Ibex: Privacy-preserving ad conversion tracking and bidding (full version)

Ke Zhong
University of Pennsylvania

Yiping Ma
University of Pennsylvania

Sebastian Angel
UPenn and MSR

ABSTRACT

This paper introduces *Ibex*, an advertising system that reduces the amount of data that is collected on users while still allowing advertisers to bid on real-time ad auctions and measure the effectiveness of their ad campaigns. Specifically, *Ibex* addresses an issue in recent proposals such as Google's Privacy Sandbox Topics API in which browsers send information about topics that are of interest to a user to advertisers and demand-side platforms (DSPs). DSPs use this information to (1) determine how much to bid on the auction for a user who is interested in particular topics, and (2) measure how well their ad campaign does for a given audience (i.e., measure *conversions*). While Topics and related proposals reduce the amount of user information that is exposed, they still reveal user preferences. In *Ibex*, browsers send user information in an encrypted form that still allows DSPs and advertisers to measure conversions, compute aggregate statistics such as histograms about users and their interests, and obliviously bid on auctions without learning for whom they are bidding. Our implementation of *Ibex* shows that creating histograms is 1.7–2.5× more expensive for browsers than disclosing user information, and *Ibex*'s oblivious bidding protocol can finish auctions within 550 ms. We think this makes *Ibex* capable of preserving a good experience while improving user privacy.

1 INTRODUCTION

Online advertising serves as the financial backbone of the free Web. Two key components of this ecosystem are the ability of ad platforms to *select* relevant ads for users and *measure* an ad's effectiveness. To do so, ad platforms track users' browsing habits to understand their behavior and demographics, which helps advertisers predict the value of showing an ad to the user and determine whether ads lead to the user performing some action such as buying a product (known as a *conversion*). A key issue with the current state of affairs is that users' browsing information is collected and shared by a multitude of providers, from publishers to ad platforms to advertisers, often without users' consent. Our animating goal in this paper is to propose an alternative: we describe *Ibex*, a system that allows advertisers to determine the value of a user so that they can bid in ad auctions, enables the selection of relevant ads for users, and supports the measurement of conversions—*all without collecting information about individual users*.

Figure 1 depicts the end-to-end process that results in an ad being shown to a user. It begins with a *user profile generation* phase in which advertisers (typically with the help of ad platforms) identify users, track their activities across different sites over time and aggregate this data, and ultimately generate a profile for each user. Then during a phase of *estimation of user value*, when a user visits a publisher's site, this visit triggers an auction. The product being

This is the full version of [96]. This version includes four appendices with proofs and additional materials.

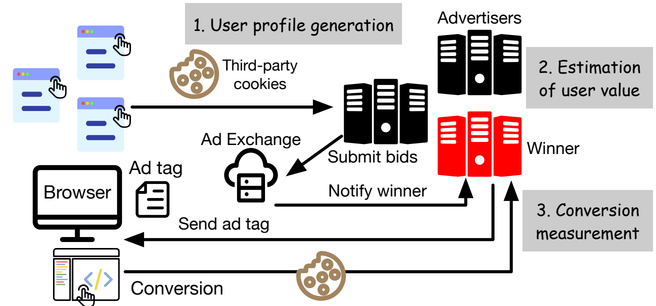

Figure 1: Lifecycle of a displayed ad today. Advertisers use third-party cookies to generate the profile of a user. Based on this profile, advertisers can determine how much to bid for the user's attention when the user visits a publisher that displays ads. After the user has seen the ad, advertisers keep track of whether the user acted on the ad (a conversion).

auctioned is real estate on the user's browser as they navigate a particular site. To determine the value of such real estate, advertisers rely on the profiles they have generated in the past for users and use that information to decide on how much to bid. Entities called *ad exchanges* run real-time auctions where they process the bids of different advertisers to determine whose ad to show to the user. After a winner is chosen and its ad is displayed in the user's browser, a phase of *conversion measurement* is used to measure the users' response to the ad: does the user interact with the ad or purchase some product or subscription as a result?

Protecting user privacy is challenging because in the current ecosystem, the above phases rely on two operations that must act on sensitive user information: *aggregation* and *bidding*. First, in the *user profile generation* and *conversion measurement* phases, *trackers* (e.g., advertisers, ad platforms, demand-side platforms) aggregate the activities of users across different sites over time using third-party cookies to get a better idea of who the users are and how they are affected by ads. Second, in the phase of *estimation of user value*, the user's profile is critical to determine how to bid for the user during an auction.

Ibex deals with the challenge of data aggregation with a new protocol that collects the information needed for the *user profile generation* and *conversion measurement* phases but does so at the granularity of groups of users rather than individuals. In particular, *Ibex* adopts the setting of recent industry proposals such as Google's Privacy Sandbox FLoC [92] and Topics API [22] that abandon the use of third-party cookies altogether. Instead, these proposals capture user profiles in a coarse, but still useful, way with a *group id* or several *topic identifiers*. *Ibex* extends these proposals to allow trackers to construct a *histogram* that conveys how many times a

particular group of users visits certain sites or performs particular actions after seeing an ad but without ever learning to which group a particular user belongs.

Ibex also designs an *oblivious bidding* protocol in which an advertiser or demand-side platform runs an arbitrary computation on the collected histograms to determine the value of each particular group of users (i.e., how much to bid for each group). Then, when a user visits a publisher and a real-time auction is conducted for that user, the bidder obliviously submits their bid for the auction without learning to which group the user belongs or which bid they even submitted. *Ibex* then takes these oblivious bids and uses them as inputs to an existing two-party private auction protocol (e.g., Addax [97]). The result is that the auctioneers learn who the winner of the auction is and how much they must pay, but they learn no information about the losing bidders' bids. Finally, the auctioneers charge the winning bidder but do so after a batch of auctions and (optionally) after some small amount of noise is added. This ensures that the final bill itself does not reveal to the bidder the group of a particular user in a particular auction.

In more detail, *Ibex*'s technical contributions are:

- **Private histogram.** A new two-party *asymmetric* private aggregation protocol that combines secret sharing and homomorphic encryption. The protocol requires one of the parties to very cheaply validate some of the inputs provided by the browser while the other party performs more (but still lightweight) operations. Crucially the two parties never talk to each other directly, which avoids a privacy vulnerability of prior aggregation protocols where one large ad platform helps aggregate reports of many advertisers and can piece together users' interests (§4.1). Having one party partially validate the inputs of the browser means that *Ibex* avoids expensive proofs to ensure that a malicious browser is not supplying bogus inputs. This means that the computation costs for browsers are minimal: they only need to split their aggregation report into additive shares and communicate the share (and some other materials) to each aggregator.
- **Oblivious bidding.** A new protocol whereby a bidder can submit the appropriate bid without learning the user's group. To do so, the bidder first pre-generates bids for different groups of users, encrypts secret shares of the bids, and stores them in a public bidding database. The user's browser fetches the encrypted bid shares corresponding to the user's group privately using *private information retrieval* (PIR) and submits the encrypted bid shares to the two auction servers that compute the auction. In the process above, no party learns the profile of the user.

Our implementation of *Ibex* shows that with *Ibex*'s private histogram aggregation, the response time when a browser visits a site increases to 1.7–2.5× over the status quo with no privacy. In the most optimistic case (out of many that we evaluate), the oblivious bidding protocol is fast enough to complete an auction in 550 ms, which is about 1.8× slower than existing non-private auctions.

Limitations. Besides data aggregation and bidding, there are other important aspects such as ad delivery that indirectly leak to the advertiser something about the user's interests. We do not implement protections for those other aspects, but recent work [87] looks at these complementary problems. Another limitation is that our auction protocol reveals to the auctioneer which advertiser wins the auction in order to bill the advertiser for the impression. Unfortunately this also means that the auctioneer could over time determine which advertisers are winning auctions for a given user and infer some of its interests. *Ibex*'s oblivious bidding protocol takes into account the user's group and can also incorporate some contextual data (e.g., type of site showing the ad, location within the page, time of day). However, if there are too many features, *Ibex*'s approach becomes impractical and more work is needed to devise a better mechanism. Finally, *Ibex*'s threat model is not as strong as we would like: we assume that browsers are malicious, but that advertisers, ad platforms, and auction servers are semi-honest.

2 BACKGROUND

This section gives a brief overview of how tracking and real-time bidding work today; we later discuss how recent industry proposals plan to change the ad ecosystem to provide better privacy for users and how *Ibex* fits into that new ecosystem.

Ad platforms can show ads that are relevant to users by understanding users' prior activities on the Web. This is done in a *user profile generation* phase in which trackers use third-party cookies and cookie matching techniques [27, 29] to assign the user a unique identifier. Trackers use this identifier to build a holistic profile of the user that includes activities such as which pages the user visits and which items the user purchases.

When a browser visits a publisher's site, the publisher—or more commonly a *supply side platform* (SSP) which is a company that represents the publisher—auctions the user on an *ad exchange*. In this auction, the exchange requests bids from interested bidders which are typically *demand side platforms* (DSPs). DSPs are companies that represent advertisers and run ad servers that have the resources required to participate in real-time auctions. One of the key steps that takes place during the auction is for bidders to *estimate the value of the user* (i.e., how much they are willing to bid). To make this decision, bidders are provided with the user's profile, demographic information, and relevant details about the publisher and the ad slot, such as size, type, and location within the page [75].

Based on this information, bidders return a bid to the ad exchange. The ad exchange runs an auction to select the winning bidder, charges the winner, and sends the ad tag of the winner to the browser; this ad tag contains information that the browser needs to retrieve and display the ad.

In the *conversion measurement* phase, trackers use third-party cookies to follow a user's actions after it clicks on the ad. Conversions refer to events such as a purchase or a signup and are used to measure the effectiveness of ads. The trackers first record the click by a browser on the publisher's site, and then record the activities of the user on the advertiser's site. Trackers match these recorded clicks and activities and attribute a conversion to a click. There are also more advanced mechanisms known as *view-through* conversions that can attribute a conversion to the *display* of an ad rather than a click (e.g., if the user sees an ad and then goes to the advertiser's site on a new tab without clicking on the ad [24]).

2.1 Where is the industry headed?

The ad industry is currently in the design phase of a more privacy-friendly ecosystem that deprecates the use of third-party cookies.

There have been several proposals put forth. Before describing them, we want to emphasize that these proposals are speculative and most have not been implemented or deployed. Our here goal is to just give a flavor of the ideas that have piqued the community's interest, and how *Ibex*'s mechanisms fit within the larger ecosystem.

Finding a proxy for user profile. Several works such as Google's Topics API [22] (a refinement of a prior proposal called FLoC [92]) and Microsoft's PARAKEET [17] introduce ideas to replace a user's profile and browsing history with something good enough to describe the interests of the user, but coarse enough to not uniquely identify the user. In these proposals, the browser is extended to include code that maps a user to some group of like-minded users or to a set of topics of interest, based on the websites that the user visit. We do not discuss how this mapping is done, as things are constantly in flux, but in *Ibex* we assume that some such mapping exists. We will use the term *group id* to describe the coarse identifier assigned to the user (*Ibex* generalizes to a set of topics).

Measuring conversion. Once users are assigned to groups, it becomes important for advertisers, DSPs, and ad platforms at large to have a mechanism to measure how users in each group interact with sites (how often do they buy a product, share an article, etc.). To do so, one proposal from Google is for browsers to send these reports to trusted hardware enclaves [2] that collect and aggregate the data, and produce noisy aggregates for ad platforms to use. Other proposals [12, 39] discuss the possibility of having multiple non-colluding servers compute this aggregate with secure multiparty computation. The way *Ibex* approaches this problem is also based on multiparty computation, but there is a key distinction in which servers *Ibex* uses, and what kind of computation they each do. As we will discuss in Section 4.1, in existing proposals an ad platform serves as one of the servers and is involved in processing *all* aggregation reports. This gives the ad platform a powerful vantage point that allows it to determine which sites a user is visiting; over time, the platform can piece together the user's interests, which is precisely what we wish to avoid.

Bidding. When a user visits the publisher and an auction is triggered, there is a question of how can the bidders determine which user is for auction and how valuable this user is to them? In Chrome's FLEDGE proposal [35], the idea is to have the auctions happen directly in the user's browser. In this way, the user's group id (or topics) stay within the browser, safeguarding the user's privacy. To enable this, each bidder must submit their *bidding function* (e.g., a machine learning model) to the browser, which then runs this function on the appropriate data to compute bids and then pick the winner. This approach is actually really appealing, but browsers can easily be modified to run code that extracts the bidding function of each DSP. In many cases, these functions represent the intellectual property of the company, and competitors could take advantage of this. There are also questions about the integrity of the auctions.

Another proposal called MaCAW [13] secret shares the user's group id (or topics) among two or more servers that run a simple bidding model (e.g., linear regression) to determine the bid for the user. This bid can then be used in an auction outside of the browser. *Ibex* also adopts this model of running the auction outside of the browser. However, *Ibex* proposes a very different mechanism than running MPC to compute a bid. Instead, *Ibex* introduces an *oblivious bidding* protocol whereby the bidder submits its bid without learning what bid it submitted. This is done with the help of the browser, as we detail in Section 5.

3 OVERVIEW OF *IBEX*

Ibex is a new ad platform that replaces a user's profile and browsing history with a *group id*. This group id is derived as per recent proposals such as Google's FLoC [92] and its successor [22]. *Ibex* then introduces various technical mechanisms to ensure that real-time bidding auctions and conversion measurement mechanisms continue to work, even when the group id is not shared (at least not in the clear) with other parties.

We begin by answering some basic questions about *Ibex*'s setting, and then discuss the threat model and goals in more detail.

Is *Ibex* necessary? One may wonder whether revealing a coarse group id is problematic. It is. Prior proposals that have done this [22, 92] have been criticized precisely because this is enough to leak sensitive user information [6, 7, 18, 23, 86]. Briefly, the criticisms state that while a single group id (or set of topics) alone might not reveal too much about a user, as time passes and the user browses the web, the group id inevitably changes—revealing new facets of the user's interests. Exposing such information allows trackers to study and understand users over time. With enough of these observations, trackers can put together a detailed profile of the user, allowing the inference of the user's browsing history.

Does *Ibex* prevent tracking? Even if one hides the group id from trackers, could they not continue to track users anyway? Indeed, *Ibex* does not prevent cross-site tracking explicitly. However, all major browsers are disabling third-party cookies which is the default way of tracking users across sites. There are other ways to track users across sites (e.g., browser fingerprinting [25]), but these methods are noisy, involved, and require the coordination of multiple sites. We therefore expect that sites will respect users' privacy if the existing functionality can be provided through other means.

How does *Ibex* work? In *Ibex*, browsers locally record all user activities and identify reports that should be sent to *aggregators* (e.g., an advertiser who tracks conversion reports). The aggregators combine the reports provided by many users and generate a profile for each group of users (rather than individuals). Each user's group id then serves as a proxy for the user's profile.

As we show in Figure 2, *Ibex* introduces algorithms that allow aggregators to function without seeing individual reports, and bidders in auctions to determine how much to bid for a user without learning who the user is. We discuss the details of aggregation in Section 4 and of bidding in Section 5. One key point worth mentioning is that *Ibex* requires multiple non-colluding servers to achieve these goals. We next discuss what we assume of the different parties in the ad ecosystem, and how we could instantiate non-colluding servers (which is a very strong assumption) in practice.

3.1 Threat model and assumptions

In *Ibex*, there are several principals: browsers, publishers (or SSPs that represent publishers), advertisers (or DSPs that represent advertisers), and ad platforms. There are also two additional roles that

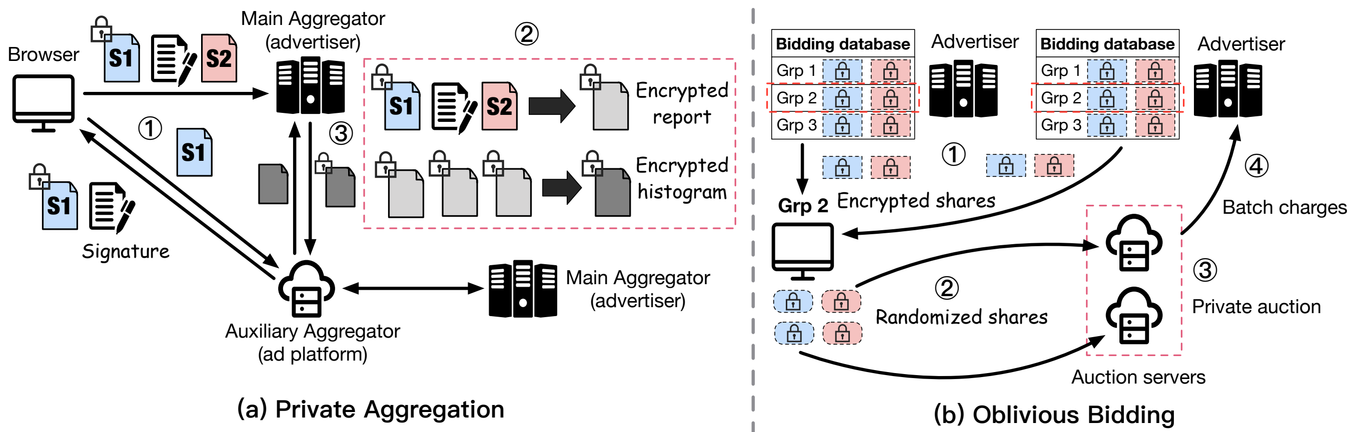

Figure 2: Overview of *Ibex*'s architecture. It consists of private aggregation (§4) and oblivious bidding (§5). Aggregation helps advertisers understand the value of different kinds of users, and they use this information to bid in auctions.

could be taken up by different combinations of these principals: auction servers and aggregation servers. We suggest the following concrete arrangement (other arrangements might work too).

For aggregation, each advertiser (or DSP) serves as a "main" aggregator, and its job is to aggregate the visits and conversion reports of the users that interact with its own site. Since *Ibex* requires *two* non-colluding aggregators, we also have one out of the many existing ad platforms serve as an "auxiliary" aggregator. In this role, the ad platform merely helps the main aggregator complete its task—it does not learn the final aggregation results (histograms).

For auctions, each publisher (or SSP) serves as one of the auction servers, and an ad platform serves as the other auction server. If having each SSP serve as an auction server is too onerous or creates too much market fragmentation (i.e., too many concurrent auction platforms), then one can use a service like *Divvi Up* [4] to take up the role of the auctioneer in place of each publisher. Divvi Up is a service provided by the non-profit ISRG [10] (the same organization in charge of Let's Encrypt [11]) that helps prop up a second server for applications that require non-collusion assumptions.

Based on the above principals and roles, we have the following threat model.

Browsers. Since browsers are under the control of users and can be easily modified, we model them as *malicious* adversaries that can deviate from any prescribed protocol.

Advertisers, publishers, and ad platforms. We model these parties as *honest-but-curious* adversaries: they will follow the prescribed protocol but will try to infer users' private information. We assume that these parties will not collude with each other. If one uses a service like Divvi Up, this model also applies to them.

Non-goals. *Ibex* does not prevent users from claiming to be interested in categories of sites or products that they are actually not. For example, claiming that they are interested in sports when in reality the user does not like sports. We are not aware of any mechanism that could enforce this against malicious users that can modify their browsers to submit false information.

4 PRIVATE HISTOGRAM AGGREGATION

In our architecture, and consistent with industry proposals (§2.1), when a user clicks on an ad or visits a site after seeing an ad (see-through conversion), the user's browser generates a report (e.g., a conversion). The goal of the aggregation protocol described in this section is for an advertiser or DSP to be able to tell which groups of users are most valuable to it (more likely to interact with its content, purchase products, etc.). In *Ibex* we wish to do this without any party learning to which group any individual user's report belongs. That is, the reports are aggregated in such a way that the advertiser learns the final aggregate value and the total number of reports, but does not learn which report belongs to which user.

Abstractly, we can model this task as follows. There are N users and d different groups. Each user holds a report belonging to one group. Among these N users, there are N_0 users with reports of group 0, N_1 users with reports of group 1, . . ., and N_{d-1} users with reports of group $d-1$. Each report is a number indicating the group. At the end of the protocol, the aggregator obtains a vector $[N_0, N_1, \ldots, N_{d-1}]$. The vector can be interpreted as a histogram with the x-axis corresponding to the different groups and the y-axis representing the number of reports belonging to each group.

4.1 Issues of existing aggregation protocols

Asking users to submit the reports in plaintext violates users' privacy, as they would be revealing their group. A promising alternative is works [34, 39, 47, 55, 65] that show how to privately aggregate data while ensuring that the inputs are well-formed (to avoid corrupting the final result). They fall into two categories:

- *Homomorphic encryption (HE)*: each user encrypts its report using a homomorphic cryptosystem that supports batching (so it can represent the value of 0 or 1 for each of the d groups within a single ciphertext) and sends the resulting ciphertext and a proof that the corresponding plaintext is well-formed to the aggregator. The aggregator sums up reports across many users and forwards the result to a third party with the decryption key who can recover the final histogram.

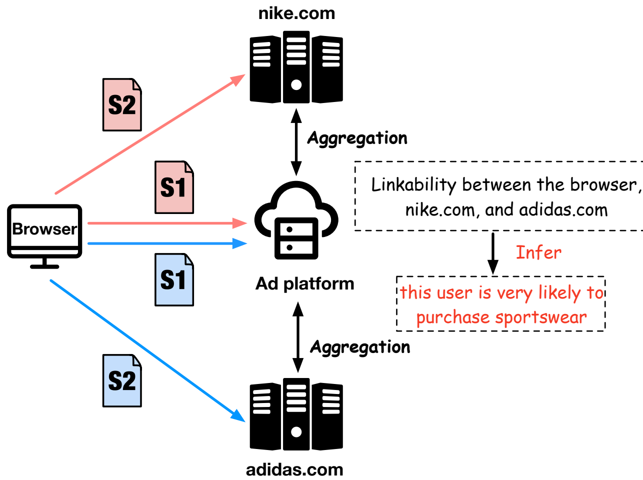

Figure 3: Linkability issue of a centralized ad platform in current advertising architecture.

- *Aggregate shares*: the user splits its report (represented as a d-bit vector) into two or more additive shares and sends them to several non-colluding aggregators. The aggregators then compute a multi-party protocol on the shares to obtain the final aggregate histogram across many users.

Neither of these types of protocols is a great fit in our context. For HE aggregation, it is costly for the browser to generate zero-knowledge proofs (ZKP) that show that the ciphertext corresponds to a well-formed plaintext. In Appendix C.2 we show that the cost of generating the required ZKP for state-of-the-art HE schemes is expensive. For multi-server aggregation protocols such as Prio [55] and Poplar [47] their symmetric nature poses a problem if a large ad platform helps aggregate reports for many advertisers. Consider the example of Figure 3 where an advertiser (e.g., nike.com or adidas.com) employs the help of a (non-colluding) ad platform to perform the aggregation. If the ad platform helps more than one advertiser to aggregate reports, then the reports must specify for which advertiser they are meant. For example, browsers will need to tell the ad platform: "this is a share of a report for nike.com". While the ad platform does not learn the user's group from the share, it still learns that the user visited nike.com. In the extreme case where a single ad platform helps all advertisers aggregate reports, it is akin to a situation where it gets to see users' entire history. We call this *advertiser linkability*.

One way to eliminate this issue is to designate two aggregators to handle the reports from *all* users and *all* sites, and ensure that the id of the advertiser is encoded in the report share and is only visible once the shares are aggregated. In this way, neither aggregator can distinguish the target advertiser of a user's report. However, this design has two shortcomings: (1) it can be abused; and (2) it increases the size of a report.

First, as advertisers are not involved in the aggregation process, a malicious user can send many random reports to the two aggregators; the advertiser cannot verify whether activities of these reports are real at all, since it only sees the final histogram. For example, an adversary can send many reports indicating many users from group 2 purchased shoes on an advertiser's site, which tricks the advertiser into thinking that users from group 2 are the most desirable. In contrast, in *Ibex*, the user sends its report as it is visiting the advertiser and making the purchase, so the advertiser knows that this is a valid report.

Second, since reports of all sites are sent to the same two aggregators, each report needs to include an additional feature, the target advertiser's id. Based on prior documentation [14], the total number of advertisers can be over 10 million. Increasing the size of a report impacts performance since many of these protocols [47, 55] have computational complexity linear in the size of the report. One exception is the work of Anderson et al. [39] which scales well. Nevertheless, it is still not a great fit in our setting since advertisers are out of the loop and cannot verify the authenticity of reports. Appendix C discusses these schemes in more detail.

4.2 Asymmetric aggregation

To solve the advertiser linkability problem, *Ibex*'s key idea is to enable an "auxiliary" aggregator (such as an ad platform) to help a "main" aggregator (an advertiser or DSP) to aggregate reports without actually learning the identity of the main aggregator that it is helping. This seemingly paradoxical property boils down to enforcing a notion of information asymmetry: we allow the main aggregator to know the identity of the auxiliary aggregator and its public key, but not the other way around. To leverage this asymmetry, all messages from the auxiliary aggregator to the main aggregator must be proxied via the browser. But as browsers can be malicious, the proxied messages must be signed by the sender. This means that the main aggregator cannot send any messages to the auxiliary aggregator, as otherwise the auxiliary aggregator would be required to know the identity of the main aggregator to validate its signatures, violating information asymmetry.

In *Ibex*, the auxiliary aggregator generates the proof required to validate the report, encrypts the report share, signs the proof and encrypted share, and gives these messages to the browser. The browser then forwards these messages and the other report share to the main aggregator. The main aggregator receives its share, the auxiliary's aggregator signed encrypted share and proof, and uses these materials to locally compute the histogram.

Challenge of adapting existing protocols. Given the above, a natural question is whether existing aggregation protocols such as Prio [55] and Poplar [47] can be easily adapted to be asymmetric. This is not the case. The reason is that in order for these protocols to validate the inputs provided by users, the aggregators need to interact with each other for two or more rounds (e.g., in Prio, aggregators use Beaver triples [44] and are required to exchange their local shares of a value). Thus, the proof cannot be generated locally by the auxiliary aggregator without receiving a message from the main aggregator (and therefore learning its identity and violating asymmetry). Ideas from group signatures [53, 76] could hide the identity of the sender, but the number of main aggregators could be in the thousands, which would lead to high costs. Instead, *Ibex* designs a protocol where validating inputs from browsers requires no communication between the two aggregators.

4.3 Histogram aggregation overview

To enable histogram aggregation, there are a few things that need to be addressed. First, we need a way to encode the user's group g into a representation that indicates the g-th bin of the histogram, and such representation should be aggregatable. Second, we need an encryption scheme that naturally represents all the bins and the encryption scheme should be aggregatable over ciphertexts. Third, as browsers are malicious, we should avoid asking the browser to do the encryption; otherwise, the browser needs to generate an expensive ZKP to prove that the ciphertext encrypts a well-formed encoding of the user's group. We discuss the high-level ideas of *Ibex*'s histogram aggregation design below and discuss the details in the sections that follow.

Histogram encoding. We observe that a *one-hot* vector, which is a vector where one entry is a 1 and all others are zeros, is a good fit to represent the aggregatable bins. The browser encodes its report, group g, into the one-hot vector $[0, \dots, 1, \dots, 0]$. Element-wise additions over the vectors give us the "bin" aggregation. For example, given three vectors, $[0, 1, 0, 0]$, $[0, 1, 0, 0]$, and $[0, 0, 1, 0]$, corresponding to two reports for group 2 and one report for group 3, their sum yields $[0, 2, 1, 0]$; the i-th element of the vector gives the total number of reports belonging to group i.

Encryption and aggregation. Given the above representation, a strawman solution is to encrypt each value in the vector individually with an additively homomorphic cryptosystem like Paillier [84]. However, this is far too expensive. Instead, cryptosystems based on the Ring-LWE assumption [78] have the advantage that each bin can be represented as a different coefficient in a polynomial. Specifically, in the BFV cryptosystem [50, 60], plaintexts are polynomials of degree at most N with integers coefficients modulo t. Formally, they are polynomials from the quotient ring $R_t = \mathbb{Z}_t[x]/(x^N + 1)$, where N is a power of 2 and t is the *plaintext modulus*. One can therefore represent the one-hot vectors $[0, 1, 0, 0]$, $[0, 1, 0, 0]$, and $[0, 0, 1, 0]$ with the monomials x, x, and x^2 in R_t, respectively. Their sum is $2x + x^2$, which is equivalent to the histogram $[0, 2, 1, 0]$.

Correct encoding and secret sharing. One way to avoid the browser generating the encryption of the one-hot vector for group g and proving that it is well-formed (i.e., that a single entry is a 1 and the rest are 0) is to have the servers generate the one-hot vector themselves. Of course, the client cannot simply give g to the servers, since this would leak g. Instead, the browser can view g as an element in a group $\mathbb{Z}_d$ (d is the total number of groups) and can randomly sample two additive shares s_1 and s_2 from $\mathbb{Z}_d$ such that $s_1 + s_2 = g \pmod d$. The browser can then send s_1 to the auxiliary aggregator and s_2 to the main aggregator. The auxiliary aggregator represents s_1 as the monomial x^{s_1} and encrypts it with its public key pk_{aux} to obtain: $c = \text{Enc}(pk_{aux}, x^{s_1})$. The auxiliary aggregator then sends (via the browser) c to the main aggregator.

Once the main aggregator receives c, it can represent its share s_2 as the monomial x^{s_2} and then perform a plaintext-ciphertext multiplication: $x^{s_2} \cdot c = \text{Enc}(pk_{aux}, x^{s_1} \cdot x^{s_2}) = \text{Enc}(pk_{aux}, x^{s_1+s_2})$. For example, suppose a user's group id is $2 \in \mathbb{Z}_4$, and the browser generates two additive shares, 1 and 1. The auxiliary aggregator takes its share (1) and generates the ciphertext $c = \text{Enc}(pk_{aux}, x^1)$. The main aggregator receives its share (also 1) and the ciphertext c. It then expresses its share as the monomial x^1, and performs a plaintext-ciphertext multiplication with c to obtain $\text{Enc}(pk_{aux}, x^2)$, which is the encoded one-hot vector representing a report for group 2. This plaintext-ciphertext multiplication essentially "shifts" the original plaintext by s_2 positions to the right.

Dealing with shift overflows. One issue with the above is that when the shift operation overflows it results in a negative coefficient. For example, if $g = 1$, its two shares can be 2 and 3 ($2 + 3 \equiv 1 \in \mathbb{Z}_4$). So the auxiliary aggregator generates a ciphertext for the monomial x^2, and the main aggregator further increases its degree by 3 (equivalent to shifting the entries in a one-hot vector to the right 3 positions). The result is: $\text{Enc}(pk_{aux}, x^{2+3}) = \text{Enc}(pk_{aux}, -x)$. This negative coefficient occurs because plaintexts are defined over $R_t = \mathbb{Z}_t[x]/(x^N + 1)$, so $x^N + 1 \equiv 0 \pmod{x^N + 1}$ and $x^{N+1} \equiv -x \pmod{x^N + 1}$. We address this by increasing the polynomial degree to $2d$; since the multiplied monomials both have degrees lower than d, multiplying them will not overflow a polynomials with degree of $2d$. This approach is very simple and more efficient than alternatives (e.g., performing rotations [61]).

End-to-end flow. We consider a single auxiliary aggregator that serves all main aggregators though *Ibex* naturally extends to support multiple auxiliary aggregators. The auxiliary aggregator generates a pair of keys for the homomorphic cryptosystem and a pair of signing and verification keys. The public key and verification key are distributed to all main aggregators.

A browser privately splits its report into two shares, s_1 and s_2, and sends s_1 to the auxiliary aggregator. The auxiliary aggregator responds to the browser with an encrypted report share and a signature generated with its signing key. The browser then sends s_2, the encrypted report share, and the signature to the main aggregator. The main aggregator first validates these materials, and shifts the encrypted report share using s_2 to obtain the appropriate encryption of the report. The main aggregator then adds up many encrypted reports over a window of time to obtain an encrypted aggregate result. Finally, the main aggregator adds some randomness to this encrypted aggregate result, submits it to the auxiliary aggregator for decryption, and removes the randomness from the decrypted result to recover the histogram.

4.4 Construction

Ibex's private histogram protocol provides the following properties.

- *Correctness.* If all parties follow the protocol, the final output of the histogram aggregation protocol is the correct distribution of the number of reports from different groups.
- *Robustness.* In *Ibex*, malformed inputs from malicious browsers can be detected and discarded by both aggregators.
- *Privacy. Ibex*'s histogram aggregation protocol hides all raw inputs of users from both aggregators, except what is implied by the output histogram to the main aggregator. Moreover, the auxiliary aggregator does not learn which user sends a report to which main aggregator.

Notation.

- All polynomials defined below have degree $2d$.

- The polynomials below are defined over the ring $\mathbb{Z}_t[x]/(x^{2d}+1)$, with plaintext modulus being t. Since each coefficient will act as a "bin" in our histogram, and the maximum value of a coefficient is t, then t is also the maximum number of reports that can be summed together before overflow happens (e.g., $t + 2 \equiv 2 \in \mathbb{Z}_t$).
- The auxiliary aggregator generates a pair of public and secret keys, *pk* and *sk*, from an additively homomorphic cryptosystem by calling HE-KeyGen. *pk* is public to all main aggregators.
- The auxiliary aggregator generates a pair of signing and verification keys, *sigkey* and *vrfkey* by calling Sig-KeyGen; *vrfkey* is public to all main aggregators.

Subroutines. Now we define subroutines that will be used later in the construction of the private histogram aggregation.

- Additive-Shares$(g) \rightarrow (s_1, s_2)$. Takes an element $g \in Z_d$, and generates two uniformly random shares $s_1, s_2 \in \mathbb{Z}_d$ such that $s_1 + s_2 \equiv g \in Z_d$. Neither share leaks any information about g.
- HE-KeyGen$(1^\lambda) \rightarrow (pk, sk)$. Takes in a security parameter λ and outputs a public key *pk* and a secret key *sk*.
- HE-Enc$(pk, pt) \rightarrow c$. Takes the public key *pk* and a plaintext polynomial *pt* and outputs a ciphertext c which encrypts *pt*.
- HE-Dec$(sk, c) \rightarrow pt$. Takes the secret key *sk* and a ciphertext c and outputs the decrypted polynomial *pt*.
- HE-Add$(c_1, c_2) \rightarrow c_{add}$. Takes two ciphertexts, c_1 and c_2 which encrypt two polynomials, and outputs a ciphertext c_{add} which encrypts the sum of the two polynomials.
- HE-Add-Plain$(c, p_2) \rightarrow c_{add}$. Takes a ciphertext c, which encrypts a polynomial p_1, and a plaintext polynomial p_2, and outputs a ciphertext c_{add} which encrypts $p_1 + p_2$.
- HE-Mul-Plain$(c, p_2) \rightarrow c_{mul}$. Takes a ciphertext c, which encrypts a polynomial p_1, and a plaintext polynomial p_2, and outputs a ciphertext c_{mul} which encrypts $p_1 \cdot p_2$.
- Sig-KeyGen$(1^\lambda) \rightarrow (sigkey, vrfkey)$. Takes security parameter λ and outputs a signing key *sigkey* and a verification key *vrfkey*.
- Sign$(sigkey, m) \rightarrow sig$. Generates a digital signature on message m using the signing key *sigkey*.
- Verify$(vrfkey, m, sig) \rightarrow valid$. Outputs whether *sig* is a valid signature for message m with the signer's verification key *vrfkey*. *valid* is set to true when *sig* is valid. Otherwise, it is set to false.

Figure 2(a) shows the architecture of *Ibex*'s private histogram. *Ibex*'s aggregation consists of three steps as follows.

Step 1: Browser generates shares and obtains materials from auxiliary aggregator. The browser runs the Additive-Shares function on its report group $g \in \mathbb{Z}_d$, and obtains two shares, s_1 and s_2, as the output. The browser sends s_1 to the auxiliary aggregator. The auxiliary aggregator first checks whether s_1 is valid, and discards the share otherwise. Then, it generates the encrypted share *cs* and signature *sig* as follows.

(1) If $s_1 \notin \mathbb{Z}_d$, discard the share and abort;
(2) $pt \leftarrow x^{s_1}$;
(3) $cs \leftarrow$ HE-Enc(pk, pt);
(4) $sig \leftarrow$ Sign$(sigkey, cs)$.

The auxiliary aggregator sends (cs, sig) back to the browser; the browser then forwards (cs, sig, s_2) to the main aggregator.

Step 2: Main aggregator validates, recovers, and aggregates reports. For each aggregation report, the main aggregator receives a tuple (cs, sig, s_2) of encrypted share, signature, and share from a browser. For each tuple, the main aggregator first validates the signature using the auxiliary aggregator's verification key *vrfkey*, then locally recovers the encrypted report *cr* if validation passes.

(1) If $s_2 \notin \mathbb{Z}_d$, discard the report and abort;
(2) If Verify$(vrfkey, cs, sig) \neq true$, discard report and abort;
(3) $pt \leftarrow x^{s_2}$;
(4) $cr \leftarrow$ HE-Mul-Plain(cs, pt).

For a number of recovered encrypted reports $(cr_1, \ldots, cr_N)$, the main aggregator combines them into the encrypted aggregation result (*cAgg*) as follows.

(5) $cAgg \leftarrow \sum_{i=1}^{N} cr_i$, this is ciphertext addition using HE-Add.

Step 3: Main aggregator obtains aggregation result. To hide the aggregation result from the auxiliary aggregator (who is the only one who can decrypt *cAgg*), the main aggregator first generates a *mask* polynomial that has uniformly random coefficients, and adds it to the local encrypted aggregation result as follows.

(1) $mask \leftarrow$ uniformly random element in $Z_t[x]/(x^{2d}+1)$.
(2) $mAgg \leftarrow$ HE-Add-Plain$(cAgg, mask)$.

The main aggregator submits *mAgg* to the auxiliary aggregator, who can decrypt *mAgg* using its secret key *sk*:

(3) $dAgg \leftarrow$ HE-Dec$(sk, mAgg)$.

The main aggregator receives the decrypted polynomial *dAgg* from the auxiliary aggregator. It removes the previously generated random mask to obtain the real aggregation result as follows.

(4) $agg \leftarrow dAgg - mask$.

Decode the histogram. Recall that in Step 1, the browser invokes Additive-Shares to generate two additive shares in $\mathbb{Z}_d$. These two shares add up to either g or $g+d$ in $\mathbb{Z}_{2d}$. Thus, the coefficients of the g-th and $(g+d)$-th term in the decrypted polynomial, *agg*, refer to the number of reports belonging to the same group g. For example, suppose d is 2 (only two groups of reports) and the decrypted polynomial that the main aggregator obtains is $1x^0 + 2x^1 + 3x^2 + 4x^3$. This means that main aggregator receives 4 (1+3) reports from group 1, and 6 (2+4) reports from group 2.

Theorem 4.1. *Ibex's private histogram aggregation protocol achieves the properties defined in Section 4.4.*

We give the full proof in Appendix B.1.

4.5 Multiple sets of HE parameters

Different aggregation tasks may require different number of groups. In *Ibex*, the auxiliary aggregator can generate different HE parameters including the secret key, public key, and the number of total groups (d). For each set of parameters, the auxiliary aggregator assigns it a unique *id*. Then, in Line 4 of Step 1, the auxiliary aggregator embeds the *id* into the message it signs $(id||cs)$.

5 OBLIVIOUS BIDDING

In today's ad auctions, each bidder is given the user's profile as well as contextual information; the bidder then generates a bid and submits the bid to the auctioneer. The auctioneer computes the

auction and selects the highest bidder as the winner, and outputs a sale price (in some auctions the sale price is the highest bid itself, in others it may be the second highest bid). In *Ibex*, each user is assigned to a group and uses its group id as a proxy for its profile. Our goal is to enable bidders to generate a bid for a given group id without actually learning the group id.

5.1 Private auctions

Bidding does not make sense without a corresponding auction protocol. So for completeness, we describe a semi-honest two-party private auction protocol where each party takes a set of bid shares as input and the parties jointly compute the winner and the sale price of the auction. Note that the auction protocol that we describe is not novel; it simply uses a generic MPC framework. That said, there are very efficient custom two-party auction protocols that *Ibex* could use instead [97]. Importantly, these works take shares of bids as inputs but they are agnostic to how the bids (and the corresponding shares) are generated. The process of bid generation is precisely what is novel in *Ibex*.

The auction protocol consists of two parts: how to encode bids into shares, and how to compute the auction using shares.

Bid encoding. Each bidder represents its bid (assumed to exist) as a binary string B and then generates two additive shares B^1 and B^2 such that $B^1 \oplus B^2 = B$. We use this binary encoding because in our context we find that it is cheaper to use an MPC framework that operates over boolean rather than arithmetic circuits to compute the auction (see Section 7.1 for details).

Private auction. Each bidder submits its first share B^1 to the first auction server and B^2 to the second auction server. The two servers collect shares from many bidders and then run a secure two-party computation that outputs the winning bidder and the auction's sale price (they basically reconstruct the bids inside the MPC and then find the highest bid). Appendix A.1 provides the pseudocode.

5.2 High-level idea of oblivious bidding

In *Ibex*, the bidders use the aggregated information about each group of users to decide their bidding strategies. When the browser visits a publisher's site and needs an ad tag, it locally selects potential bidders from a list provided by the publisher (this is actually how header bidding [3] works today and how Google's FLEDGE [35] proposal is intended to work). If the list is too large, the browser can filter them based on which ones are the best match for the user by leveraging the local profile stored within the browser.

To hide the user profile but allow bidders to bid for a user, each bidder encodes its bids into two shares, encrypts each share with a different auction server's key, and stores them in the bidding database. Note that one auction server can only decrypt one share but not the other; without it, it cannot learn the bid. The browser then uses a single-server *private information retrieval* (PIR) protocol to privately read encrypted bid shares from the invited bidder's bidding database while hiding which element of the database is read. The browser then randomizes the encrypted bid shares before submitting each encrypted bid share to the corresponding auction server. Each auction server then decrypts the received randomized bid shares, runs a private two-party auction protocol, and tells the browser where to fetch the winner's ad.

Why and how to randomize encrypted bid shares. The bidding database is pre-generated and public; auction servers can access it too. This means that when an auction server receives an encrypted bid share from the browser, it could simply scan the bidding databases of all bidders and find the matching ciphertext, thereby learning the user's group id.

Ibex avoids this by randomizing the bid shares—not just the ciphertext themselves but also the underlying plaintext (the share itself). It is important to randomize the plaintext because the auction server has the corresponding secret key so it can decrypt the ciphertext. To do so, recall that the two shares B^1 and B^2 of the binary representation of a bid B are uniformly random, and $B = B^1 \oplus B^2$. A browser can generate a uniformly random mask *mask* of the same length as B and add *mask* to the shares. Since $(B^1 \oplus mask) \oplus (B^2 \oplus mask) = B$, the bid B is unchanged but the auction servers do not obtain the original B^1 and B^2. A *bit homomorphic* encryption scheme [63, 64] supports adding randomness (XOR over bits) to the encrypted shares.

5.3 Properties

Ibex's oblivious bidding protocol provides the following properties.

- *Correctness.* If the browser and all bidders follow the protocol, the two auction servers each receive a valid bid share for the user's group from each bidder. If the auctioneers are also honest, then the output of the auction is correct (e.g., the winner is the highest bidder and the sale price is the winner's bid).
- *Robustness.* Misbehavior of malicious browsers (§5.5) can be detected in *Ibex*'s oblivious bidding protocol.
- *Privacy. Ibex*'s oblivious bidding protocol hides an honest user's group id from the auction servers and bidders.

5.4 Construction

Below is the notation and subroutines that we will use.

- Each of the two servers holds a set of public and secret keys of an additively homomorphic cryptosystem. We use the Goldwasser-Micali's cryptosystem [63, 64] since it encrypts each bit individually and supports homomorphic XOR over ciphertexts.[1] We denote the set of keys of the first auction server as (pk_1, sk_1) and the keys of the second auction server as (pk_2, sk_2). pk_1 and pk_2 are made available to all bidders and browsers.
- There are d total groups and k bidders are invited to an auction.
- The range of bids is from 0 to $2^\ell - 1$. All bit strings and lists of ciphertexts below have length ℓ.

Subroutines. Below are the subroutines that we use. Appendix A provides details on their constructions.

- Bid-Encode(b) $\to (B^1, B^2)$. Encodes bid $b \in [0, 2^\ell - 1]$ as a bit string B, and generates shares B^1 and B^2 such that $B^1 \oplus B^2 = B$.
- Priv-Auction($B^1_1, \ldots, B^1_k, B^2_1, \ldots, B^2_k$) $\to (bs, id)$. Runs a semi-honest two-party auction protocol between the auction servers. The i-th

[1] We make this choice since the two-party auction protocol that we use takes as input boolean shares. If we had instantiated *Ibex* with an auction protocol that uses integer shares, we would have used cryptosystems like Paillier [84] or ElGamal [59].

server inputs the bid shares of the k invited bidders, $B^i_1, \ldots, B^i_k$. This subroutine outputs the sale price bs and the winner's index id without leaking anything else.

- Share-Enc$(pk, B) \rightarrow (C)$. Encrypts each bit in the bit string B into a ciphertext using additive bit homomorphic encryption with public key pk, and outputs the list of bit encryption C.
- Share-Dec$(sk, C) \rightarrow (B)$. Decrypts each ciphertext in C using the secret key sk and outputs the bit string B of decrypted bits.
- Share-Add$(C_1, C_2) \rightarrow (C_{add})$. Takes two lists of ciphertexts C_1 and C_2, which encrypt the bit string B_1 and B_2 respectively, and outputs the list of ciphertexts C_{add}, which encrypts each bit in the bit string $(B_1 \oplus B_2)$.
- PIR-Read$(g) \rightarrow (entry)$. Protocol that fetches the g-th row of some database without revealing the index g.

5.4.1 Workflow of oblivious bidding. *Ibex*'s oblivious bidding is depicted in Figure 2(b). It consists of 5 steps. In the *initialization* step (not depicted), bidders use information they acquire about each group of users through *Ibex*'s aggregation (§4) and pre-generate bids for each group. A bidder generates its bid shares, encrypts them, and posts the ciphertexts to a bidding database. In Step (1), the browser identifies potentially interested bidders using its local history. In Step (2), the browser fetches the encrypted bid shares for its group from the bidding database, randomizes the encrypted shares, and submits the randomized encrypted shares to the auction servers. In Step (3), the two auction servers decrypt the received ciphertexts, compute the auction, and notify the browser of the winner's ad tag. Finally, the auction servers store the sale price and the winner of each auction. After a long enough time window, Step (4) occurs, in which the auction servers issue a bill to each bidder for all pending charges. The details are as follows.

Initialization: Bidders set up bidding database. Each bidder generates a bid for each group of users, encodes, splits, and encrypts the shares. For the group of users for whom the bidder is not interested, it simply chooses a bid of zero. It sets up its bidding database db as follows.

- Initialize an empty database db;
- For each group $g \in [1, d]$, the bidder generates its bid $b[g]$ using any algorithm of its choice and then does the following;
- $(B^1[g], B^2[g]) \leftarrow$ Bid-Encode$(b[g])$;
- $C^1[g] \leftarrow$ Share-Enc$(pk_1, B^1[g])$;
- $C^2[g] \leftarrow$ Share-Enc$(pk_2, B^2[g])$;
- Set the g-th row of db to $(C^1[g], C^2[g])$.

Step 1: Browser selects Bidders. The browser locally uses its browsing history to select k bidders for the auction using a local ads selection algorithms [66, 67, 87, 88] without leaking the user's browsing history and preferences. Google's FLEDGE proposal [35] also has a similar mechanism.

Step 2: Browser fetches and randomizes encrypted shares. The browser locally computes its group id g. For each invited bidder i, it reads the encrypted bid shares from each bidder's database and does the following to randomize the encrypted shares and generate the encrypted shares with randomness, CR^1_i and CR^2_i.

(1) $(C^1_i[g], C^2_i[g]) \leftarrow$ PIR-Read(g) to i's database;
(2) $mask_i \xleftarrow{R} \{0,1\}^\ell$
(3) $R^1_i \leftarrow$ Share-Enc$(pk_1, mask_i)$;
(4) $R^2_i \leftarrow$ Share-Enc$(pk_2, mask_i)$;
(5) $CR^1_i \leftarrow$ Share-Add$(C^1_i[g], R^1_i)$;
(6) $CR^2_i \leftarrow$ Share-Add$(C^2_i[g], R^2_i)$.

The browser submits CR^1_i and CR^2_i for each bidder i to the auction servers. Note that the browser does not disclose the identities of the bidders it selects to the auction servers; the servers only learn the number of bidders and the index of each encrypted share in the list. This is important, as otherwise the auction servers would learn all of the bidders that are invited to the auction and can use that information to better infer the user's interests over time. We limit the auction servers to only learning the identity of the winner.

Step 3: Auction servers compute auction. The servers decrypt all ciphertexts they receive, and run the auction protocol.

(1) S_1 computes $BR^1_i \leftarrow$ Share-Dec(sk_1, CR^1_i);
(2) S_2 computes $BR^2_i \leftarrow$ Share-Dec(sk_2, CR^2_i);
(3) $(bs, id) \leftarrow$ Priv-Auction$(BR^1_1, \ldots, BR^1_k, BR^2_1, \ldots, BR^2_k)$.

The auction servers respond to the browser with the index of the winner, id. The browser then sends to the auctioneers the winner's identity, which they use to notify the winner, request the ad tag, and forward it to the browser.

Step 4: Delayed and batch charges. The auction servers record the winner and the sale price of each auction. In *Ibex*, a bidder is not immediately charged for winning the auction as that would allow the bidder to link a recent click of a user with the auction's sale price, thereby leaking the user's group id. Instead, the auction servers charge each advertiser in a batch after the number of its winning auctions exceeds a certain threshold (e.g., every ten thousand auctions). Note that there might still be a small leakage from the aggregate value. For example, suppose that every bid from the bidder is even except for the bid for group 5 which is odd. Then, if the final aggregate is odd, the bidder can infer that at least one of the ten thousand auctions that it won was for a bidder of group 5. If even this type of leakage is unaccepted, one of the auctioneers could add a careful amount of noise to the final value before disclosing it to the bidder. Differential privacy [57, 58, 81] can be used to analyze how much noise must be added and the sensitivity of this aggregate value. Further, it can shed light on the tradeoff between privacy and utility (i.e., what premium must the bidder pay due to the added noise).

5.5 Malicious browsers

The protocol above ensures that the user's group id is kept confidential. But as a browser can be malicious, it may try to submit arbitrary encrypted shares to the auction servers. This can lead to a sale price that is higher or lower than what the winning bidder actually bid. To defend against malicious browsers, when each bidder sets up the database, it also provides all of its bids in a shuffled order to the auction servers. This means that the auction servers learn all of the bidder's bids, but not their mapping to specific group ids. Note that in the status quo, ad exchanges learn all of the bids as well, so this is not an additional source of leakage. When the private auction outputs the winner's bid as the sale price, the auction servers can

check if this sale price is on the list of bids that the bidder previously supplied. If not, the auction servers mark this auction as invalid and do not charge the winner. The auction servers can also mark the browser as potential fraud and deny services to these browsers.

One small caveat is that this does allow a malicious browser to create a fake bid that is one of the other bids submitted by a bidder. However, the browser could always do that because there is currently no mechanism to prevent a browser from claiming that it belongs to a different group than it actually does.

5.6 Update bidding database

The bidders may update their bids for different groups of users at any time. When it changes the bids, it generates a whole new bidding database by running the **initialization** step (§5.4.1) even though the bids for some groups of users will remain the same. It then sends the shuffled bid list to the auction servers as before (§5.5). This way, the auction servers only learn the advertiser changed bids for some groups of users, but cannot learn the group ids of the updated bids.

Browsers could cache the fetched encrypted bid shares of a bidder when it selects the bidder for an auction. If the next time the browser selects the same bidder for an auction, instead of issuing PIR queries again, it sends "if-modified-since" requests to the advertiser. If the bidder's bidding database has not been updated since the last query of the browser, the browser does not need to issue a PIR query again. This reduces the end-to-end latency of oblivious bidding (§7.4).

THEOREM 5.1. *Ibex's oblivious bidding protocol achieves the properties defined in Section 5.3.*

We give the full proof in Appendix B.2.

Above we describe the basic version of our bidding protocol. The limitation is that bidders need to pre-generate the bids and the bidding database is static. Frequently updating the bidding database can invalidate the cached encrypted bid shares of a browser. Besides, the bidder might want to customize its bids based on other dynamic factors, such as who the publisher is, and the time of day. We discuss some ideas to support these features in Section 9.

6 INTEGRATE *IBEX* FOR AGGREGATION

We classify the aggregation tasks into two categories, first-site aggregation and cross-site aggregation. We discuss how we can apply *Ibex*'s private histogram aggregation for the two kinds of aggregation tasks.

6.1 First-site aggregation

In first-site aggregation tasks, the user activities, such as viewed pages, are known to the aggregators (advertisers). Advertisers want to understand the activities of different groups of users, such as their browsing and purchasing preferences on their sites. For example, an advertiser needs to understand the interest of groups of users in particular items as in the *user profile generation* phase. To do so, the advertisers first classify the contents on their sites (e.g., pages or items) into different categories. For example, advertisers can classify their pages or items into the 392 categories from the Internet Advertising Bureau's (IAB) contextual taxonomy [15]. For different types of content, the advertiser maintains one encrypted histogram which includes how many times a certain group of users has viewed this type of content.

As the advertiser sees all activities of users on its site, the user only needs to split and share its group id with the aggregators. When a browser visits an advertiser's site for the first time, it uses the private histogram aggregation protocol to send its group id privately while the advertiser works as the main aggregator and the ad platform works as the auxiliary aggregator. The advertiser obtains the encrypted group of the user, issues a first-site cookie to the browser, and uses the cookie as an identifier to record this user's visits or purchases on its site. When the browser visits the advertiser's site again before the cookie expires, it does not recompute its group id; it only needs to include the assigned cookie.

For each activity (viewed content) of a user, the advertiser adds the encrypted group id to the encrypted histogram of that type of content. For example, if a user visits sports-related pages three times, the advertiser can add the encrypted group id of this user three times to the encrypted histogram of sports contents. After the advertiser aggregates enough aggregation reports (e.g., all reports in one day), the advertiser submits each encrypted histogram to the ad platform for decryption.

The process above enables each advertiser to understand user interests on its site. Appendix D also provides a way for advertisers to understand the user interests by combining aggregation results on different sites as needed.

6.2 Cross-site aggregation

In many cases, aggregation needs to account for actions of users across different sites. For example, during the *conversion measurement* phase, one might want to attribute an action (e.g., conversion) on one site to some activity that happened on another site. This requires the browser to locally record important activities on different sites, such as the click of an ad. To do this, *Ibex* follows an existing conversion measurement proposal [32]. In short, when the user performs an activity such as a purchase at an advertiser's site, the advertiser sends the browser a request for an attribution report. The browser uses its local history to attribute this conversion to a previously clicked or seen ad of the advertiser and asynchronously sends the report.

Different from the original proposal that asks browsers to send reports in plaintext, the browser uses the private histogram protocol to send its attribution report in a way that hides the identity of the publisher that led to the user's visit and subsequent conversion. The report contains the user's group id and the id of the publisher's site. Each combination of the group and id of the publisher's site corresponds to a unique report number. For example, if the advertiser's ads are displayed on N publishers and there are d user groups, there will be $N \cdot d$ types of reports in total. For each kind of user action (e.g., conversion), the advertiser maintains one encrypted histogram. After the advertiser aggregates enough reports, the advertiser submits each encrypted histogram to the ad platform for decryption.

7 EVALUATION

In this section, we would like to answer the following questions:

(1) What are the costs of *Ibex*'s private histogram and oblivious bidding for each party?
(2) How does *Ibex*'s private histogram affect the user's browsing response time compared to the non-private method?
(3) What is the end-to-end latency of *Ibex*'s oblivious bidding and how does it compare to non-private auction?

7.1 Implementation and evaluation setting

We answer the above questions in the context of the following implementations and evaluation environment.

***Ibex*'s implementation.** *Ibex* consists of about 2.5K lines of C++. In private histogram aggregation, we use SEAL's [36] implementation of the BFV scheme [60] as the HE scheme and OpenSSL 3.0.0 [16] for our basic cryptographic operations, and instantiate the signature algorithm with RSA. In oblivious bidding, we use the open-sourced implementation [19] of SealPIR [40] as the single-server PIR scheme and use the GMP library 6.2.1 [21] to implement Goldwasser-Micali (GM) encryption scheme [63, 64] with 2048-bit modulus. We implement the semi-honest two-party private auction protocol using the sh2pc protocol [5] in EMP toolkit [90]. To reduce the latency of fetching bid shares from the PIR servers, we split the bidding database into eight chunks, and process the PIR query for each chunk on a different core.

Why use a bit-homomorphic cryptosystem? We need each auctioneer to receive a secret share of bidders' bids, and these shares need to be defined over the plaintext space of a homomorphic cryptosystem so that we can apply the mask (Step 2 in Section 5.4). The GM cryptosystem satisfies this. We can also use additive or multiplicative shares and an additive or multiplicatively homomorphic cryptosystems, and then define the auction using an arithmetic MPC. In any case, the constraints we face are that (1) we want ciphertexts to be as small as possible since we store them in the PIR database (so larger ciphertexts increase PIR's costs); and (2) we need the MPC to be defined over the same ring or field as the plaintext space of the homomorphic cryptosystem to avoid performing expensive modular reductions inside the MPC.

Method and metrics. Besides the microbenchmarks, our key metrics are the response time of a browser's HTTP requests to the aggregator's web server and the end-to-end latency of the oblivious bidding. The oblivious bidding includes the events after the browser locally selects the advertisers for auction, but before the browser fetches and displays the winner's ad.

Evaluation environment. We run all our experiments on AWS c5.4xlarge instances (8-core Intel Xeon Platinum 8000 series processor with hyper-threading and 32 GB RAM) running Ubuntu 20.04. To measure the response time to HTTP requests of the private histogram when the browser visits the aggregator's site, we run the client in US East (Ohio) and the server in US West (California). To measure the end-to-end latency of oblivious bidding, we run the client in US East (Ohio), the advertisers' PIR servers and one auction server in US West (California), and another auction server in US West (Oregon). We use one c5.24xlarge instance (48-core machine) to run 6 advertisers' PIR servers.

Group size	2^{14}	2^{15}	2^{16}
Browser costs			
Generate shares (μs)	0.38	0.37	0.43
Share size (bit)	14	15	16
Auxiliary aggregator costs			
Encrypt and sign (ms)	25.58	47.05	91.33
Encrypted share and signature size (MB)	0.66	1.35	2.72
Decrypt aggregation result (ms)	3.35	7.12	14.81
Main aggregator costs			
Validate and recover encrypted report (ms)	2.20	4.50	9.23
Aggregation (ms)	0.36	0.84	1.90
Encrypted report size (MB)	0.66	1.35	2.72

Figure 4: Microbenchmarks for operations of each party in private histogram aggregation, the reported numbers are mean over results of 10 trials.

Parameters. We experiment with the user group size close to the FLoC's trial experiment [33]. For private histogram, we choose the HE parameters in such a way that it can support the aggregation of 2^{27} encrypted reports (more than one hundred million), which is sufficient to handle the daily visits to the New York Times [20]. For oblivious bidding, we experiment with 24 invited bidders in an auction and 14-bit bids, which is consistent with disclosed reports from ad exchanges [94, 95].

7.2 Microbenchmarks: costs of each party

7.2.1 Private histogram aggregation. We report our microbenchmark evaluation results in Figure 4, with varied group size. Most computation of each party can be computed off the critical path. And we detail the costs of each party below.

Browser's costs. An *Ibex*-enabled browser asynchronously computes its group id locally. And the group id can be recomputed once a week or so based on FLoC's trial [92]. The only computation on the browser is to locally encode and split its group id into shares. The shares have the same bit length as the group size and the time to generate shares is negligible. It first sends one share to an auxiliary aggregator by attaching the share in its HTTP request to obtain the encrypted report share and signature. It then appends the encrypted report share and signature in the request to the main aggregator. The average size of these materials is 0.66, 1.35, and 2.72 MB for group size of 2^{14}, 2^{15}, and 2^{16} respectively.

Auxiliary aggregator's costs. For each received report share, the auxiliary aggregator encrypts the share and signs the encrypted share. The most costly part of this operation is one homomorphic encryption to encrypt the share. The time to encrypt and sign the share and the size of generated materials grow linearly with the group size. For an aggregation task with the group of 2^{16}, such materials are 2.72 MB and it takes about 91.33 ms to generate. The auxiliary aggregator is also responsible to decrypt the encrypted aggregation result (histogram) for the main aggregator. The time of decryption also grows linearly with the size of the group and it takes 14.81 ms to decrypt with a group of size 2^{16}.

Main aggregator's costs. For each report, the main aggregator first validates and recovers the encrypted report locally. Costs of this operation grow linearly with the group size and it takes 9.23

Group size	2^{14}	2^{15}	2^{16}
Browser costs			
Generate one PIR query (ms)	1.94	1.70	1.75
One PIR query size (KB)	90.75	90.86	90.73
Decode PIR reply (ms)	1.42	1.42	1.41
Bidder costs			
Process one PIR query (ms)	230.02	375.70	650.46
One PIR reply size (KB)	181.05	181.19	181.15

Number of bidders	6	12	24
Browser costs			
Randomize bidding shares (ms)	0.27	0.55	1.1
Size of bidding shares (KB)	21	42	84
Auction server costs			
Decrypt bidding shares (ms)	0.88	1.77	3.53
Private auction (ms)	19.64	19.83	19.80

Figure 5: Microbenchmark for operations of each party in oblivious bidding with varied group sizes and varied number of bidders in an auction. The costs of PIR operations are the costs on a database chunk. For example, the entire database consists of 2^{16} rows and it is split into 8 smaller chunks that have 2^{13} rows each. The numbers are the mean over 10 trials.

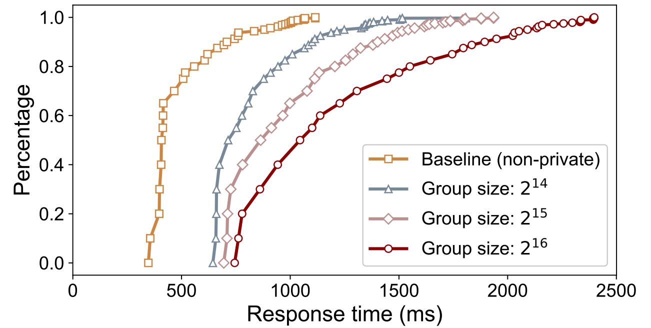

Figure 6: CDF (cumulative distribution function) of the response time of HTTP requests to a single-threaded web server using *Ibex*'s private aggregation under different group sizes and a non-private aggregation method. For each setting, we sample 300 data points where the HTTP requests are issued at the same rate which is far below the rate that will saturate the server.

ms with group size of 2^{16}. Aggregating recovered encrypted reports is a lot cheaper as it only requires ciphertext additions, and it only takes a few milliseconds to sum up two encrypted reports. Adding randomness to the final aggregation result is a one-time cost when the main aggregator needs to decrypt the aggregation result and this operation shares the same cost as one homomorphic encryption and summing up two ciphertexts. Removing the randomness is also a one-time cost and takes several milliseconds. The recovered encrypted report has the same size as the encrypted share.

7.2.2 Oblivious bidding. Figure 5 shows the costs of each party of private bidding with varied group sizes and we detail them below.

Browser's costs. In oblivious bidding, a browser first locally selects bidders to join the auction. It then concurrently issues PIR requests to the selected bidders' PIR servers. The time of generating one PIR request and the size of each request is constant for varied group sizes since we use the same set of parameters for PIR. Generating one request takes less than 2 ms and the size of one request is around 90 KB. To decode the PIR reply from the PIR servers, it takes about 1.4 ms to recover the encrypted bid shares. After retrieving the shares the browser randomizes these shares, and the time grows linearly with the number of invited bidders; the size of the bid shares is only tens of KB. It takes 1.1 ms to randomize the bid shares for an auction with 24 bidders.

Bidder's costs. The bidder splits its bidding databases into 8 chunks. It processes the PIR query on all 8 chunks in parallel. The time to answer one PIR query is linear to the group size while the reply size is constant. For each chunk, the bidder can answer a query in 650 ms for a group size of 2^{16}. The bidder can further split the entire database into more chunks to reduce the latency of answering PIR queries, but it requires more cores.

Auction server's costs. Auction servers receive the randomized shares from the browsers. They each first decrypt the randomized shares and then run the private auction protocol. The time to decrypt shares is linear to the number of invited bidders. It takes 0.88 ms, 1.77 ms, and 3.53 ms for 6, 12, and 24 advertisers respectively. The local computation time of the private auction remains roughly the same with the varied number of bidders and takes around 20 ms per auction server.

7.3 Browsing response time comparison

Most operations of *Ibex*'s private aggregation protocol can be computed off the critical path. The part that influences user experience most is response time when the user visits an aggregator's site. The private aggregation requires the browser to attach additional materials in the request when visiting the aggregator's site while the non-private methods [22, 92] directly expose the user's group id in plaintext. To compare the two, we build a single-threaded web server implemented in Python that receives HTTP requests from the clients and responds with a 1 MB webpage, and use wrk2 [26] to benchmark the response time of browsers' HTTP requests. The browser attaches the group id or the additional materials in the payload of POST requests. Figure 6 shows the CDF of the response time of HTTP requests under different settings. The median response time of the browser using *Ibex*'s private aggregation is 1.7–2.5× slower than that of the non-private method.

7.4 End-to-end latency of oblivious bidding

Figure 7 shows the end-to-end latency of *Ibex*'s oblivious bidding compared with the non-private method. With 24 invited bidders, the non-private auction takes 300 ms to complete while *Ibex*'s oblivious bidding takes 1.54 sec, 1.78 sec and 2.25 sec with group sizes of 2^{14}, 2^{15}, and 2^{16} respectively. This is 4.97–7.26× more costly than the non-private auction. The browser needs to send around 17.1 MB of extra data (PIR queries and the bid shares) with 24 invited bidders.

Note that the browser might have cached the bid shares of some bidders locally. If the bidder has not updated its bidding database since the last query of the browser, the browser can reuse the value

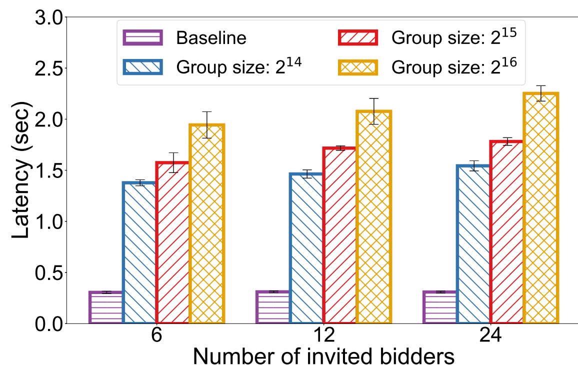

Figure 7: End-to-end latency of *Ibex*'s oblivious bidding and auction protocol. The baseline (purple bar) is a non-private auction with bids provided in the clear.

in its cache. In the overly optimistic case when all bid shares are local and the browser does not need to issue PIR queries, oblivious bidding only takes around 550 ms to complete with 24 invited bidders, which is 1.8× slower than the non-private auction.

While these numbers are high, existing studies [1, 8, 28, 91] show that page loading takes several seconds today, so *Ibex* should be able to run the auction and display the ad asynchronously (using AJAX) without significantly impacting page load.

8 RELATED WORK

This section describes other efforts that relate to *Ibex*.

8.1 Privacy-preserving advertising

AdVeil [87] uses Tor [56] and anonymous tokens to hide the identity of the user submitting a report, while ObliviAd [42] performs all adverting operations (ad selection, reporting) in a TEE. *Ibex* does not require each user to be equipped with Tor, nor does it use TEEs which are riddled with vulnerabilities [52, 54, 74, 83, 89, 93]. PPAD [48] computes statistics at the granularity of groups, but reveals which group a user belongs to (which *Ibex* does not do). BAdASS [70] assumes that bidders use a linear model to privately generate a bid using secret shares of the user profile. In contrast, *Ibex* bidders can choose arbitrary bids for each group. IPA [37] has a way to measure conversions on blinded ids but does not discuss how to use this mechanism to allow bidders to adjust their estimation of user value.

8.2 Private or verifiable auctions

Parkes et al. [85] and VEX [41] provide auction integrity but the auctioneer learns the bids. Other works [69, 72] hide the bids but lack integrity. Addax [97] provides both. Some designs [43, 51] run MPC among the bidders. These works are orthogonal to *Ibex*.

9 DISCUSSION

This section discusses potential extensions and improvements.

Bottleneck of *Ibex* in practice. *Ibex*'s costs during the auction is still more expensive than current non-private methods. The main costs come from bidders' PIR servers processing queries. One possibility is to experiment with other single-server PIR protocols such as Spiral [80], PIR schemes that do preprocessing for cheaper computation [71], or fast multi-server PIR schemes [49, 73, 79, 82] with proper deployment of multiple PIR servers.

Hide winner's identity. In the oblivious bidding, auction servers in the end learn which winner's ad is viewed by the user. Combining multiple winners of auctions for the same user allows the auction server to infer the user's interest. One way to limit the auction server's view is to hide the winner's identity from the auction servers. After learning the winner's index, the browser directly fetches an ad tag from the winner and displays the ad. Instead of directly sending both auction servers the winner's identity, the browser could split the identity into two shares (the identity can be the id of the winner among all advertisers), and send each auction server one share. The auction servers input the shares of the winner's id and the sale price of each auction and run a two-party computation (2PC) program that outputs the batch charges of each bidder without leaking the winner of each auction.

Dynamic features during bidding. Instead of just relying on the user profile, bidders may want to bid based on information about the publisher's site (where on the page the ad is shown), time of day when a user visits, etc. To address this issue, the bidder can set up a different bidding database for each combination of features. For example, a bidder sets up a bidding database for the publisher of news sites and morning visits. The browser chooses the bidding database it needs to read according to its visit. The bidders can decide which features of bidding are sensitive to a user and include that feature in the database. For example, each row in the database represents the bidding share for a combination of a type of publisher site and user group.

Other tracking methods. While *Ibex* prevents tracking via group identifiers or third-party cookies, there are other more noisy tracking methods such as browser fingerprinting [25] and IP addresses. Ongoing efforts specifically address these tracking issues, such as Apple's private relay service [9] that hide IP address and browsing activities in Safari, can be integrated with *Ibex*.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Amrita Roy Chowdhury, for their helpful comments that improved the content and presentation of this work. This work was funded in part by NSF grant CNS-2045861 and DARPA contract HR0011-17-C0047.

REFERENCES

[1] About pagespeed insights. https://developers.google.com/speed/docs/insights/v5/about.
[2] Aggregation service for the attribution reporting api. https://github.com/WICG/attribution-reporting-api/blob/main/AGGREGATION_SERVICE_TEE.md.
[3] Back to basics: What is header bidding? https://www.lotame.com/back-basics-header-bidding/.
[4] Divvi up: A privacy-respecting system for aggregate statistics. https://divviup.org/.
[5] EMP sh2pc. https://github.com/emp-toolkit/emp-sh2pc.
[6] Google Has a New Plan to Kill Cookies. People Are Still Mad. https://www.wired.co.uk/article/google-floc-cookies-chrome-topics.
[7] Google's Topics API: Rebranding FLoC Without Addressing Key Privacy Issues. https://brave.com/web-standards-at-brave/7-googles-topics-api/.

[8] Here's what we learned about page speed. https://backlinko.com/page-speed-stats.
[9] icloud private relay overview. https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf.
[10] Internet security research group. https://abetterinternet.org/.
[11] Let's encrypt: A nonprofit certificate authority providing tls certificates to 260 million websites. https://letsencrypt.org/.
[12] Masked learning, aggregation and reporting workflow (masked lark). https://github.com/WICG/privacy-preserving-ads/blob/main/MaskedLARK.md.
[13] Multi-party computation of ads on the web (macaw). https://github.com/WICG/privacy-preserving-ads/blob/main/MACAW.md.
[14] Number of active advertisers on Facebook from 1st quarter 2016 to 3rd quarter 2020. https://www.statista.com/statistics/778191/active-facebook-advertisers/.
[15] Openrtb protocol buffer 2.5.0. https://developers.google.com/authorized-buyers/rtb/downloads/openrtb-proto.
[16] OpenSSL. https://www.openssl.org.
[17] Parakeet. https://github.com/WICG/privacy-preserving-ads/blob/main/Parakeet.md.
[18] Privacy analysis of FLoC. https://blog.mozilla.org/en/mozilla/privacy-analysis-of-floc/.
[19] SealPIR: A computational PIR library that achieves low communication costs and high performance. https://github.com/microsoft/SealPIR.
[20] Similarweb. https://www.similarweb.com.
[21] The GNU Multiple Precision Arithmetic Library. https://gmplib.org/gmp6.2.
[22] The Topics API. https://github.com/patcg-individual-drafts/topics/.
[23] This is how Google plans to track you now. https://www.slashgear.com/this-is-how-google-plans-to-track-you-now-25708910/.
[24] Understand your conversion tracking data. https://support.google.com/google-ads/answer/6270625.
[25] What is fingerprinting and why you should block it. https://www.mozilla.org/en-US/firefox/features/block-fingerprinting/.
[26] wrk2: a http benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2.
[27] Cookie synching. https://www.admonsters.com/cookie-synching/, 2010.
[28] Find out how you stack up to new industry benchmarks for mobile page speed. https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf, 2017.
[29] Cookie matching. https://developers.google.com/authorized-buyers/rtb/cookie-guide, 2020.
[30] Lattigo v2.1.1. Online: http://github.com/ldsec/lattigo, Dec. 2020.
[31] PALISADE Lattice Cryptography Library (release 1.10.6). https://palisade-crypto.org/, Dec. 2020.
[32] Attribution reporting api. https://github.com/WICG/conversion-measurement-api, 2021.
[33] FLoC origin trial & clustering. https://www.chromium.org/Home/chromium-privacy/privacy-sandbox/floc, 2021.
[34] Private aggregation. https://github.com/WICG/conversion-measurement-api/blob/main/SERVICE.md, 2021.
[35] Fledge api. https://developer.chrome.com/docs/privacy-sandbox/fledge/, 2022.
[36] Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL, Mar. 2022.
[37] Privacy preserving attribution for advertising. https://blog.mozilla.org/en/mozilla/privacy-preserving-attribution-for-advertising/, 2022.
[38] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. Cryptology ePrint Archive, Paper 2021/576, 2021. https://eprint.iacr.org/2021/576.
[39] E. Anderson, M. Chase, F. B. Durak, E. Ghosh, K. Laine, and C. Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Paper 2021/1490, 2021. https://ia.cr/2021/1490.
[40] S. Angel, H. Chen, K. Laine, and S. Setty. Pir with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.
[41] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference*, 2013.
[42] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.
[43] S. Bag, F. Hao, S. F. Shahandashti, and I. G. Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15, 2020.
[44] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1991.
[45] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.
[46] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.
[47] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.
[48] S. T. Boshrooyeh, A. Küpçü, and Ö. Özkasap. Ppad: Privacy preserving group-based advertising in online social networks. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, 2018.
[49] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
[50] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2012.
[51] F. Brandt. A verifiable, bidder-resolved auction protocol. In *Proceedings of the 5th International Workshop on Deception, Fraud and Trust in Agent Societies*, 2002.
[52] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.
[53] D. Chaum and E. van Heyst. Group signatures. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1991.
[54] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.
[55] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
[56] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, 2004.
[57] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2006.
[58] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4), 2014.
[59] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4), 1985.
[60] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.
[61] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2012.
[62] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2014.
[63] S. Goldwasser and S. Micali. Probabilistic encryption; how to play mental poker keeping secret all partial information. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1982.
[64] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2), 1984.
[65] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
[66] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
[67] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2009.
[68] S. Halevi and V. Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. https://eprint.iacr.org/2020/1481.
[69] M. Harkavy, J. D. Tygar, and H. Kikuchi. Electronic auctions with private bids. In *3rd USENIX Workshop on Electronic Commerce (EC 98)*, 1998.
[70] L. J. Helsloot, G. Tillem, and Z. Erkin. Badass: Preserving privacy in behavioural advertising with applied secret sharing. In *Provable Security*, 2018.
[71] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.
[72] H. Kikuchi, S. Hotta, K. Abe, and S. Nakanishi. Distributed auction servers resolving winner and winning bid without revealing privacy of bids. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, 2000.
[73] D. Kogan and H. Corrigan-Gibbs. Private blocklist lookups with Checklist. In *Proceedings of the USENIX Security Symposium*, 2021.
[74] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the*

USENIX Security Symposium, 2017.
[75] H. Liao, L. Peng, Z. Liu, and X. Shen. Ipinyou global rtb bidding algorithm competition dataset. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, 2014.
[76] B. Libert, S. Ling, F. Mouhartem, K. Nguyen, and H. Wang. Signature schemes with efficient protocols and dynamic group signatures from lattice assumptions. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2016.
[77] Y. Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://ia.cr/2016/046.
[78] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 2013.
[79] Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR. In *Proceedings of the USENIX Security Symposium*, 2022.
[80] S. J. Menon and D. J. Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.
[81] I. Mironov. Rényi differential privacy. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2017.
[82] H. Mozaffari and A. Houmansadr. Heterogeneous private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.
[83] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
[84] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999.
[85] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. *Electronic Commerce Research and Applications*, 2008.
[86] E. Rescorla and M. Thomson. Technical comments on FLoC privacy. https://mozilla.github.io/ppa-docs/floc_report.pdf, 2021.
[87] S. Servan-Schreiber, K. Hogan, and S. Devadas. Adveil: A private targeted-advertising ecosystem. Cryptology ePrint Archive, Report 2021/1032, 2021. https://eprint.iacr.org/2021/1032.
[88] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.
[89] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.
[90] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.
[91] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
[92] Y. Xiao and J. Karlin. Federated learning of cohorts. https://wicg.github.io/floc/, 2021.
[93] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.
[94] S. Yuan, J. Wang, B. Chen, P. Mason, and S. Seljan. An empirical study of reserve price optimisation in real-time bidding. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014.
[95] W. Zhang, S. Yuan, J. Wang, and X. Shen. Real-time bidding benchmarking with ipinyou dataset. https://arxiv.org/abs/1407.7073, 2015.
[96] K. Zhong, Y. Ma, and S. Angel. Ibex: Privacy-preserving ad conversion tracking and bidding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
[97] K. Zhong, Y. Ma, Y. Mao, and S. Angel. Addax: A fast, private, and accountable ad exchange infrastructure. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

```
1:  function Priv-Auction(B_1^1, ..., B_k^1, B_1^2, ..., B_k^2)
2:      # Auction server one inputs its bidding shares B_1^1, ..., B_k^1.
3:      # Auction server two inputs its bidding shares B_1^2, ..., B_k^2.
4:
5:      # Reconstruct the bids privately using bid shares.
6:      for i in range(k) do
7:          B_i = B_i^1 ⊕ B_i^2
8:      # Compute auction and find the winner with highest bid.
9:      bs = max(B_i) for i ∈ [1, k]
10:     winner = index_of_bs
11:
12:     Output (bs, winner) to two auction servers
```

Figure 8: Pseudocode of *Ibex*'s private auction protocol. k **is the number of invited bidders in the auction.**

A SUBROUTINES OF OBLIVIOUS BIDDING

A.1 Private auction protocol design

In *Ibex*, we represent bids as bit strings and define the auction protocol as an MPC over a boolean circuit using the emp-toolkit [90]. We give the pseudocode in Figure 8.

A.2 Bit additively homomorphic cryptosystem

Prior works [63, 64] give constructions for additively homomorphic encryption for bits. Specifically, the scheme encrypts a single bit and supports a group operation over ciphertexts that results in the XOR of the underlying plaintexts. It consists of the following algorithms.

- BIT-HE-KeyGen$(1^\lambda) \longrightarrow (pk, sk)$. Takes in a security parameter λ of the encryption scheme and outputs a public key pk and a secret key sk.
- BIT-HE-Enc$(pk, b) \longrightarrow (c)$. Takes in a public key pk and a plaintext bit b and outputs a ciphertext c.
- BIT-HE-Dec$(sk, c) \longrightarrow (b)$. Takes in a secret key sk and a ciphertext c and outputs a plaintext bit b.
- BIT-HE-Add$(c_1, c_2) \longrightarrow (c_{add})$. Takes in two ciphertexts, c_1 and c_2 and outputs a ciphertext c_{add}, such that c is the ciphertext of the XOR of the two bits underlying c_1 and c_2.

A.3 Construction

Each bid value B in our bidding protocol is represented as a length-ℓ bit string. We use $B[i]$ to denote the i-th bit in B. When encrypting, we encrypt bit by bit. Below we define algorithms used in the construction of the oblivious bidding.

- Share-Enc$(pk, B) \longrightarrow (C)$.
  (1) Initialize a length-ℓ ciphertext vector C_{add}.
  (2) For $i \in [0, \ell - 1]$: set $C[i] \longleftarrow$ BIT-HE-Enc $(pk, B[i])$.
  (3) Output C.
- Share-Dec$(sk, C) \longrightarrow (B)$.
  (1) Parse C as $(C[0], \ldots, C[\ell - 1])$.
  (2) Set an ℓ bit value B to be 0.
  (3) For $i \in [0, \ell - 1]$: set $B[i] \longleftarrow$ BIT-HE-Dec $(sk, C[i])$.
  (4) Output B.
- Share-Add$(C_1, C_2) \longrightarrow (C_{add})$.
  (1) Parse C_1 as $(C_1[0], \ldots, C_1[\ell-1])$ and C_2 as $(C_2[0], \ldots, C_2[\ell-1])$.
  (2) Initialize a length-ℓ ciphertext vector C_{add}.
  (3) For $i \in [0, \ell - 1]$: set $C_{add}[i] \longleftarrow$ BIT-HE-Add$(C_1[i], C_2[i])$.
  (4) Output C_{add}.

B PROOF FOR *IBEX*'S PROPERTIES

B.1 Proof of private histogram's properties

Correctness. There are d groups in total; for each group i, there are N_i users. The desired output is a vector of $[N_0, N_1, \ldots, N_{d-1}]$. When all parties follow the protocol, for each report from group g, the sum of two additive shares produced by Additive-Shares is either g or $g + d$ in $\mathbb{Z}_{2d}$. Thus, the main aggregator receives N_i encryptions of either (x^i) or (x^{i+d}). It sums up these ciphertexts to produce a ciphertext which is the encryption of the polynomial $(\sum_{i=1}^{2d} \beta_i \cdot x^i)$, where β_i is the coefficient of i-th term in the polynomial. And for each $i \in [0, d-1]$, $\beta_i + \beta_{i+d} = N_i$.

The main aggregator removes the added random mask in Step 3. Thus, the effect of the random mask is offset in the final output. The final histogram the main aggregator obtains is $[\beta_0 + \beta_d, \ldots, \beta_i + \beta_{i+d}, \ldots, \beta_{d-1} + \beta_{2d-1}]$. For all $i \in [0, d-1]$, $\beta_i + \beta_{i+d} = N_i$, the main aggregator obtains the correct histogram.

Robustness. The shares received by the main aggregator and the auxiliary aggregator are both values in $\mathbb{Z}_d$. Here a malicious browser cannot change the histogram, and the sum of the shares defines its input. Since both aggregators are honest, they will convert the value to a correct polynomial. The only way that a malicious browser can change the histogram is by adding plaintexts to the polynomial sent by the auxiliary aggregator. This is not possible by the security of the signature scheme.

Privacy. We first analyze what an auxiliary aggregator learns in the private histogram protocol. The auxiliary aggregator receives report shares from different browsers and encrypted histograms added with randomness from main aggregators. The report shares are uniformly random thus leaking no information about the original reports of the users. When the browser sends materials to its target main aggregator, the auxiliary aggregator does not communicate with the main aggregator at all thus it cannot link a user with a target main aggregator. When a main aggregator asks the auxiliary aggregator to decrypt a histogram encrypted in a ciphertext c, the main aggregator first adds (homomorphically) a uniformly random mask to the content of c. When the auxiliary aggregator decrypts c, it will obtain a uniformly random value and will not learn the main aggregator's histogram. Thus, the auxiliary aggregator does not learn anything about the users' reports, links between the users and main aggregators, and the histograms of the main aggregators.

Now we analyze what a main aggregator learns in the private histogram protocol. We do so by using a simulation proof [77]. We denote the main aggregator as $\mathcal{M}$ and the auxiliary aggregator as $\mathcal{X}$. The number of reports for $\mathcal{M}$ is denoted as N. For simplicity, we assume each report is sent from a unique user and these are denoted as $\mathcal{B}_1, \ldots, \mathcal{B}_N$. We first give the definition of an ideal function of the private histogram aggregation protocol.

Ideal functionality $\mathcal{F}$ of private histogram aggregation

Inputs: N reports, $G_1, \ldots, G_N$ where $G_i \in [0, d-1]$, from users, $\mathcal{B}_1, \ldots, \mathcal{B}_N$, respectively.
Outputs: $\mathcal{M}$ learns the histogram of reports, h (i.e., $[N_0, N_1, \ldots, N_{d-1}]$). The histogram's j-th bar N_j is the number of reports belonging to group j. $N_j = \sum_{j \in [N]} \mathbb{I}(G_i = j)$, where $\mathbb{I}$ is the identity function, when the condition holds it equals 1, otherwise it's 0.

Below we give a formal description of *Ibex*'s private aggregation protocol. Note that we only care about privacy of honest users, so in the description the materials for aggregation that users send to $\mathcal{M}$ and $\mathcal{X}$ are all valid.

***Ibex*'s private histogram aggregation**

Step 1 ($\mathcal{X}$ generate keys):

- $\mathcal{X}$ runs HE-KeyGen to generate the public and secret keys, *pk* and *sk*, and runs Sig-KeyGen to generate the signature and verification keys, *vrfkey* and *sigkey*.
- *pk* and *vrfkey* are sent to all main aggregators.

Step 2 (Users generate report shares):

- For $i = 1$ to N: each user $\mathcal{B}_i$ runs Additive-Shares(G_i) to produce the two shares, S_i^1, S_i^2, of its report G_i, and sends its first share S_i^1 to $\mathcal{X}$.

Step 3 ($\mathcal{X}$ encrypts and signs):

- For each received share S_i^1, $\mathcal{X}$ first validates the share is in valid range of $[0, d-1]$ and generates the encrypted share C_i^1 = HE-Enc($pk, x^{S_i^1}$).
- $\mathcal{X}$ generates the signature Sig_i = Sign($sigkey, C_i^1$)
- $\mathcal{X}$ sends back the C_i^1 and Sig_i to $\mathcal{B}_i$.

Step 4 ($\mathcal{M}$ recovers encrypted aggregation result):

- Each user $\mathcal{B}_i$ sends (S_i^2, C_i^1, Sig_i) to $\mathcal{M}$.
- $\mathcal{M}$ runs Verify($vrfkey, Sig_i$) to validate the signature, checks S_1^2 is in the valid range of $[0, d-1]$, and computes the encrypted report CG_i on the valid shares by computing HE-Mul-Plain($C_i^1, x^{S_i^2}$).
- $\mathcal{M}$ computes encrypted histogram $cAgg = \sum_{i=1}^N CG_i$. It generates uniform randomness r and computes $mAgg$ = HE-Add-Plain($cAgg, r$).

Step 5 ($\mathcal{M}$ learns decrypted histogram):

- $\mathcal{M}$ sends *mAgg* to $\mathcal{X}$ for decryption. $\mathcal{X}$ decrypts to get the polynomial $dAgg$ = HE-Dec($sk, mAgg$) and sends *dAgg* back to $\mathcal{M}$.
- $\mathcal{M}$ computes the polynomial $agg = dAgg - r$. *agg* is decoded to the real histogram h (i.e., the vector of $[N_0, N_1, \ldots, N_{d-1}]$)

Simulation for main aggregator. We now build the simulator *Sim* for main aggregator $\mathcal{M}$ and use $\mathcal{A}$ to denote an adversary who corrupts $\mathcal{M}$ in the following simulation.

Simulator *Sim* for main aggregator $\mathcal{M}$

Step 1 (Generate keys):

- *Sim* runs HE-KeyGen to generate a pair of public and secret keys, pk' and sk', and runs Sig-KeyGen to generate a pair of signature and verification keys, $sigkey'$ and $vrfkey'$.
- *Sim* sends pk' and $vrfkey'$ to $\mathcal{A}$.

Step 2 (Generate random shares):

- $\mathcal{F}$ outputs h (i.e., $[N_0, N_1, \ldots, N_{d-1}]$) to *Sim*.
- *Sim* randomly assigns a value to a report G_i' for $i = 1$ to N. And the numbers of reports in each group satisfy the histogram ($[N_0, N_1, \ldots, N_{d-1}]$).
- *Sim* calls Additive-Shares to generate two shares ${S'}_i^1, {S'}_i^2$ on each reports G_i'.

Step 3 (Send encrypted shares and signature to $\mathcal{A}$):

- For each report share ${S'}_i^1$, *Sim* generates the encrypted shares ${C'}_i^1$ = HE-Enc($pk', x^{{S'}_i^1}$), and generates the signature ${Sig'}_i$ = Sign($sigkey', {C'}_i^1$) , for $i = 1$ to N.
- *Sim* sends $({S'}_i^2, {C'}_i^1, {Sig'}_i)$ to $\mathcal{A}$, for $i = 1$ to N.

Step 4: (Recover encrypted reports):

- For $i = 1$ to N, $\mathcal{A}$ runs Verify($vrfkey', {Sig'}_i$) to validate the signature, checks that ${S'}_i^2$ is in the valid range, and generates the encrypted report ${CG'}_i$ = HE-Mul-Plain(${C'}_i^1, x^{{S'}_i^2}$).
- $\mathcal{A}$ computes $cAgg' = \sum_{i=1}^N {CG'}_i$. It samples uniform randomness r' and computes $mAgg'$ = HE-Add-Plain($cAgg', r'$).
- $\mathcal{A}$ sends $mAgg'$ to *Sim* for decryption.

Step 5: (Compute histogram):

- *Sim* computes $dAgg'$ = HE-Dec($sk', mAgg'$) to decrypt $mAgg'$. It sends back the decrypted result $dAgg'$ to $\mathcal{A}$.
- $\mathcal{A}$ computes $agg' = dAgg' - r'$.

In the real world (*Ibex*'s histogram protocol), the view of $\mathcal{A}$ includes: $\{pk, vrfkey, (S_i^2, C_i^1, Sig_i)_{i \in [N]}, r, cAgg, mAgg, dAgg, agg, h\}$. In the ideal world (simulation using output from the ideal functionality $\mathcal{F}$), the view of the adversary $\mathcal{A}$ includes: $\{pk', vrfkey', ({S'}_i^2, {C'}_i^1, {Sig'}_i)_{i \in [N]}, r', cAgg', mAgg', dAgg', agg', h\}$.

HE public key pk and pk', signature verification key *vrfkey* and $vrfkey'$ are computationally indistinguishable. S_i^2 and ${S'}_i^2$ are uniform random numbers. C_i^1 and ${C'}_i^1$ are HE ciphertexts and give no information thus are indistinguishable. Sig_i and ${Sig'}_i$ are signatures on indistinguishable ciphertexts using indistinguishable signature keys thus are also indistinguishable. *cAgg* and $cAgg'$, *mAgg* and $mAgg'$, are both HE ciphertexts and indistinguishable. *agg* and agg' have the same distribution and are indistinguishable. r and r' are uniformly random. *dAgg* and $dAgg'$ have the same distribution and are indistinguishable as they are *agg* or agg' adding uniform randomness r or r'. The final histogram h is learned in both real and ideal worlds.

B.2 Proof of oblivious bidding's properties

This section discusses the security properties of *Ibex*'s oblivious bidding protocol: correctness, robustness, and privacy. One caveat for privacy is that we prove that a single auction leaks no information about the auctioned user's group, but we do not reason about the aggregate charge that *Ibex* gives bidders for a batch of auctions (e.g., how much the bidder needs to pay the auctioneers for the last 10,000 auctions).

Correctness. When all parties follow the protocol, the browser will correctly retrieve the encrypted bid shares from the selected bidders and add randomness to the encrypted shares without changing the bids of bidders. With these correct bidding shares, *Ibex*'s private auction protocol will correctly compute the winner's index and sale price.

Robustness. Malicious browsers may submit arbitrary encrypted bid shares to the auction servers or respond to the auction servers with an arbitrary bidder's identity as the winner. However, the bids that each bidder is willing to pay have been committed to the auction server by sending the shuffled list of all the bids. The auctions with a sale price that is not committed by the bidder will be detected and abandoned by the auction servers.

Privacy. The bidder does not learn the user's group id because the PIR protocol hides the row (group id) read by the browser. And the charge to an bidder is deferred and only made available later as the sum of a batch of charges. Our proof below focuses on a single auction and does not reason about what this sum of charges leaks about users (we discuss this separately in Step (4) of Section 5.4.1).

We prove that the honest user's group id won't be learned by the auction server in the oblivious bidding protocol using a simulation proof [77]. Let there be k invited bidders in one auction, and let users be grouped into d different groups. Each invited bidder is denoted $\mathcal{D}_i$ for $i = 1$ to k. The list of all the bids of $\mathcal{D}_i$ for different groups of users is denoted as BID_i, and $BID_i[g]$ indicates the bid of $\mathcal{D}_i$ for user from group g. The shuffled bid list of BID_i is denoted as $SBID_i$. The two auction servers are denoted as $\mathcal{S}_1$ and $\mathcal{S}_2$.

We first give the formal definition of an ideal functionality of one auction using the oblivious bidding protocol. As we only care about honest user's privacy, we assume the user's browser does not misbehave in the auction in the ideal functionality and the following simulation.

Ideal functionality $\mathcal{F}$ of oblivious bidding

Inputs: For one auction, the browser inputs its group id grp. For each invited bidder $\mathcal{D}_i$ where $i \in [1, k]$, the bidder inputs its bid BID_i. BID_i is a list of d bids where $BID_i[j]$ is $\mathcal{D}_i$'s bid for users from group j.

Outputs: $\mathcal{S}_1$ and $\mathcal{S}_2$ learn the outcome of the auction $o = (wid, w, bs)$ and the randomly shuffled list $SBID_i$ of each bidder's bid list (BID_i) for $i = 1$ to k. wid is the index of the winner among the k invited bidders, w is the identity of the winner (i.e., the index of the winner among *all* bidders in the system), and bs is the sale price of auction. The browser learns wid and the identities of the k invited bidders.

Below we give a formal description of *Ibex*'s oblivious bidding protocol for one auction.

Ibex's oblivious bidding

Step 1 (Auction servers generate keys):

- $\mathcal{S}_1$ and $\mathcal{S}_2$ use BIT-HE-KeyGen to generate public key and secret key for the bit additively homomorphic encryption scheme, pk_1, sk_1 and pk_2, sk_2 respectively.
- pk_1 and pk_2 are sent to all browsers, bidders, and auction servers.

Step 2 (Bidders set up bidding database):

- Each bidder $\mathcal{D}_i$, for $i = 1$ to k, computes Bid-Encode($BID_i[j]$), for a group j for $j = 1$ to d. For each bid $BID_i[j]$, Bid-Encode outputs $S_i^1[j]$ and $S_i^2[j]$, which are the two additive shares of the bit string of $BID_i[j]$.
- For $i = 1$ to k, $\mathcal{D}_i$ sets up its bidding database DB_i such that the j-th row (for $j = 1$ to d) of the database is the two encrypted bid shares under different keys, CS_i^1 =Share-Enc($pk_1, S_i^1[j]$) and CS_i^2 =Share-Enc($pk_2, S_i^2[j]$). DB_i is made public to all browsers, bidders, and auction servers.
- $\mathcal{D}_i$ sends the shuffled list of its bids $SBID_i$ to $\mathcal{S}_1$ and $\mathcal{S}_2$.

Step 3 (Browser fetches and randomizes bid shares):

- The browser belonging to group grp uses PIR-Read to read the grp-th column from each bidder's bidding database, and obtains the encrypted shares CS_i^1 and CS_i^2, for $i = 1$ to k.
- The browser adds randomness to the CS_i^1 and CS_i^2 to generate CR_i^1 and CR_i^2 following Step (2) in Section 5.4.1, for $i = 1$ to k
- Browser sends CR_i^1 to $\mathcal{S}_1$ and CR_i^2 to $\mathcal{S}_2$, $i = 1$ to k.

Step 4 (Servers privately compute auction):

- For $i = 1$ to k, $\mathcal{S}_1$ computes the decrypted share BR_i^1 = Share-Dec(sk_1, CR_i^1) and $\mathcal{S}_2$ computes the decrypted share BR_i^2 = Share-Dec(sk_2, CR_i^2).
- $\mathcal{S}_1$ and $\mathcal{S}_2$ run the private auction protocol Priv-Auction with inputs BR_i^1 and BR_i^2, for $i = 1$ to k. Priv-Auction outputs the winner's id wid and the sale price bs.

Step 5 (Servers check outcome of auction):

- $\mathcal{S}_1$ and $\mathcal{S}_2$ send wid to the browser.
- Browser notifies both servers of the identity w of the winning bidder.
- $\mathcal{S}_1$ and $\mathcal{S}_2$ check the outcome of the auction. If bs is not in the shuffled bid list of bidder w, the auction is marked as invalid, and sets the flag v to false. Otherwise, v is set to true.

Simulation for auction server. We now build the simulator Sim for one of the auction servers. For simplicity, we assume $\mathcal{S}_1$ is the semi-honest adversary and use $\mathcal{A}$ to denote an adversary who corrupts $\mathcal{S}_1$ in the following simulation.

Simulator *Sim* for one auction server

Step 1 (Generate keys):

- $\mathcal{A}$ uses BIT-HE-KeyGen to generate the public and secret keys, pk'_1 and sk'_1.
- *Sim* uses BIT-HE-KeyGen to generate the public and secret keys, pk'_2 and sk'_2 and sends pk'_2 to $\mathcal{A}$.

Step 2 (Set up bidding database and commit bids):

- $\mathcal{F}$ outputs $o = (wid, w, bs)$ and all shuffled bid lists $SBID_i$, for $i = 1$ to k, to *Sim*. *Sim* sends all $SBID_i$, for $i = 1$ to k, to $\mathcal{A}$.
- For $i = 1$ to k and $j = 1$ to d, *Sim* generates random bids $BID'_i[j]$, and calls Bid-Encode to generate two additive shares $S'^1_i[j]$ and $S'^2_i[j]$.
- For $i = 1$ to k and $j = 1$ to d, *Sim* sets up its bidding database DB'_i such that the j-th row of the database is the two encrypted bid shares, CS^1_i =Share-Enc($pk_1, S'^1_i[j]$) and CS^2_i =Share-Enc($pk'_2, S'^2_i[j]$). *Sim* sends DB'_i to $\mathcal{A}$.

Step 3 (Generate randomized bid shares):

- For $i = 1$ to k, *Sim* randomly generates bids B'_i such that $B_{wid} = bs$ and all other bids are smaller than bs.
- For $i = 1$ to k, *Sim* calls Bid-Encode on B'_i to generate two bid shares BR'^1_i and BR'^2_i.
- For $i = 1$ to k, *Sim* computes the encrypted bid shares CR'^1_i = Share-Enc(pk'_1, BR'^1_i) and sends CR'^1_i to $\mathcal{A}$.

Step 4 (Privately compute auction):

- For $i = 1$ to k, $\mathcal{A}$ computes the decrypted bid share BR'^1_i = Share-Dec(sk'_1, CR'^1_i).
- $\mathcal{A}$ inputs BR'^1_i and *Sim* inputs BR'^2_i for $i = 1$ to k to the ideal functionality of the two-party private auction protocol Priv-Auction to compute the auction. The outcome of the auction is wid and bs.

Step 5 (Check outcome of auction):

- *Sim* sends w to $\mathcal{A}$.
- $\mathcal{A}$ checks whether the shuffled bid list of bidder w contains bs. If not, $\mathcal{A}$ marks this auction as invalid and sets flag v' as false, otherwise it's true.

In the real world (*Ibex*'s oblivious bidding), the view of $\mathcal{A}$ includes: $\{(SBID_i, CR^1_i, DB_i, BR_i)_{i\in[k]}, pk_1, sk_1, pk_2, w, wid, bs, v\}$.

In the ideal world (simulation using output from the ideal functionality), the view of $\mathcal{A}$ includes: $\{(SBID_i, CR'^1_i, DB'_i, BR'_i)_{i\in[k]}, pk'_1, sk'_1, pk'_2, w, wid, bs, v'\}$.

The output wid, w, bs, and shuffled bid lists $SBID_i$ are learned in both real and ideal worlds. The decrypted bid shares BR_i and BR'_i are uniform random bit vectors, thus they are indistinguishable. CR^1_i and CR'^1_i are both ciphertexts encrypted using pk_1 which encrypt BR_i and BR'_i respectively. pk_1 and pk'_1, sk_1 and sk'_1, pk_2 and pk'_2 are random keys and computationally indistinguishable. Each row in DB_i and DB'_i consists of two encrypted shares. The shares encrypted using pk_2 or pk'_2 are indistinguishable. The shares encrypted with pk_1 or pk'_1 can be decrypted by $\mathcal{A}$ but they decrypt to be uniform random bit vectors thus are indistinguishable. As we simulate only for an honest user that follows the protocol, the verification of the outcome of an auction always passes (i.e., v and v' are always true).

C ALTERNATIVES OF PRIVATE HISTOGRAM

C.1 Multi-party histogram aggregation

In this section, we briefly describe several multi-party private histogram protocols [39, 47, 55] that can be potentially extended to implement the private histogram aggregation protocol in *Ibex*, and then discuss why they are not a great fit for our setting.

Prio [55] and Prio+ [38]. Prio and Prio+ are protocols that support privacy-preserving aggregation while protecting its functionality in the face of malicious inputs from clients. They require multiple non-colluding servers and supports various aggregation tasks such as average, min/max, and histogram. A client in their protocol needs to prove to the servers that its input satisfies a certain predicate, and in terms of histogram aggregation, the predicate checks that the input is a one-hot vector. That is, the client holding report r generates a vector of ℓ elements $[v_1, v_2, \ldots, v_\ell]$ where only the r-th element v_r is one and all other elements are zeros. Then its needs to generate a proof to show that all v_i is either 0 or 1, and $\sum_{i=1}^{\ell} v_i = 1$. Both proof size and the cost to verify the proof are linear to ℓ. And this leads to expensive communication costs from client to server as the proof size is linear to ℓ.

Poplar [47]. Poplar uses Distributed Point Functions (DPF) [62] to build a private hitter protocol, which can be used for private histogram aggregation. However, since Ibex needs a histogram with x-coordinates (i.e., the group) spanning over the entire domain, directly using Poplar requires doing a private hitter for each group. This means Poplar also suffers from similar linear costs as Prio and Prio+ when computing the histogram. Even though Poplar optimizes the client-to-server communication, the two aggregators still need to invoke ℓ DPF evaluation functions for a histogram with ℓ groups. In other words, Poplar is not tailored to computing this type of histograms. In addition, the two aggregators need to validate that the keys they receive are well-formed using a sketching protocol and then use the keys to construct the histogram.

Anderson et al. [39]. Anderson et al. propose a histogram aggregation protocol with differential privacy using three non-colluding servers under the semi-honest setting. A report is a number and each user splits its report into two additive shares and sends the two shares to the two designated aggregators respectively. These two aggregators will each hold a list of users' report shares. The two aggregators then involve another aggregator to shuffle the two lists of users' report shares. In the end, two aggregators will each hold one shuffled list of shares. To produce the histogram of all reports, each aggregator reveals its own shuffled list and sums up the two shares at the same index in the two lists. Summing up the two shares in the two lists at the same index produces a report of a user. And counting how many times a number occurs in the sums gives the histogram. As the list has been shuffled the aggregators cannot link one report with its original sender user.

All the above protocols require direct interactions between the two aggregation servers. If one uses them the way we envision in *Ibex*, this brings forth the *linkability* issue that we discuss in

Section 4.1. However, there is another way to use this systems, which is to change the roles of the aggregators to two separate ad platforms. In this way, the two aggregators can aggregate all of the reports. But this change has two drawbacks.

First, since the reports for all advertisers are sent to the same two aggregators, a report needs to at least include the advertiser id as an additional feature. Considering that the total of advertisers can be as large as 10 million (more than 2^{23}) [14], this can be expensive. For example, with a group size of 2^{16}, the value of ℓ used in Prio and Poplar would need to be at least 2^{39}. Based on prior estimates [39], Poplar could take 24 hours to aggregate a report while Prio would be even slower due to the expensive client-to-server communication. The approach of Anderson et al. [39] remains performance since additional features do not significantly impact its performance.

The second drawback is that since neither aggregator server is an advertiser (for whom the report is meant), we lose visibility into the veracity of reports. For example, an attacker can send bogus reports to the two aggregators that do not refer to a real conversion or a real human visit to the advertiser's site, and the aggregators have no mechanism to verify whether these reports are authentic. Meanwhile, the advertiser can only see the final histogram and not the individual reports, so it cannot determine which ones are valid. Crucially, the advertiser will use this histogram to understand the interests of users from different groups and determine its bidding strategy. The incorrect histogram with fake reports can lead to the advertiser incorrectly assigning value to a given group of users.

C.2 Single-server histogram aggregation

Another possibility is to use additive homomorphic encryption to do histogram aggregation under a single server setting. The challenge is to validate that the inputs from clients are well-formed. Indeed, *Ibex*'s approach is basically an instance of this idea, with the caveat that we use the second server to help ensure that inputs are well-formed rather than relying on expensive cryptography. Below we discuss how other works do it.

Aggregation using ElGamal encryption. AdScale [65] is a single-server aggregation protocol that uses the additive ElGamal encryption scheme [59]. Each user encrypts its report using ElGamal encryption and generates a proof that the ciphertext is the encryption of a well-formed report. The aggregator validates the proof and sums the ciphertext together. However, in their construction, decrypting the aggregation result requires computing a discrete log. Therefore, they only support summations over a maximum of 2^{16} ciphertexts. Given that daily traffic on one site can be tens of million [20], the costs can large (aggregating every 2^{16} reports requires one decryption). This solution is therefore not scalable enough to handle our particular use case. Instead, *Ibex* can increase the plaintext modulus used in the HE scheme to accommodate more encrypted reports to be aggregated. This also increases costs, but much more modestly than AdScale's approach.

Aggregation using RLWE-based homomorphic encryption. State-of-the-art HE libraries [30, 31, 36, 68] allow the computation of additions and multiplications over encrypted data without access to plaintext values. One can build single-server histogram aggregation based on these HE schemes. However, doing so requires the browser to generate a zero-knowledge proof that states that the ciphertext encrypts a well-formed one-hot vector. The size of this statement would be *quadratic* in the polynomial degree (i.e., the group size) as the encryption phase consists of multiplication of polynomials whose complexity is *quadratic* of polynomial degree. Instead, *Ibex* uses another honest but curious server without asking browsers to generate any proof.

D EXTENSION: AGGREGATE GLOBAL USER INTERESTS

Section 6.1 discusses how to apply the private histogram aggregation protocol to compute the histogram of activities of users from different groups on an advertiser's site. In some cases, advertisers selling the same type of products may want to obtain a global histogram that shows how many times each group of users visits a certain type of content on all their sites. In the meanwhile, they do not want to share the local histogram with others. For example, Adidas, Nike, and Puma all sell sportswear, and they want to combine their local histograms of user visits to their sites to generate a global histogram of user visits.

To do so, the advertisers and the ad platform can run a *secure aggregation* protocol as a blackbox. Such protocols allow a server to obtain the sum of inputs from multiple clients without learning any individual client's input. Here, the advertisers run as clients and the ad platform runs as the server; the inputs are advertisers' histograms. Since the number of groups is large (e.g., 2^{16}), we can use the secure aggregation protocols tailored for large input size [45, 46]. In the end, the ad platform sends the aggregated global histogram to the advertisers.