop techniques that reconcile the essential role of advertising in funding the tech ecosystem with the critical need for stronger user privacy protections online.

8 RELATED WORK

Private ad targeting. Academic work on private ad targeting stretches back over two decades to the work of Juels [74], which first proposed shifting ad targeting to the client and fetching ads using private information retrieval (PIR) [46]. Many follow-on works have adopted, improved, and extended these ideas: on-device machine learning [71, 96] or locality sensitive hashing [78, 88] for ad targeting; more efficient PIR [66, 78, 88] or batch downloads [96] for ad fetching; homomorphic encryption [66, 96, 103] or anonymous tokens [16, 88] for ad reporting; secure coprocessors and oblivious RAM for running exchanges [16]; and verifiability for ad auctions [13, 83, 104].

Some of these ideas have recently made their way into browsers. Google's Privacy Sandbox [6] is trialing solutions for private ad targeting in Chrome without third-party cookies—TURTLEDOVE [99] and FLEDGE [54] for ad retargeting, and Topics [55] (previously FLoC [98]) for interest-based targeting. Our divisible e-cash scheme offers a private billing solution missing in the ad-retargeting systems. The Brave browser has also deployed its own private ad-targeting system [24]. A recently proposed extension (concurrent with our work) suggests using a scheme analogous to our simple (leaky) e-cash scheme to pay clients for ad interactions [25]. Our divisible e-cash scheme offers a drop-in replacement to fix the same leakage present in their scheme. Major browser vendors have also proposed solutions for the orthogonal problem of private ad-conversion measurement [80, 95, 101].

Electronic cash. Chaum [40] presented the first electronic cash (e-cash) scheme. A vast body of work has since extended e-cash with additional properties, such as traceability [41], auditability [87], small wallet sizes [28], money laundering detection [29], fair exchange [33], and more. These features significantly expand e-cash's utility to a variety of application settings.

For our private ad-retargeting setting, we take a minimalist approach to e-cash design. We tailor our e-cash scheme to take advantage of the setting's unique dynamics (e.g., market incentives), resulting in a scheme that may not be as full-featured as prior schemes, but is relatively simple and more efficient along metrics most critical to our setting.

For example, traceability has been a particularly important feature of e-cash schemes—if a customer spends a coin twice, their identity will be revealed [41]. This property not only discourages double-spending, but also reduces redemption latency: Merchants can confidently release goods to buyers before the bank completes its double-spending check, knowing they can seek recourse if discrepancies are later detected.

However, we find that in our private ad-retargeting setting, traceability is unnecessary. Owing to the efficiency of our divisible e-cash scheme, the exchange can quickly check for any double-spending attempts before informing the client, via the publisher, to display the ad. Moreover, advertisers who attempt to defraud the system by using spent coins are only hurting their own bottom lines: If the exchange rejects their coins, then their ads will remain unseen by potential clients.

E-cash has also inspired the vast bodies of work on anonymous credentials [12, 17, 18, 27, 30–32, 38, 39, 50, 90, 97] and decentralized cryptocurrencies [8, 11, 19, 45, 67, 75, 79]. While all of these tools are relevant to the problem of private billing, our minimalist approach leads us to a different part of the design space. For example, we have not identified the attribution-proving features of anonymous credentials or the decentralization features of cryptocurrencies as strictly necessary to our specific ad-retargeting setting. Still, they may prove useful as new requirements emerge or in adjacent private ad-targeting settings. For instance, cryptocurrencies may eliminate the reliance on a trusted, centralized exchange: A client can run a smart contract to hold a fair auction, where advertisers deposit collateral and the winner's assets are transferred to the publisher.

Instead, we find that divisible e-cash schemes are best positioned to meet the utility and efficiency demands of private billing in private ad-retargeting systems. Starting from a long line of work on divisible e-cash [15, 22, 34–37, 82, 84], our divisible e-cash scheme uses modern zk-SNARKs [68, 89] and a symmetric-key-based CPRF [63], which leads to concrete efficiency improvements over prior schemes that rely on Groth-Sahai proofs [69, 70] and pairing-based CPRFs [21]. These instantiations enable optimizations along measures most critical to our application—latency, client costs, and server operating costs—whereas prior schemes focus optimizations on spend latency (i.e., zero-knowledge proof generation) and communication. Again, our application setting also allows us to focus on core e-cash properties—unforgeability and unlinkability—whereas prior schemes support richer properties, such as traceability.

9 CONCLUSION

Recent private ad-retargeting systems, such as the Protected Audience API, allow a client to receive retargeted ads privately: the exchange does not learn which advertisers and publishers the client has visited, and advertisers and publishers do not learn which of the other the client has visited. This work identifies a missing piece in such systems—private billing for ad impressions—and proposes a straightforward fix—e-cash. Prior e-cash schemes are not well-suited to this setting, so our key contribution is a new divisible e-cash scheme that achieves strong privacy guarantees and meets crucial efficiency requirements: low latency, low client costs, and low server operating costs. We hope the Protected Audience API will upstream our proposed fix and that this work will inform the design of future private ad-targeting systems.

ACKNOWLEDGMENTS

We thank our shepherd, Niklas Carlsson, and the anonymous reviewers for their valuable feedback. This work was funded in part by the NSF under Award CNS-2054869 and a Graduate Research Fellowship, the Simons Foundation, UBRI, NTT Research, Capital One, Facebook, Google, Mozilla, NASDAQ, and MIT's FinTech@CSAIL Initiative.

REFERENCES

[1] 2021. Online Advertising Costs in 2021. https://www.topdraw.com/insights/is-online-advertising-expensive/.
[2] 2023. Apple Developer Program. https://developer.apple.com/programs/.
[3] 2024. Protected Audience API Overview. https://developers.google.com/privacy-sandbox/relevance/protected-audience.
[4] Accessed: 2022-11-25. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.
[5] Accessed: 2022-11-25. BLAKE3. https://github.com/BLAKE3-team/BLAKE3.
[6] Accessed: 2022-11-25. Building a more private, open web. https://privacysandbox.com/.
[7] Accessed: 2022-11-25. Challenge Bypass using the Ristretto group. https://github.com/brave-intl/challenge-bypass-ristretto.
[8] Accessed: 2022-11-25. Monero. https://www.getmonero.org.
[9] Accessed: 2022-11-25. Referer. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer.
[10] Accessed: 2022-11-25. What is click fraud? | How click bots work. https://www.cloudflare.com/learning/bots/what-is-click-fraud/.
[11] Accessed: 2022-11-25. Zcash. https://z.cash.
[12] Norio Akagi, Yoshifumi Manabe, and Tatsuaki Okamoto. 2008. An Efficient Anonymous Credential System. In *FC*.
[13] Sebastian Angel and Michael Walfish. 2013. Verifiable auctions for online ad exchanges. In *SIGCOMM*.
[14] arkworks contributors. 2022. arkworks zkSNARK ecosystem. https://arkworks.rs.
[15] Man Ho Au, Willy Susilo, and Yi Mu. 2008. Practical Anonymous Divisible E-Cash from Bounded Accumulators. In *FC*.
[16] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. 2012. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *IEEE S&P*.
[17] Foteini Baldimtsi and Anna Lysyanskaya. 2013. Anonymous Credentials Light. In *CCS*.
[18] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. 2009. Randomizable Proofs and Delegatable Anonymous Credentials. In *CRYPTO*.
[19] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE S&P*.
[20] Dan Boneh and Victor Shoup. 2020. A graduate course in applied cryptography. *Draft 0.5* (2020).
[21] Dan Boneh and Brent Waters. 2013. Constrained Pseudorandom Functions and Their Applications. In *ASIACRYPT*.
[22] Florian Bourse, David Pointcheval, and Olivier Sanders. 2019. Divisible E-Cash from Constrained Pseudo-Random Functions. In *ASIACRYPT*.
[23] Sean Bowe. 2017. BLS12-381: New zk-SNARK Elliptic Curve Construction. https://electriccoin.co/blog/new-snark-curve/.
[24] Brave. 2020. An Introduction to Brave's In-Browser Ads. https://brave.com/intro-to-brave-ads/.
[25] Brave. 2022. Security and privacy model for ad confirmations. https://github.com/brave/brave-browser/wiki/Security-and-privacy-model-for-ad-confirmations.
[26] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. 2016. An Empirical Study of Web Cookies. In *WWW*.
[27] Jan Camenisch and Els Van Herreweghen. 2002. Design and implementation of the *idemix* anonymous credential system.
[28] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. 2005. Compact E-Cash. In *EUROCRYPT*.
[29] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. 2006. Balancing Accountability and Privacy Using E-Cash (Extended Abstract). In *SCN*.
[30] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO*.
[31] Jan Camenisch and Anna Lysyanskaya. 2002. A Signature Scheme with Efficient Protocols. In *SCN*.
[32] Jan Camenisch and Anna Lysyanskaya. 2004. Signature Schemes and Anonymous Credentials from Bilinear Maps. In *CRYPTO*.
[33] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. 2007. Endorsed E-Cash. In *IEEE S&P*.
[34] Sébastien Canard and Aline Gouget. 2007. Divisible E-Cash Systems Can Be Truly Anonymous. In *EUROCRYPT*.
[35] Sébastien Canard and Aline Gouget. 2010. Multiple Denominations in E-cash with Compact Transaction Data. In *FC*.
[36] Sébastien Canard, David Pointcheval, Olivier Sanders, and Jacques Traoré. 2015. Divisible E-Cash Made Practical. In *PKC*.
[37] Sébastien Canard, David Pointcheval, Olivier Sanders, and Jacques Traoré. 2015. Scalable Divisible E-cash. In *ACNS*.
[38] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. 2014. Algebraic MACs and Keyed-Verification Anonymous Credentials. In *CCS*.
[39] Melissa Chase, Trevor Perrin, and Greg Zaverucha. 2020. The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption. In *CCS*.
[40] David Chaum. 1982. Blind Signatures for Untraceable Payments. In *CRYPTO*.
[41] David Chaum, Amos Fiat, and Moni Naor. 1988. Untraceable Electronic Cash. In *CRYPTO*.
[42] David Chaum and Torben P. Pedersen. 1992. Wallet Databases with Observers. In *CRYPTO*.
[43] Brian X. Chen. 2021. The Battle for Digital Privacy Is Reshaping the Internet. https://www.nytimes.com/2021/09/16/technology/digital-privacy.html.
[44] Brian X. Chen and Kate Conger. 2021. What the Privacy Battle Upending the Internet Means for You. https://www.nytimes.com/2021/09/16/technology/personaltech/internet-privacy-chrome-safari.html.
[45] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. 2017. Decentralized Anonymous Micropayments. In *EUROCRYPT*.
[46] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (1998), 965–981.
[47] Clearcode. 2021. The AdTech Book. https://adtechbook.clearcode.cc/.
[48] Bennett Cyphers and Gennie Gebhart. 2019. Behind the One-Way Mirror: A Deep Dive Into the Technology of Corporate Surveillance. https://www.eff.org/wp/behind-the-one-way-mirror.
[49] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. 2022. Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups. https://datatracker.ietf.org/doc/pdf/draft-irtf-cfrg-voprf-15.
[50] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. 2018. Privacy Pass: Bypassing Internet Challenges Anonymously. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 164–180.
[51] Henry de Valence. 2018. The Ristretto Group. https://ristretto.group/.
[52] Statista Research Department. 2023. Digital advertising spending worldwide from 2021 to 2026. https://www.statista.com/statistics/237974/online-advertising-spending-worldwide/.
[53] Artyom Dogtiev. 2022. Ad Fraud Statistics. https://www.businessofapps.com/ads/ad-fraud/research/ad-fraud-statistics/.
[54] Sam Dutton. 2022. FLEDGE API. https://developer.chrome.com/docs/privacy-sandbox/fledge/.
[55] Sam Dutton. 2022. The Topics API. https://developer.chrome.com/docs/privacy-sandbox/topics/.
[56] Gilad Edelman. 2020. Follow the Money: How Digital Ads Subsidize the Worst of the Web. https://www.wired.com/story/how-digital-ads-subsidize-worst-web/.
[57] Gilad Edelman. 2020. Why Don't We Just Ban Targeted Advertising? https://www.wired.com/story/why-dont-we-just-ban-targeted-advertising/.
[58] Laura Edelson, Tobias Lauinger, and Damon McCoy. 2020. A Security Analysis of the Facebook Ad Library. In *IEEE S&P*.
[59] Dominic Farolino. 2022. Fenced frame. https://wicg.github.io/fenced-frame/.
[60] Sara Fischer and Ina Fried. 2022. Lawsuit: Google, Facebook execs conspired to manipulate ad auctions. https://www.axios.com/2022/01/18/google-facebook-conspired-manipulate-ad-auctions-states-lawsuit.
[61] Jeff Gary and Ashkan Soltani. 2019. First Things First: Online Advertising Practices and Their Effects on Platform Speech. https://knightcolumbia.org/content/first-things-first-online-advertising-practices-and-their-effects-on-platform-speech.
[62] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge University Press.
[63] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1984. How to Construct Random Functions (Extended Abstract). In *FOCS*.
[64] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM J. Comput.* 17, 2 (1988), 281–308.
[65] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Security*.
[66] Matthew Green, Watson Ladd, and Ian Miers. 2016. A Protocol for Privately Reporting Ad Impressions at Scale. In *CCS*.
[67] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *CCS*.
[68] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT*.
[69] Jens Groth and Amit Sahai. 2008. Efficient Non-interactive Proof Systems for Bilinear Groups. In *EUROCRYPT*.

[70] Jens Groth and Amit Sahai. 2012. Efficient Noninteractive Proof Systems for Bilinear Groups. *SIAM J. Comput.* 41, 5 (2012), 1193–1232.
[71] Saikat Guha, Bin Cheng, and Paul Francis. 2011. Privad: Practical Privacy in Online Advertising. In *NSDI*.
[72] Ryan Henry. 2014. *Efficient Zero-Knowledge Proofs and Applications.* Ph. D. Dissertation. University of Waterloo, Ontario, Canada.
[73] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. 2014. Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model. In *ASIACRYPT*.
[74] Ari Juels. 2001. Targeted Advertising…And Privacy Too. In *CT-RSA*.
[75] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE S&P*.
[76] Rani Molla. 2019. The cost of an ad-free Internet: $35 more per month. https://www.vox.com/recode/2019/6/24/18715421/internet-free-data-ads-cost.
[77] Mozilla. 2021. Enhanced Tracking Protection in Firefox for desktop. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop.
[78] Muhammad Haris Mughees, Gonçalo Pestana, Alex Davidson, and Benjamin Livshits. 2021. PrivateFetch: Scalable Catalog Delivery in Privacy-Preserving Advertising. *arXiv preprint arXiv:2109.08189* (2021).
[79] Satoshi Nakamoto. 2008. A peer-to-peer electronic cash system.
[80] Maud Nalpas, Sam Dutton, and Alexandra White. 2022. Attribution Reporting. https://developer.chrome.com/en/docs/privacy-sandbox/attribution-reporting/.
[81] Rae Nudson. 2020. When targeted ads feel a little too targeted. https://www.vox.com/the-goods/2020/4/9/21204425/targeted-ads-fertility-eating-disorder-coronavirus.
[82] Tatsuaki Okamoto and Kazuo Ohta. 1991. Universal Electronic Cash. In *CRYPTO*.
[83] Gonçalo Pestana, Iñigo Querejeta-Azurmendi, Panagiotis Papadopoulos, and Benjamin Livshits. 2021. THEMIS: A Decentralized Privacy-Preserving Ad Platform with Reporting Integrity. *arXiv preprint arXiv:2106.01940* (2021).
[84] David Pointcheval, Olivier Sanders, and Jacques Traoré. 2017. Cut Down the Tree to Achieve Constant Complexity in Divisible E-cash. In *PKC*.
[85] James Reyes and Sanjay Saravanan. 2022. Building the next generation of digital advertising with MPC. https://iacr.org/submit/files/slides/2022/rwc/rwc2022/104/slides.pdf.
[86] Emma Roth. 2021. Facebook and Instagram will delete 'sensitive' ad targeting groups linked to race, politics. https://www.theverge.com/2021/11/9/22773038/meta-detailed-targeting-ads-facebook-instagram-messenger.
[87] Tomas Sander and Amnon Ta-Shma. 1999. Auditable, Anonymous Electronic Cash Extended Abstract. In *CRYPTO*.
[88] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. 2021. AdVeil: A Private Targeted-Advertising Ecosystem. *IACR Cryptol. ePrint Arch.* (2021).
[89] Srinath T. V. Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *CRYPTO*.
[90] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. 2019. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. In *NDSS*.
[91] Tristram Southey. 2023. Protected Audience API: Our new name for FLEDGE. https://privacysandbox.com/news/protected-audience-api-our-new-name-for-fledge.
[92] John Teague. 2022. Page Weight. https://almanac.httparchive.org/en/2021/page-weight.
[93] Headerbidding Team. 2023. Addressing Latency Issue in Digital Advertising. https://headerbidding.co/latency-issue-digital-advertising/.
[94] David Temkin. 2021. Charting a course towards a more privacy-first web. https://blog.google/products/ads-commerce/a-more-privacy-first-web/.
[95] Martin Thomson. 2022. Privacy Preserving Attribution for Advertising. https://blog.mozilla.org/en/mozilla/privacy-preserving-attribution-for-advertising/.
[96] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. 2010. Adnostic: Privacy Preserving Targeted Advertising. In *NDSS*.
[97] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. 2010. BLAC: Revoking Repeatedly Misbehaving Anonymous Users without Relying on TTPs. *ACM Trans. Inf. Syst. Secur.* 13, 4 (2010), 39:1–39:33.
[98] WICG. 2021. Federated Learning of Cohorts (FLoC). https://github.com/WICG/floc.
[99] WICG. 2021. TURTLEDOVE. https://github.com/WICG/turtledove.
[100] John Wilander. 2017. Intelligent Tracking Prevention. https://webkit.org/blog/7675/intelligent-tracking-prevention/.
[101] John Wilander. 2021. Introducing Private Click Measurement, PCM. https://webkit.org/blog/11529/introducing-private-click-measurement-pcm/.
[102] Zcash. 2017. What is Jubjub? https://bitzecbzc.github.io/technology/jubjub/index.html.
[103] Ke Zhong, Yiping Ma, and Sebastian Angel. 2022. Ibex: Privacy-preserving Ad Conversion Tracking and Bidding. In *CCS*.
[104] Ke Zhong, Yiping Ma, Yifeng Mao, and Sebastian Angel. 2022. Addax: A fast, private, and accountable ad exchange infrastructure. *IACR Cryptol. ePrint Arch.* (2022), 1299.

A SIMPLE E-CASH

We present our simple e-cash scheme in Construction A.

B DIVISIBLE E-CASH SECURITY ANALYSIS

We now analyze the security of our divisible e-cash scheme presented in Construction 1.

Lemma B.1 (Unforgeability). *Construction 1 is an unforgeable e-cash scheme assuming the signature scheme is EUF-CMA-secure, the commitment scheme is binding, the zk-SNARK proof system is knowledge-sound, and the Merkle tree is built from a collision-resistant hash function.*

Proof. Let $\mathcal{A}$ be an adversary that participates in the unforgeability game from Definition 4.2. The adversary outputs a sequence of issue and redeem requests, and wants to satisfy $\mathsf{bal} < 0$ at the end of the game. We argue that this can only happen with negligible probability by stepping through a sequence of games, where Game 0 is the original unforgeability game.

Game 1. Recall that the adversary generates redeem requests with parameters cs and n. Every cs is a set of tuples $(i, k_{\mathcal{S}}, \phi, \pi)$, where π is a zk-SNARK proof of knowledge of some witness $(\sigma, \mathsf{root}^*, \mathsf{com}, \mathsf{open}, \pi_{\mathsf{path}})$.

In this game, in addition to the winning condition, the adversary must also output valid SNARK witnesses

$$\{(\sigma_j, \mathsf{root}_j^*, \mathsf{com}_j, \mathsf{open}_j, \pi_{\mathsf{path},j})\}_{j \in [Q]},$$

one witness for every element of cs over all redeem requests. Each witness corresponds to a tuple $(i_j, k_{\mathcal{S},j}, \phi_j, \pi_j)$ in some cs in some redeem request. Because the zk-SNARK is knowledge-sound, there is an extractor that lets us construct an adversary $\mathcal{A}'$ from $\mathcal{A}$ that satisfies this requirement with about the same probability as $\mathcal{A}$'s winning probability in Game 0.

Game 2. In this game, we strengthen the winning condition and require that in order to win, the adversary must further ensure that for all $j \in [Q]$, redeem request number j is made where com_j is equal to a string req used in an earlier issue request. Because the signature scheme is EUF-CMA-secure, this additional constraint only negligibly changes the adversary's winning probability. Recall that every issue request causes 2^ℓ to be added to the variable bal.

Game 3. In this game, we further strengthen the winning condition and require that in order to win, the adversary must further ensure that for every pair $u, v \in [Q]$, if $\mathsf{com}_u = \mathsf{com}_v$, then $\mathsf{root}_u^* = \mathsf{root}_v^*$. Because the commitment scheme is binding, this additional constraint only negligibly changes the adversary's winning probability.

Game 4. We strengthen the winning condition one last time and require that in order to win, the adversary must further ensure that for every pair $u, v \in [Q]$, if $\mathsf{root}_u^* = \mathsf{root}_v^*$ and $\pi_{\mathsf{path},u}$ and $\pi_{\mathsf{path},v}$ either leads to the same node in the tree or leads to two nodes where one dominates the other, then the quantities

$$\mathbf{y}_u \leftarrow \mathsf{CPRF.Expand}(k_{\mathcal{S},u},\ \ell - i_u) \text{ and}$$
$$\mathbf{y}_v \leftarrow \mathsf{CPRF.Expand}(k_{\mathcal{S},v},\ \ell - i_v)$$

computed in Bank.Redeem satisfy $\mathbf{y}_v \subseteq \mathbf{y}_u$. Because the Merkle tree is built from a collision-resistant hash function, this additional

Construction A. Simple e-cash scheme. Let parameter $\ell \in \mathbb{N}$ be the max log-denomination.

```
Bank.Setup(1^λ) → (st, pp).
  • Choose a group $\mathbb{G}$ of prime order $q$ with generator $g$, a hash function $H : \{0,1\}^* \to \mathbb{G}$.
  • For $i \in [0, \ell]$:
      – Sample signing key $\mathsf{sk}_i \xleftarrow{\$} \mathbb{Z}_q$.
      – Compute public key $\mathsf{pk}_i \leftarrow g_i^{\mathsf{sk}}$.
  • Set spent serial numbers $\mathsf{dsl} \leftarrow \varnothing$.
  • Set state $\mathsf{st} \leftarrow (\mathsf{sk}_0, \dots, \mathsf{sk}_\ell, \mathsf{dsl})$.
  • Set public parameters
        $\mathsf{pp} \leftarrow (1^\lambda, q, \mathbb{G}, g, H, k, \mathsf{pk}_0, \dots, \mathsf{pk}_\ell)$.
  • Output $(\mathsf{st}, \mathsf{pp})$.
Bank.Issue(st, req, n) → (st, rsp).
  • If $n \notin \{2^0, 2^1, \dots, 2^\ell\}$, then output $(\mathsf{st}, \perp)$.
  • Parse state st as $(\mathsf{sk}_0, \dots, \mathsf{sk}_\ell, \mathsf{dsl})$.
  • Parse request req as $\tilde{\tau}$.
  • Set log-denomination $i \leftarrow \log(n)$.
  • Compute blind signature $\tilde{\sigma} \leftarrow \tilde{\tau}^{\mathsf{sk}_i}$.
  • Compute proof $\pi \leftarrow \mathsf{DLEQ}(g, \mathsf{pk}_i, \tilde{\tau}, \tilde{\sigma})$.
  • Set response $\mathsf{rsp} \leftarrow (\tilde{\sigma}, \pi)$.
  • Output $(\mathsf{st}, \mathsf{rsp})$.
Bank.Redeem(st, cs, n) → (st, b).
  • If $n \notin [0, 2^{\ell+1})$, then output $(\mathsf{st}, 0)$.
  • Parse state st as $(\mathsf{sk}_0, \dots, \mathsf{sk}_\ell, \mathsf{dsl})$.
  • Set initial state $\mathsf{st}_0 \leftarrow \mathsf{st}$.
  • Set amount $m \leftarrow 0$.
  • For $(i, x, \sigma) \in \mathsf{cs}$:
      – If $x \in \mathsf{dsl}$, then output $(\mathsf{st}_0, 0)$.
      – If $\sigma = H(x)^{\mathsf{sk}_i}$, then add $x$ to dsl and set $m \leftarrow m + 2^i$.
  • If $n = m$, then output $(\mathsf{st}, 1)$. Otherwise, output $(\mathsf{st}_0, 0)$.
Cust.Setup(1^λ) → wlt.
  • Set wallet $\mathsf{wlt} \leftarrow \varnothing$.
  • Output wlt.
Cust.IssueRequest(wlt, n) → (st, req).
  • Set log-denomination $i \leftarrow \log(n)$.
  • Sample serial number $x \xleftarrow{\$} \{0,1\}^\lambda$.
  • Sample blinding factor $r \xleftarrow{\$} \mathbb{Z}_q$.
  • Compute blinded coin $\tilde{\tau} \leftarrow H(x)^r$.
  • Set request $\mathsf{req} \leftarrow \tilde{\tau}$.
  • Add $(i, x, r)$ to wlt.
  • Output $(\mathsf{wlt}, \mathsf{req})$.
Cust.IssueProcess(wlt, rsp) → wlt.
  • Parse response rsp as $(\tilde{\sigma}, \pi)$, where $\pi$ is a proof for $\mathsf{DLEQ}(g, \mathsf{pk}_i, \tilde{\tau}, \tilde{\sigma})$.
  • If proof $\pi$ is not valid, then output wlt.
  • Find $(i, x, r) \in \mathsf{wlt}$, where $\tilde{\tau} = H(x)^r$. If no such coin exists, then output wlt.
  • Compute signed coin $\sigma \leftarrow \tilde{\sigma}^{1/r}$.
  • Update the coin to $(i, x, \sigma)$ in wlt.
  • Output wlt.
Cust.RedeemRequest(wlt, n) → (wlt, cs).
  • Set initial wallet $\mathsf{wlt}_0 \leftarrow \mathsf{wlt}$.
  • Set coins $\mathsf{cs} \leftarrow \varnothing$.
  • Let $n_{\ell-1} || \cdots || n_0$ be the binary representation of $n$.
  • For $i \in [0, \ell - 1]$ such that $n_i = 1$:
      – Find and remove $(i, x, \sigma)$ from wlt and add it to cs. If no such coin exists, then output $(\mathsf{wlt}_0, \varnothing)$.
  • Output $(\mathsf{wlt}, \mathsf{cs})$.
```

constraint only negligibly changes the adversary's winning probability.

Completing the proof. We now argue that the adversary cannot win Game 4. Indeed, each issue request from $\mathcal{A}$ adds 2^ℓ to the variable bal. Every redeem request from $\mathcal{A}$ spends a *disjoint* set of leaves from one of the trees created in an issue request, where each tree has 2^ℓ leaves. Therefore, at the end of the game, we must have $\mathsf{bal} \geq 0$, meaning that the adversary cannot win.

Finally, because the winning probability in Game 4 is zero, and the winning probability in Game 0 is at most negligibly different than in Game 4, it follows that the winning probability in Game 0 is negligible, as required. □

Lemma B.2 (Unlinkability). *Construction 1 is an unlinkable e-cash scheme assuming the commitment scheme is hiding, the CPRF is secure and key-pseudorandom, the Merkle tree hash function is a hiding commitment, and the zk-SNARK proof system is zero-knowledge.*

Proof. Let $\mathcal{A}$ be an adversary that participates in the unlinkability game from Definition 4.3. The adversary outputs a sequence of issue and redeem requests with a challenge request, and wants to satisfy $b = b'$ at the end of the game. We argue that this can only happen with negligible advantage by stepping through a sequence of games, where Game 0 is the original unlinkability game.

In what follows, let Adv_i be $\mathcal{A}$'s advantage in Game i, let $\mathsf{Adv}^{\mathsf{zk}}$ be $\mathcal{A}$'s advantage against the zero-knowledge property of the zk-SNARK, let $\mathsf{Adv}^{\mathsf{hid}}$ be $\mathcal{A}$'s advantage against the hiding property of the commitment, let $\mathsf{Adv}^{\mathsf{kp}}$ be $\mathcal{A}$'s advantage against the key-pseudorandomness property of the CPRF, let $\mathsf{Adv}^{\mathsf{mt}}$ be $\mathcal{A}$'s advantage against the hiding commitment property of the Merkle tree hash function.

Game 1. In this game, we modify the challenger to simulate the zk-SNARKs. Instead of computing $(\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{SNARK.KeyGen}(1^\lambda, C)$ in Bank.Setup, the challenger invokes a simulator $\mathcal{S}$ to obtain $(\mathsf{pk}_\Pi, \mathsf{vk}_\Pi, \tau) \leftarrow \mathcal{S}(1^\lambda, C)$, where τ is some trapdoor information. For each of at most Q_r redeem requests and for each of the at most ℓ zk-SNARK proofs, instead of building the proof as $\pi \leftarrow \mathsf{SNARK.Prove}(\mathsf{pk}_\Pi, \phi, w)$ in

Cust.RedeemRequest, the challenger again invokes the simulator to obtain $\pi \leftarrow \mathcal{S}(\mathsf{pk}_\Pi, \phi, \tau)$. Because the zk-SNARK is zero-knowledge, we have $|\mathsf{Adv}_1 - \mathsf{Adv}_0| \leq \ell \cdot Q_r \cdot \mathsf{Adv}^{\mathsf{zk}}$.

Game 2. In this game, we modify the challenger to replace commitments with random elements from the commitment space C. For each of at most Q_i issue requests, instead of computing $(\mathsf{com}, \mathsf{open}) \leftarrow \mathsf{Commit}(\mathsf{root}^*)$ in Cust.IssueRequest, the challenger samples $\mathsf{com} \xleftarrow{\$} C$. Because the commitment scheme is hiding, we have $|\mathsf{Adv}_2 - \mathsf{Adv}_1| \leq Q_i \cdot \mathsf{Adv}^{\mathsf{hid}}$.

Game 3. In this game, we modify the challenger to replace constrained keys with random strings and zk-SNARK statements ϕ as Merkle hashes over leaves generated by expanding the now-random constrained keys.

For each of at most Q_r issue requests and for each of at most ℓ subtrees, instead of computing $k_{\mathcal{S}} \leftarrow \mathsf{CPRF.Constrain}(k, \mathcal{S})$ and sending $\phi = \mathsf{Merkle.Hash}(\mathsf{CPRF.Expand}(k_{\mathcal{S}}, \ell - i))$ in Cust.RedeemRequest, the challenger samples $k_r \xleftarrow{\$} \{0,1\}^\lambda$ and sends $\phi_r \leftarrow \mathsf{Merkle.Hash}(\mathsf{CPRF.Expand}(k_r, \ell - i))$. Because the CPRF is key-pseudorandom and the Merkle tree hash function is a hiding commitment, we have $|\mathsf{Adv}_3 - \mathsf{Adv}_2| \leq \ell \cdot Q_r \cdot (\mathsf{Adv}^{\mathsf{kp}} + \mathsf{Adv}^{\mathsf{mt}})$.

Completing the proof. We now argue that the adversary has no advantage in Game 3. Indeed, all of the responses provided to $\mathcal{A}$ are independent of b, so $\mathsf{Adv}_3 = 0$. By summing over $\mathcal{A}$'s advantages in the hybrid games, we can bound $\mathsf{Adv}_0 \leq \ell \cdot Q_r \cdot (\mathsf{Adv}^{\mathsf{zk}} + \mathsf{Adv}^{\mathsf{kp}} + \mathsf{Adv}^{\mathsf{mt}}) + Q_i \cdot \mathsf{Adv}^{\mathsf{hid}}$, which is negligible, as required. □

C OPTIMIZED DIVISIBLE E-CASH

We present our optimized divisible e-cash scheme in Construction B (its differences with Construction 1 are highlighted) and prove its security.

Lemma C.1 (Unforgeability). *Construction B is an unforgeable e-cash scheme assuming the signature scheme is EUF-CMA-secure, the commitment scheme is binding, the zk-SNARK proof system is knowledge-sound, the Merkle tree is built from a collision-resistant hash function, and the CPRF PRG is collision-resistant.*

Proof. The proof is essentially the same as the proof for Lemma B.1, except there is an additional game in which we strengthen the winning condition as follows: In order to win, the adversary must generate a collision on the leftmost leaf of a redeemed subtree. Because the CPRF PRG is collision-resistant, this additional constraint only negligibly changes the adversary's winning probability. □

Lemma C.2 (Unlinkability). *Construction B is an unlinkable e-cash scheme assuming the commitment scheme is hiding, the CPRF is secure and key-pseudorandom, the Merkle tree hash function is a hiding commitment, and the zk-SNARK proof system is zero-knowledge.*

Proof. The proof is essentially the same as the proof for Lemma B.2. □

D PROTECTED AUDIENCE API WITH PRIVATE BILLING

Here, we walk through how private billing with e-cash maps onto Protected Audience (Figure 5).

Step 0-A: Advertisers obtain coins from exchange. Advertisers (each indexed by $i \in [k]$) and the exchange begin by running the client and server setups, respectively, for the e-cash scheme. Advertisers trade in hard currency (USD_i) to the exchange for electronic coins (stored in updates to wallet wlt_i) to later pay publishers for ad impressions. (Alternatively, advertisers can pay hard currency to the exchange at the end of a billing period.)

For the divisible e-cash scheme, advertisers will also precompute zk-SNARK proofs offline. If an advertiser knows the distribution of values it will spend, it can precompute proofs accordingly to save on precomputation costs (instead of naïvely precomputing a proof for every leaf of every tree in its wallet).

Step 0-B: Client visits advertiser websites. When a client visits an advertiser's website and performs certain actions, such as placing an item in a shopping cart, the advertiser invokes a Protected Audience JavaScript API to store a callback URL (cbURL_i) on the client's browser. The client's browser uses this callback to later run ad auctions.

Step 1: Client visits publisher website. When the client visits a publisher's website, the publisher invokes a Protected Audience JavaScript API to indicate its desire to serve a retargeted ad on its page.

Step 2: Client runs ad auction. The client's browser initiates an ad auction (e.g., a first-price, open-bid auction) by calling the stored callback URLs for the advertisers it has visited. Each callback returns the corresponding advertiser's bid (bid_i) for the impression. The client's browser runs the auction (argmax for a first-price auction) and notifies the winning advertiser (win to advertiser j) of the clearing price for the impression (implicit in a first-price auction).

Step 3: Winning advertiser sends ad and coins to client. The winning advertiser sends the URL of its ad creative (adURL), which is used to render the ad, to the client's browser, along with coins (cs) amounting to the clearing price.

Step 4: Client and publisher forward coins to exchange. The client's browser forwards the coins to the publisher, who forwards them to the exchange.

Step 5: Exchange checks validity of coins. If the coins are valid and have not been double-spent, then the exchange accepts ($b = 1$), sends the publisher hard currency (USD) for the value of the redeemed coins, and notifies the client of acceptance through the publisher. (Alternatively, the exchange can pay hard currency to the publisher at the end of a billing period.) Otherwise, the exchange sends rejects ($b = 0$) and the client's browser retries with the next winner of the auction.

Step 6: Client displays ad. If the exchange accepts, then the client's browser displays the ad on the publisher's website in an isolated "fenced frame" [59], which prevents the publisher from learning who the winning advertiser is.

Construction B. Divisible e-cash scheme with collision-resistant PRG optimization. Let parameter $\ell \in \mathbb{N}$ be the height of a full tree.

$\underline{\mathsf{Bank.Setup}(1^\lambda) \to (\mathsf{st}, \mathsf{pp}).}$

- Generate signing key pair $(\mathsf{sk}_\Sigma, \mathsf{pk}_\Sigma) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$.
- Generate proving and verifying key $(\mathsf{pk}_\Pi, \mathsf{vk}_\Pi) \leftarrow \mathsf{SNARK.KeyGen}(1^\lambda, C)$, where the coin validity circuit C is for the following relation: *"Given the leftmost leaf of a subtree* leaf *(the statement), I know a signature σ, commitment* com*, opening information* open*, full root* root**, and Merkle proof* π_{path} *(the witness) such that:*
  - *the signature is on the commitment to the full root:* $\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, \mathsf{com}) = 1$ *with server public key* pk,
  - *the commitment opens to the full root:* $\mathsf{VerifyCom}(\mathsf{root}^*, \mathsf{com}, \mathsf{open}) = 1$*, and*
  - *the Merkle proof is a valid leaf membership proof:* $\mathsf{Merkle.Verify}(\mathsf{leaf}, \mathsf{root}^*, \pi_{\mathsf{path}}) = 1$."
- Set double-spend list $\mathsf{dsl} \leftarrow \emptyset$.
- Set state $\mathsf{st} \leftarrow (\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Set public parameters $\mathsf{pp} \leftarrow (1^\lambda, \mathsf{pk}_\Sigma, \mathsf{pk}_\Pi, \mathsf{vk}_\Pi)$.
- Output $(\mathsf{st}, \mathsf{pp})$.

$\underline{\mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}, n) \to (\mathsf{st}, \mathsf{rsp}).}$

- If $n \neq 2^\ell$, then output $(\mathsf{st}, \perp)$.
- Parse st as $(\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Parse req as com.
- Compute signature $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_\Sigma, \mathsf{com})$.
- Set response $\mathsf{rsp} \leftarrow \sigma$.
- Output $(\mathsf{st}, \mathsf{rsp})$.

$\underline{\mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, n) \to (\mathsf{st}, b).}$

- If $n \notin [0, 2^{\ell+1})$, then output $(\mathsf{st}, 0)$.
- Parse state st as $(\mathsf{sk}_\Sigma, \mathsf{dsl})$.
- Set initial state $\mathsf{st}_0 \leftarrow \mathsf{st}$.
- Set amount $m \leftarrow 0$.
- For $(i, k_{\mathcal{S}}, \phi, \pi)$ in cs:
  - Compute $\mathbf{y} \leftarrow \mathsf{CPRF.Expand}(k_{\mathcal{S}}, \ell - i)$.
  - If $\mathsf{y} \cap \mathsf{dsl} \neq \emptyset$ or $\mathbf{y}[0] \neq \phi$ or $\mathsf{SNARK.Verify}(\mathsf{vk}_\pi, \phi, \pi) = 0$, then output $(\mathsf{st}_0, 0)$.
  - Add $\mathbf{y}$ to dsl in st and set $m \leftarrow m + 2^{\ell - i}$.
- If $n = m$, then output $(\mathsf{st}, 1)$. Otherwise, output $(\mathsf{st}_0, 0)$.

$\underline{\mathsf{Cust.Setup}(1^\lambda) \to \mathsf{wlt}.}$

- Set wallet $\mathsf{wlt} \leftarrow \emptyset$.
- Output wlt.

$\underline{\mathsf{Cust.IssueRequest}(\mathsf{wlt}, n) \to (\mathsf{wlt}, \mathsf{req}).}$

- Generate principal key $k \leftarrow \mathsf{CPRF.KeyGen}(1^\lambda)$.
- Compute full tree leaves $\mathbf{y} \leftarrow \mathsf{CPRF.Expand}(k, \ell)$.
- Compute full root $\mathsf{root}^* \leftarrow \mathsf{Merkle.Hash}(\mathbf{y})$ and store the full tree tree.
- Compute commitment $(\mathsf{com}, \mathsf{open}) \leftarrow \mathsf{Commit}(\mathsf{root}^*)$.
- Set request $\mathsf{req} \leftarrow \mathsf{com}$.
- Add $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \perp)$ to wlt.
- Output $(\mathsf{wlt}, \mathsf{req})$.

$\underline{\mathsf{Cust.IssueProcess}(\mathsf{wlt}, \mathsf{rsp}) \to \mathsf{wlt}.}$

- Parse response rsp as σ.
- Find $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \perp) \in \mathsf{wlt}$ such that $\mathsf{Sig.Verify}(\mathsf{pk}_\Sigma, \sigma, \mathsf{com}) = 1$ and update $\perp$ to σ.
- Output wlt.

$\underline{\mathsf{Cust.RedeemRequest}(\mathsf{wlt}, n) \to (\mathsf{wlt}, \mathsf{cs}).}$

- Set initial wallet $\mathsf{wlt}_0 \leftarrow \mathsf{wlt}$.
- Set coins $\mathsf{cs} \leftarrow \emptyset$.
- Let $n_\ell || \cdots || n_0$ be the binary representation of n.
- For $i \in [0, \ell]$ such that $n_i = 1$:
  - Find $(k, \mathsf{root}^*, \mathsf{tree}, \mathsf{com}, \mathsf{open}, \sigma) \in \mathsf{wlt}$ such that tree contains an unredeemed subtree with leftmost leaf leaf covering 2^i leaves with labels $\mathcal{S}$ and mark the subtree as redeemed. If no such subtree exists, then output $(\mathsf{wlt}_0, \emptyset)$.
  - Compute constrained key $k_{\mathcal{S}} \leftarrow \mathsf{CPRF.Constrain}(k, \mathcal{S})$.
  - Build Merkle proof $\pi_{\mathsf{path}} \leftarrow \mathsf{Merkle.Prove}(\mathsf{leaf}, \mathbf{x})$, where $\mathbf{x}$ is the vector of leaves of tree.
  - Build SNARK proof $\pi \leftarrow \mathsf{SNARK.Prove}(\mathsf{pk}_\Pi, \phi, w)$, where the statement is $\phi \leftarrow \mathsf{leaf}$ and the witness is $w \leftarrow (\sigma, \mathsf{root}^*, \mathsf{com}, \mathsf{open}, \pi_{\mathsf{path}})$.
  - Add $(i, k_{\mathcal{S}}, \phi, \pi)$ to cs.
- Output $(\mathsf{wlt}, \mathsf{cs})$.

Protected Audience with private billing meets all of the stated goals: Correctness is straightforward and revenue preservation follows from unforgeability of the e-cash scheme. It achieves privacy against the exchange, since the exchange's interactions involve only issuing coins to advertisers and redeeming coins from publishers. It achieves privacy against advertisers and publishers due to the use of fenced frames, which unlinks client interactions with advertisers and publishers outside the attacker's coalition. A formal proof of security for the system is left as future work.

D.1 Protected Audience API with malicious client mitigation

In order to protect against malicious clients wasting advertisers' resources, the exchange must now be involved in ad auctions, but additional message roundtrips can be avoided. The extended Protected Audience API workflow with malicious client mitigations is as follows, with changes highlighted.

Step 0-A: Advertisers obtain coins from exchange. Advertisers (each indexed by $i \in [k]$) and the exchange begin by running the client and server setups, respectively, for the e-cash scheme. Advertisers trade in hard currency (USD_i) to the exchange for electronic coins (stored in updates to wallet wlt_i) to later pay publishers for

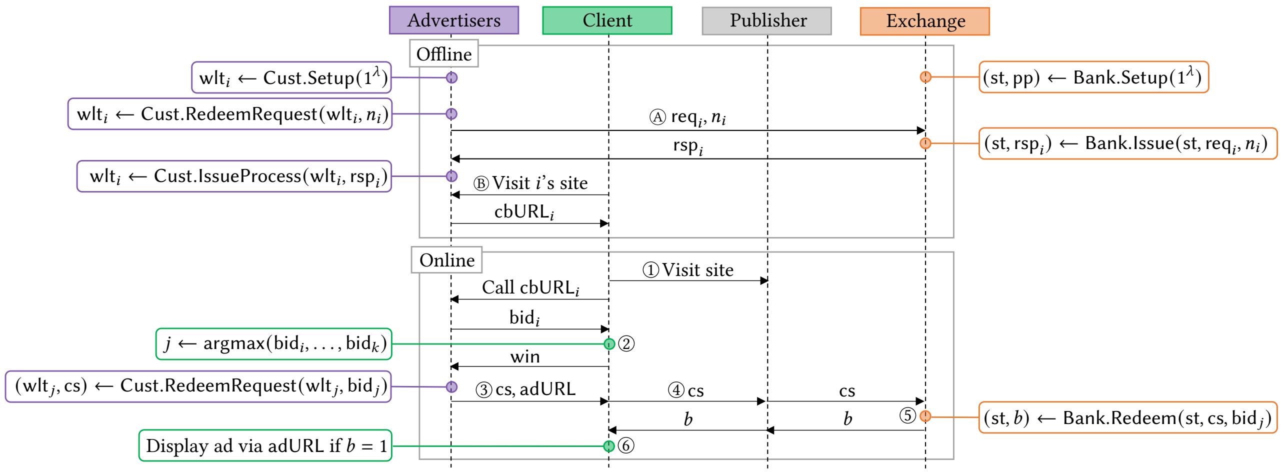

Advertisers
Client
Publisher
Exchange
Offline
$\mathsf{wlt}_i \leftarrow \mathsf{Cust.Setup}(1^\lambda)$
$(\mathsf{st}, \mathsf{pp}) \leftarrow \mathsf{Bank.Setup}(1^\lambda)$
$\mathsf{wlt}_i \leftarrow \mathsf{Cust.RedeemRequest}(\mathsf{wlt}_i, n_i)$
Ⓐ req_i, n_i
rsp_i
$(\mathsf{st}, \mathsf{rsp}_i) \leftarrow \mathsf{Bank.Issue}(\mathsf{st}, \mathsf{req}_i, n_i)$
$\mathsf{wlt}_i \leftarrow \mathsf{Cust.IssueProcess}(\mathsf{wlt}_i, \mathsf{rsp}_i)$
Ⓑ Visit i's site
cbURL_i
Online
① Visit site
Call cbURL_i
bid_i
$j \leftarrow \mathsf{argmax}(\mathsf{bid}_i, \ldots, \mathsf{bid}_k)$
②
win
$(\mathsf{wlt}_j, \mathsf{cs}) \leftarrow \mathsf{Cust.RedeemRequest}(\mathsf{wlt}_j, \mathsf{bid}_j)$
③ cs, adURL
④ cs
cs
b
b
⑤
$(\mathsf{st}, b) \leftarrow \mathsf{Bank.Redeem}(\mathsf{st}, \mathsf{cs}, \mathsf{bid}_j)$
Display ad via adURL if $b = 1$
⑥

Figure 5: Workflow of Protected Audience with e-cash-based private billing.

ad impressions. (Alternatively, advertisers can pay hard currency to the exchange at the end of a billing period.)

For the divisible e-cash scheme, advertisers will also precompute zk-SNARK proofs offline. If an advertiser knows the distribution of values it will spend, it can precompute proofs accordingly to save on precomputation costs (instead of naïvely precomputing a proof for every leaf of every tree in its wallet).

Step 0-B: Client visits advertiser websites. When a client visits an advertiser's website and performs certain actions, such as placing an item in a shopping cart, the advertiser invokes a Protected Audience JavaScript API to store a callback URL (cbURL_i) on the client's browser. The client's browser uses this callback to later run ad auctions.

Step 0-C: Exchange preallocates auction IDs to publishers offline. The exchange preallocates a set of unique auction IDs to publishers, which they later use to run valid and identifiable auctions.

Step 1: Client visits publisher website. When the client visits a publisher's website, the publisher invokes a Protected Audience JavaScript API to indicate its desire to serve a retargeted ad on its page and includes an auction ID (aucID) in the HTTP response.

Step 2: Client runs ad auction. The client's browser initiates an ad auction (e.g., a first-price, open-bid auction) by calling the stored callback URLs for the advertisers it has visited. Each callback returns the corresponding advertiser's bid (bid_i) for the impression. The client's browser runs the auction (argmax for a first-price auction) and notifies the winning advertiser (win to advertiser j) of the clearing price for the impression (implicit in a first-price auction).

Step 3: Winning advertiser sends ad and coins to client. The winning advertiser sends the URL of its ad creative (adURL), which is used to render the ad, to the client's browser, along with a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk_x}, \mathsf{cs}\|\mathsf{aucID}\|\mathsf{payID})$, where $\mathsf{pk_x}$ is the exchange's public key, the coins cs amount to the clearing price, and payID is an opaque payment ID.

Step 4: Client and publisher forward ciphertext to exchange. The client's browser forwards the ciphertext to the publisher, who forwards it to the exchange.

Step 5: Exchange decrypts and checks validity of coins. The exchange decrypts the ciphertext to obtain $(\mathsf{cs}, \mathsf{aucID}, \mathsf{payID}) \leftarrow \mathsf{Dec}(\mathsf{sk_x}, \mathsf{ct})$. If the coins are valid and have not been double-spent, then the exchange accepts ($b = 1$), sends the publisher hard currency (USD) for the value of the redeemed coins, and notifies the client of acceptance through the publisher. (Alternatively, the exchange can pay hard currency to the publisher at the end of a billing period.) Otherwise, the exchange sends rejects ($b = 0$) and the client's browser retries with the next winner of the auction.

Step 6: Client displays ad. If the exchange accepts, then the client's browser displays the ad on the publisher's website in an isolated "fenced frame" [59], which prevents the publisher from learning who the winning advertiser is.

Step 7: Exchange publishes auction-payment ID pairs. The exchange publishes the auction-payment ID pair (aucID, payID), which allows the advertiser to check that their payment went through.