Blinder – MPC Based Scalable and Robust Anonymous Committed Broadcast

Ittai Abraham[1], Benny Pinkas[1,2], and Avishay Yanai*[1]

[1]VMware Research, Israel
[2]Bar-Ilan University, Israel

February 29, 2020

Abstract

Anonymous Committed Broadcast is a functionality that extends DC-nets and allows a set of clients to privately commit a message to set of servers, which can then simultaneously open all committed messages in a random ordering. Anonymity holds since no one can learn the ordering or the content of the client's committed message.

We present Blinder, the first system that provides a scalable and fully robust solution for anonymous committed broadcast. Blinder maintains both properties of *security* (anonymity) and *robustness* (aka. 'guaranteed output delivery' or 'availability') in the face of a global active (malicious) adversary. Moreover, Blinder is *censorship resistant*, meaning that a honest client cannot be blocked from participating.

Blinder obtains its security and scalability by carefully combining classical and state-of-the-art techniques from the fields of anonymous communication and secure multiparty computation (MPC). In order to demonstrate scalability, we evaluate Blinder with up to 1 million clients, up to 100 servers and a message size of up to 10 kilobytes. In addition, we show that it is a perfect fit to be implemented on a GPU. A GPU based implementation of Blinder with 5 servers, which accepts 1 million clients, incurs a latency of less than 8 minutes; faster by a factor of > 100 than the 3-servers Riposte protocol (SOSP '15), which is not robust and not censorship resistant; we get an even larger factor when comparing to AsynchroMix and PowerMix (CCS '19), which are the only constructions that guarantee fairness (or robustness in the online phase).

1 Introduction

In the 80s Chaum [Cha88] introduced a breakthrough protocol that enables a set of parties to communicate in an anonymous manner. Chaum presented it as the *Dining*

*Part of the work was done while in Bar-Ilan University.

Cryptographers problem and subsequent solutions are then called DC-networks (or a DC-net in short).

Cast in modern terms, a DC-net protocol is an instance of a secure multi-party computation protocol. The ideal functionality of a DC-net protocol is to collect inputs from all parties such that all inputs are empty messages except one, which contains a meaningful value. The functionality then broadcasts that message (with the meaningful value). Anonymity holds since the identity of the sender of the meaningful value is unknown and can potentially be any one of the participating parties. The set of participating parties is often called an *anonymity set*. Obviously, the larger the anonymity set the stronger the anonymity of the participating clients.

Systems like Dissent [WCFJ12], Verdict [GWF13] and Riposte [CBM15] have extended the DC-nets architecture in several important ways:

1. To increase scalability, they adopt the *client-server* paradigm, where there are N clients that form the anonymity set and submit messages, but only $n \ll N$ servers implement the functionality. This is a major step since a direct DC-net (with a full graph between clients) incurs an overall communication complexity of $O(N^2)$ whereas servers-aided solution reduces this to roughly $O(n \cdot N)$.

2. Instead of dealing with a single meaningful message at each round, they allow multiple clients to send meaningful messages, such that at the end of a round the servers output those messages in a random unknown order. Anonymity holds since the mapping between clients and messages is kept secret.

3. Those works devise mechanisms to detect and mask malicious clients that try to disrupt the system (e.g. DoS attacks).

4. Newer solutions (like Riposte [CBM15] and Express [ECZB19]) use techniques from function secret sharing (FSS) [BGI16] to reduce client to server and server to server communication down to $O(\log N)$ and $O(1)$ resp. per client, which enables a lighter client side and a faster verification of a client's message.

The protocol we present in this work implements an ideal functionality (Definition 2.1) for anonymous committed broadcast (ACB) and guarantees security via the simulation paradigm. Let us highlight important properties that the functionality captures: We want the protocol to be *correct*, meaning that the messages submitted by the client will eventually be broadcast; we want it to be *secure*, which in our context means that the system preserves the anonymity of the clients, so the mapping between the clients and the messages output by the protocol must be kept secret; and finally, we want it to be *robust*[1], meaning that the protocol makes progress thwarting any disruption attempt of the adversary. This includes an adversary that corrupts clients, servers, network or any combination of these. We give more details on the power of the adversary in Definition 2.1.

Another important property is *censorship resistance*, i.e. a corrupted server must not be able to block honest clients from submitting their message. Of course, if

[1] A property also known as 'guaranteed output delivery' or 'availability' and hereafter referred to by 'robustness'.

the network adversary is so powerful that it can spawn as many clients as it wishes and block any honest client that it wishes, then the effective anonymity set of *any solution* can be shrunk to only single client – by having the adversary spawn $N-1$ malicious clients and block all honest clients except for one targeted honest client. This attack completely de-anonymizes that single targeted honest client. That attack is well recongnized in the setting of such a strong adversary [SDS02]. To allow a non-trivial anonymity set (circumvent this attack), in this work we assume a slightly weaker adversary. We assume there are at least $\rho N, \rho \in (0,1)$, honest clients submitting their message, granting the network adversary the power to spawn 'only' $(1-\rho)N$ clients. The adversary can inspect, but cannot block, all network channels (a global adversary). We stress that even in this weaker model, it is necessary to prevent a corrupted server from blocking honest clients, otherwise, the same deanonymization attack is possible (i.e. even though the adversary could not block honest clients from the network, it can still do so via a corrupted server). Indeed, protocols like [CBM15, ECZB19] suffer from that type of an attack. On the other hand, Blinder has a mechanism for preventing this, which allows preserving an anonymity set of size ρN.

Limitations of previous work. Previous work on anonymous committed broadcast in the client to server DC-net model suffer from at least one of the following limitations:

1. *Non Robust*: the protocol is resilient to clients' disruption attempt, but (even a single) corrupted server may halt the system. A corrupted server may choose to halt the execution *adaptively*, namely, even at the very last step of the protocol, after observing the output messages (in case it wishes some message will not go public).
2. *Suspectable to Censorship*: a corrupted server could arbitrarily block messages from honest clients, dropping the effective anonymity set size to only one. This holds even in the weaker model mentioned above (in which the adversary may corrupt at most $(1-\rho)$ clients).
3. *Non Scalable*: the protocols can only be run by a few servers or can accept a relatively small number of clients. Removing these limitation is important for security as increasing the number of servers typically increases trust in the system and accepting more clients fortifies their anonymity.

The aforementioned works, Dissent [WCFJ12], Verdict [GWF13] and Riposte [CBM15] all suffer from the first two limitations, that is, even a single server may halt their execution and may arbitrarily block honest clients from submitting their messages (leading to an anonymity set of size only one). With regard to generic MPC based works, in MxMix [AKTZ17] the n servers run the shuffling protocol of Hamada et al. [HKI+12] that is only secure against a semi-honest adversary and works with $n=3$ servers only, On the other hand AsynchroMix and PowerMix [LYK+19] suggest two generic ways for shuffling the N messages submitted to the servers via secret sharing. Those protocols achieve *fairness* (or 'robustness in the online phase') and can run over may servers, however, having the servers running a generic shuffling protocol requires either $O(N^3)$ computation overhead or $O(\log^2 N)$ communication rounds, which considerably limit their ability of handling more than few thousands of clients.

1.1 Our Contributions.

We present Blinder, the first system that provides a scalable and robust solution for anonymous committed broadcast. Blinder maintains *security* (anonymity) , *robustness* and *censorship resistance* in the face of a global malicious adversary, over a synchronous network (all messages are delivered within some bounded time). In more detail,

1. *Robustness*: it keeps operating correctly and securely even in the presence of an adversary who controls the entire network, controls $t < n/4$ of the servers and may spawns $(1-\rho)N$ malicious clients.

   To achieve a robust construction, we make sure that all the building blocks of the protocol are robust, however, we find that this is a necessary but not sufficient condition. Interestingly, having each client sharing its message using a Shamir sharing does not lead to a robust protocol, **even when all servers behave honestly**. We discuss that observation and other robust sub-protocols.

2. *Censorship resistance*: if there are ρN honest clients then the effective anonymity set is of size ρN, in contrast to previous work in which anonymity set size drops to one.

3. *Scalability*: Blinder can be deployed with any number of servers and can support an anonymity set size in the millions, with a relatively low latency, outperforming even systems with weaker security guarantees.

   We observe that the arithmetic circuit to be securely evaluated by the servers can significantly benefit from a GPU deployment, which makes the system practical even for anonymity sets of millions of clients. This makes our protocol unique in the field of anonymous communication, in which protocols almost always rely on either symmetric or asymmetric cryptographic primitives that could only marginally benefit from a GPU (see discussion in Appendix A). We implemented Blinder on both CPU and GPU and extensively evaluated it over varying number of servers $n \in [100]$ varying number of clients $N \in [2 \cdot 10^6]$ and varying message size $L \in \{32 \text{ bytes}, \ldots, 10 \text{ kilobytes}\}$ (32 bytes messages is the only case reported by AsynchroMix and PowerMix, which may be useful for e.g., publishing secret keys. 160 byte messages correspond to anonymous micro-blogging application, e.g. anonymous Twitter. Larger message size may be applicable to anonymously transacting cryptocurrencies as well as whistleblowing small files). Let us present some examples of Blinder's performance:

   - The CPU variant with $\boldsymbol{n = 100}$ **servers** could handle and process $N = 10,000$ clients in 11 seconds. In comparison, AsynchroMix and PowerMix [LYK+19] was evaluated with up to 4096 clients (which took more than 2 minutes). Blinder is $20\times$ faster than [LYK+19] on a comparable setting. Of course, Blinder can handle many more clients with a reasonable latency. E.g., it takes about 7 minutes to serve $100,000$ clients.
   - Blinder's GPU variant can serve 1 million clients with 160 byte messages in less than 8 minutes. This is more than $100\times$ faster than the non robust

system, Riposte, under similar setting. Also, this is the first robust system that can scale to such number of clients.

The source code of Blinder can be found in github.com/cryptobiu/MPCAnonymousBloging

1.2 Overview of our techniques

Blinder is heavily based on Shamir's threshold secret sharing [Sha79] and on advances in secure computation [DN07, BH08, CGH+18] on shared secrets.

There already exist secret sharing based constructions for anonymous communication [AKTZ17, LYK+19] in which the clients first share their message toward the servers, which then run a *secure shuffle* protocol of the messages, such that the mapping between initial and final locations of each message remains secret. These solutions do not scale well since secure shuffling incurs either a large computation overhead (as in PowerMix) or a large number communication rounds (as in McMix and AsynchroMix). Alternatively, Blinder adapts a different approach inspired by Riposte [CBM15], which in turn is based on distributed point functions[2] (DPF) [GI14, BGI16]. In that approach we offload some of the computation and communication burden from the servers to the clients, by which we completely remove the need for a secure shuffle protocol.

Using a DPF-like technique, the servers maintain a (initially empty) table and the clients themselves randomly and *obliviously* choose the final location of their message in that table. Namely, the servers do not know the chosen location. This requires a client to essentially write to *every location* in the table. So clients write their 'real' message to the chosen location and an empty message (or zero) to the rest of the table. As 'writing' is done by secret sharing, the servers could not tell which location contains the message and which contains zero. The homomorphism properties of Shamir's sharing allow the servers to aggregate submissions of many clients, resulting a single table with many messages, one from each client. This however, requires each client to deal $O(N)$ sharings, which rapidly becomes the bottleneck. To solve this problem we extend a DPF-based techniques to work efficiently with Shamir's sharing.

Informally, in a DPF scheme [GI14] there are two servers $\mathcal{S}_1, \mathcal{S}_2$ that maintain a table T, of size $O(N)$, additively shared between them; so $\mathcal{S}_1$ (resp $\mathcal{S}_2$) has T_1 (resp T_2) such that $T = T_1 + T_2$ (entry-wise). To read a single entry at location i from T, a client interacts with both servers and submits a query q_1 to $\mathcal{S}_1$ and query q_2 to $\mathcal{S}_2$. The servers locally process the queries and respond with x_1 and x_2 to the client, that combines these answers to obtain $x = x_1 + x_2 = T[i]$. A DPF scheme is oblivious, i.e. the servers do not learn anything about the index i. While the original DPF scheme allows *reading* from T, the construction in [CBM15, ECZB19] allows also *writing*, a necessary condition for anonymous communication. On the other hand, those systems in [CBM15, ECZB19] inherit the limit of DPF constructions and can work efficiently with two servers only.

The main benefit of relying on DPF is that the size of the queries q_1 and q_2 can be made sub-linear in T, even though when processing the queries the servers have to 'touch' every entry of T. Specifically, it is possible to compress the query up to

[2] A distributed point function (DPF) is related to private information storage [OS97] and 2-server private information retrieval (PIR) [CG97].

size $O(\log T)$. Applications like anonymous communication set the table size, T, be proportional to the number of clients, N and prefer that to be as large as possible (to increase the anonymity set size). Thus, compressing the query size has a huge impact on the communication complexity.

In Blinder, we further extend the notion and construction of DPF (1) to support *many servers*, (2) to obtain *robustness* while preserving (3) the *low communication* complexity between servers.

The first two improvement achieved by using Shamir sharing instead of additive sharing. This modification allows using very efficient techniques for securely computing on shared secrets, which is required in order to decompress a client's write query, to verify that a query is well formed and to aggregate queries of many clients to a single final table.

The third improvement is achieved by integrating the MPC protocol by Chida et al.: A basic sub-protocol in secure computation is a multiplication of shared secrets. Specifically, given sharings of secrets x and y from a finite field $\mathbb{F}$, the servers can obtain a sharing of $x \cdot y$ without revealing x or y. This requires a communication of $O(|\mathbb{F}|)$ bits per server. In our protocol, decompression of a query incurs $O(N)$ multiplication of secrets, leading to a protocol with communication overhead of $O(N^2|\mathbb{F}|)$ bits, which is impractical when N is large. Fortunately, a recent observation [CGH+18] suggests a new type of arithmetic gate, namely, the *sum of products* gate. That is, given sharings of secrets $x_1, \ldots, x_\ell$ and $y_1, \ldots, y_\ell$ from $\mathbb{F}$, the servers can obtain a sharing of $\sum_{i=1}^{\ell} x_i \cdot y_i$ without revealing any of the intermediate values. Surprisingly, this incurs the same communication overhead as if they perform *only one secure multiplication*, that is, only $O(|\mathbb{F}|)$ bits per server. This drastically reduces server to server communication overhead and makes our construction highly competitive.

Furthermore, as we will show, the entire computation can be represented as an instance of matrix multiplication, which is a perfect task for a GPU.

1.3 Applications

Apart from an application to fearless whistle-blowing of fraud or incompetence to the public [WCFJ12, Bal04, Kre01, TFKL99, Vol99, Wal19], a system for anonymous committed broadcast has other interesting applications:

Differential privacy. Recent results in differential privacy analyzed the *shuffle model*, where users anonymously submit their private inputs to a server. This setup is a trust model which sits between the classical curator model (where a server knows the inputs of all clients and adds noise to answers to queries) and the local privacy model (where each client adds noise to its input). It was shown that the shuffle model provides substantially better tradeoffs between privacy and accuracy when aiming to achieve differential privacy [BEM+17, EFM+19, CSU+19, BBGN19]. However, current works analyze the advantages of this model without describing how it can be implemented. Our work essentially implements the anonymous channel that is assumed by the shuffle model. To support that, Blinder has to serve thousands to million clients with relatively short messages (e.g. telemetry) (representing their personal noisy data) of tens to hundreds of bytes.

P2P payments and front-running prevention. While Bitcoin's transactions are fully public, there exist peer-to-peer payment systems, like Zcash [BCG+14] and Monero [vS13], that preserve the privacy of their users (allowing payer, payee and amount privacy). However, in those systems privacy is only preserved in the application layer but not in the network layer. That is, having an access to the ledgers reveals nothing about 'shielded' transactions (except from the fact that they happened). However, being able to inspect the network could lead to their deanonymization by simply tracking the IP address of the transaction's sender. Currently, users of those systems are instructed to use Tor [Fou19] in order to hide their identity over the network layer, which is known to be insufficient [HVC10, MD05, MZ07, Ray00]. Instead, Zcash/Monero users could potentially use Blinder to broadcast their transactions and be fully protected also over the network layer. To be useful for systems like Zcash and Monero, Blinder has to support relatively large messages of length $> 1KB$. A standard shielded transaction in Zcash is of length $1 - 2KB$ [BCG+14] and the system currently processes 6 shielded transactions per second [Bit20].

A related problem is front-running in decentralized exchanges (DEX). DEXs typically execute trades over smart contracts, avoiding a trusted party that might steal funds. Using smart contracts means that the entire order book is pubic and transparent, which implies a fundamental weakness of *front-running*, or "dependency of trade orders" [KMS+16, LCO+16, DGK+19], referring to the practice of entering into a trade to capitalize on advanced knowledge of pending transactions. Users can *front-run* orders by observing past and most recent orders and placing their own orders with higher fees to ensure they are mined first. Blinder can circumvent that problem as all messages are *committed* and opened only after all trade orders are submitted, removing the aforementioned dependency. We remark that current DEX constructions are not anonymized, hence, using Blinder only for simultaneous opening of trade orders would be an overkill. Yet, research is striving for anonymized solution for DEX, for which Blinder is a perfect fit to serve as the communication medium.

1.4 Related Work

Since Chaum's breakthrough in the '80s there was a huge body of work on anonymous communication of all forms (dialing, messaging and broadcasting) The earliest proposals were *mix networks* [BFK00, BL, CJK+16, Cha81, DDM, GT96, KEB98, BCC+15, BCZ+13] with the leading ones being Loopix [PHE+17] and Multiparty Routing [SAKD17], and *dining cryptographers (DC) network* [Cha88, WCFJ12, SGRE04]. Mix networks rely on a set of servers (called mixes) to shuffle messages before delivering them to recipients. This shuffling is often accompanied by encryption, batching, and chaffing (the addition of dummy traffic) to prevent traffic analysis. Since all operations are relatively lightweight, these systems enjoy lower latency and higher throughput than many other works in the literature – including our work. However, such systems are susceptible to replay, duplicate or drop message attacks by malicious mixes [KAPR06, LRWW04, MD04, NS03, Pfi94, PP89, Ray00, SW06, Wik03].

On the other hand DC networks provide stronger security guarantees, with the price of being peer-to-peer (full graph of communication between clients) and are based on all-to-all broadcast of messages, which results in high costs. Consequently,

these systems typically accommodate only dozens of users. Verdict [GWF13], Dissent [CF10, WCFJ12], Riposte [CBM15] and PowerMix [LYK+19] make great strides to reduce these costs and support thousands of users with a reasonable latency, in the price of putting a limited trust on the infrastructure, e.g. assuming some of the servers are honest. Yet, scaling those systems to support large anonymity-sets (tens of thousands to millions) incurs a huge, impractical latency. For instance, supporting a anonymity-set of a million clients with 160B messages in Riposte [CBM15] incurs a latency of more than 11 hours; in Atom [KCDF17] this can take about half an hour, but since they aim for horizontal scalability (similar to a mix net, in which it is not necessary for every server to 'touch' every message) they require at least 1024 servers to support that (thus, we consider Atom's setting to be different than our interest in this paper).

Onion routing [DMS04, MOT+11, MWB13, RSG98], and Tor [DMS04] in particular, is widely adopted due to their relative low latency and ability to support millions of users. However, these systems are unable to resist traffic analysis attacks [HVC10, MD05, MZ07, Ray00], even those performed by local adversaries [CZJJ12, KAL+15, PNZE11, WG13]. A very recent work, Loopix [PHE+17], aims at enhancing security guarantees of a mix networks by introducing ideas like Poisson mixers and loop messages, which allow it to resist a global passive adversary. However, there are several aspects by which Loopix differs from Blinder, which makes it insufficient for securely compute the ACB functionality (Definition 2.1): First, Loopix is designed for *messaging* whereas in this work we aim to solve *committed broadcast* form of communication. We could think of ways to modify Loopix to support broadcast, for example, by assigning egress providers to collect messages and broadcast them at the end of a phase. However, this would significantly degrade security as now an adversary that controls a row or a column of mixers could break anonymity. In their paper, they report an implementation setup with up to 500 clients, which might indicate a scalability issue as their memory consumption is hight (possibly for preventing replay attacks), in this work, however, we aim to reach millions of clients. In terms of security, Loopix guarantees traffic analysis resistance by a set of properties (like sender/receiver anonymity) and assumes some of the nodes (the egress providers) remain semi-honest and non-colluding with malicious mix nodes. On the other hand, Blinder follows a stronger simulation based security, that does not assume a specific adversarial strategy (or a specific property to protect), rather, it guarantees protection against any kind adversarial behaviour. In addition, in Blinder no node (server) is assumed to remain semi-honest or to not collude with other corrupted nodes/clients.

Finally, there are *mailbox*-based systems [BDG15, CB95, KOR+04, KLDF16, SCM05, vdHLZZ15, LZ16, AS16] for anonymous 'dialing' and messaging, in which clients retrieve messages from a 'mailbox' (or a 'dead drop') kept secret by a third party servers. Those systems are mainly designed for peer-to-peer communication rather than broadcasting.

In the rest of this section we present a detailed comparison to client-server based DC-net constructions, as this is the model that Blinder follows. Moreover, to match our simulation based security definition we focus only on simulatable constructions. This is to exclude ones that rely on other notions of privacy like differential privacy (DP) in which some controlled, but not trivial, amount of information is leaked to the adversary.

The leading works on anonymous broadcast in the client-server DC-net model are Riposte[3] [CBM15], AsynchroMix, PowerMix [LYK+19] and McMix [AKTZ17]. While McMix, AsynchroMix and PowerMix all implement a secure shuffling, Riposte takes another approach (that Blinder follows as well), in which the task of hiding the mapping between clients and messages is offloaded to the clients via an extension to the DPF construction.

Riposte [CBM15] demonstrates a system with a high throughput and which scales to millions of clients. It is run by two database servers (that process the clients' DPF queries) and an additional audit server (that verifies that messages are well-formed), assuming no collusion between them (i.e. the adversary may corrupt only a single server). Riposte also presents a multi-server construction. However, that construction is very inefficient and not fully implemented, therefore, in our experiments we run only Riposte's 3-servers version whereas analytical comparison to the multi-server version appears in Table 1.

Building on the DPF construction of [GI14, BGI16] allows Riposte to heavily rely on modern processors' intrinsics for computing AES. However, the they do not scale efficiently to more than 2 database servers, are not robust and not censorship resistant, i.e. a malicious server not only can halt the execution whenever it wishes, it could also arbitrarily block messages of honest clients. Even in Riposte's multi server construction, a single malicious server is sufficient to mount those attacks. In contrast, Blinder runs efficiently with any number of servers, and is both robust and censorship resistant.

McMix [AKTZ17] and AsynchroMix/PowerMix [LYK+19] rely on generic MPC protocols assuming an honest majority. McMix implements the secure shuffling by Hamada et al. [HKI+12], which works efficiently only for 3 *semi-honest* (passive) servers. That protocol, as well, could not efficiently extend to more than 3 servers.

Asynchromix and PowerMix scale beyond the 3 servers limit, are actively secure and assume that less than a third of the servers collude, an assumption that enables them to achieve both censorship resistance and *fairness* (or robustness in the online phase, see Section 2). The aim of Blinder is to achieve *full-robustness*, that is, also in the offline phase. We remark that, by aligning their collusion assumption to that of Blinder (i.e. $t < n/4$ instead of $t < n/3$), those protocols could achieve full-robustness as well. The drawback of Asynchromix and PowerMix is that they do not scale to more than a few thousands of clients. This is due to the fact that AsynchroMix has a high round complexity of $O(\log^2 N)$ and PowerMix runs a $O(N^3)$ computation for solving power equations. In contrast, Blinder scales to millions of clients.

We refer to another recent concurrent and independent work, Talek [CSM+20], since it is also utilizes a GPU. Talek aims to specifically solve group messaging via information theoretic PIR and explicitly redirects to solutions tailored for broadcast if that is the required application. Talek may scale in both number of clients and servers, however it has several limitations: it is not robust and not censorship resistant; any user in a trusted group can block writes to that groups log. Similarly, a faulty server can impact availability.

[3] A concurrent and independent work, Express [ECZB19], improves Riposte both in performance and in its trust assumption.

We present a qualitative comparison in Table 1 (we recommend to read notations in Section 2 first).

work	client comp.*	server comp.*	client-server comm.*	server-server comm.*	server-server rounds	collusion; adversary type	cryptographic assumptions	fair	robust	censorship resist.	N_h
McMix [AKTZI7] 3 servers	$O(1)$	$O(N \log N)$	$O(1)$	$O(N \log N)$	$O(\log N)$	$n=3;\ t=1$ semi-honest**	information theoretic	–	–	✓	ρN
Riposte [CBM15] 2 servers + audit	$O(\sqrt{N})$	$O(N^2)$	$O(\sqrt{N})$	$O(N \cdot \sqrt{N})$	$O(1)$	$n=3;\ t=1$ malicious	computational (OWF)	✗	✗	✗	1
Riposte [CBM15] n servers	$O(n \cdot \sqrt{N})$	$O(N^2)$	$O(\sqrt{N})$	$O(N)$	$O(1)$	$t<n$ malicious	computational (DDH)	✗	✗	✗	1
AsynchroMix [LYK+19] switching network	$O(n)$	$O(n \cdot N \cdot \log^2 N)$	$O(1)$	$O(n \cdot N \cdot \log^2 N)$	$O(\log^2 N)$	$t<n/3$ malicious	information theoretic	✓	✗	✓	ρN
PowerMix [LYK+19]	$O(n)$	$O(N^3)$	$O(1)$	$O(N^2)$	$O(1)$	$t<n/3$ malicious	information theoretic	✓	✗	✓	ρN
Blinder this work	$O(n \cdot \sqrt{N})$	$O(N^2)$	$O(\sqrt{N})$	$O(N)$	$O(1)$	$t<n/4$ malicious	information theoretic	✓	✓	✓	ρN

Table 1: Qualitative comparison of Blinder to leading client-server DC-net constructions with simulation based definition. OWF refers to the existence of one-way functions and DDH refers to the existence of a group under which the decisional Diffie-Helman problem is hard. 'client-server' refer to the message size between a client to a *single* server, the overall communication should be multiplied by n; *All values in this column are multiplied by L (the message length). ** This relates to their implementation. Theoretically McMix stands against a malicious adversary as well.

1.5 Paper Organization

Notation and definitions are presented in Section 2. We present a basic (impractical) construction of Blinder in Section 3 and an optimized, highly efficient version in Section 4. These versions rely on the fact that the preprocessing of Blinder is robust; to this end we show how to obtain a robust preprocessing in Section 5.1. We present an evaluation and comparison of Blinder in Section 6 and discuss our conclusion in Section 7.

2 Notation and Problem Definition

Denote by $[x]$ the set $\{1,\ldots,x\}$. Indexes begin at 1. λ is the statistical security parameter. A function $\mu:\mathbb{N}\to\mathbb{R}$ is negligible if for every positive polynomial $\mathsf{poly}(\cdot)$ there exists an integer N_{poly} s.t. for every $x > N_{\mathsf{poly}}$ it holds that $|\mu(x)| < 1/\mathsf{poly}(x)$. We refer to a *negligible probability* as $\mu(x)$ for some security parameter x, in addition, an *overwhelming probability* is $1-\mu(x)$.

Blinder runs on a set of n servers $\mathcal{S}_1,\ldots,\mathcal{S}_n$, of which $t < n/4$ may be actively (maliciously) corrupted and colluding. The servers emulate a trusted party that runs the anonymous committed broadcast (ACB) functionality defined below.

Definition 2.1 (ACB Functionality) *The functionality:*

1. *interacts with N clients $\mathcal{C}^1, \ldots, \mathcal{C}^N$ of which ρN are honest and $(1-\rho)N$ are controlled by the adversary.*
2. *interacts with n servers, $\mathcal{S}_1, \ldots, \mathcal{S}_n$, of which $t < n/4$ are controlled by the adversary.*
3. *maintains a table, A, of $c_1 \cdot N$ entries for a constant $c_1 > 1$.*
4. *waits for a message m^k from each client $\mathcal{C}^k$ and appends m^k to a uniformly random entry in A. All messages are of a fixed length of L bytes.*
5. *after receiving a message from all clients, the functionality outputs to all servers and the adversary the messages of every entry with at most c_2 messages, for a constant $c_2 \geq 1$. If an entry has more than one message, they are output in a random order.*

Note that in the ACB functionality there is no interaction between the servers or between the clients. The servers simply obtain a permuted list of messages that were submitted by the clients. However, in an implementation of that functionality the servers may interact with each other and with the clients. In that respect, we assume a global adversary (who may inspect all network channels) and assume they are secure, which could be achieved in practice by setup of a public key infrastructure (PKI).

Let us describe some interesting properties that are reflected by a protocol that securely computes the ACB functionality. First, it has properties of a standard commitment scheme:

- *Hiding*: before the output step (5), the adversary has no knowledge on the messages of the honest clients.
- *Binding*: a client cannot modify its message after submission in step (4).

Then, there is the anonymity set property:

- *Anonymity set*: the adversary learns nothing about the mapping between the ρN honest clients and the messages of honest clients that are output by the functionality.

Obviously, for a given ρ, the larger the number of clients, N, supported by the functionality, the better anonymity provided to them. Observe that in the ACB functionality the adversary cannot block a message from an honest client, therefore, the *effective anonymity set size*, denoted N_h, equals the number of honest clients, ρN. Finally, we are interested in the robustness property:

- *Robustness*: also known as 'availability' or 'guaranteed output delivery', means that no matter what the adversary does, it cannot make the system halt without producing output *to all servers*. This is stronger than the *fairness* property, achieved by [LYK+19], in which the system produces outputs to the adversary only if it also produces outputs to the honest participants.
- Robustness also covers the case of a malicious client. Obviously, by the definition, it is impossible for a client to break correctness or security of the functionality.

Although we list some interesting properties for such a system, we remark that the security of Blinder follows a simulation based definition [Gol04]. This is straightforward for Blinder as it uses only standard simulatable building blocks in a black-box manner.

For readability reasons, the building blocks used by Blinder are presented in the context of each section.

We stress that the security guarantees claimed by Blinder are limited to a stand-alone execution, which we call an 'epoch'. For a continuous execution of Blinder across many epochs, a standard heuristic is to have every client ever interacted with the system submitting a 'cover' (empty) blinded message on every epoch. We leave it out of the scope of the paper.

In the next section we present an implementation of the ACB functionality with $c_1 = 2.7$ and $c_2 = 2$ (constants inherited from [CBM15])).

3 The Basic Blinder

As a starting point, we present a basic construction, which includes all aspects of the system. In particular, this basic construction is already resilient to malicious clients/servers.

Blinder heavily relies on Shamir Secret Sharing (SSS) [Sha79] and on state-of-the-art MPC components on top of it. We detail SSS here and each required component before using it.

3.1 Shamir Secret Sharing (SSS)

Let us fix a finite field $\mathbb{F}$ over which subsequent computations will be done. Concretely, think of a prime field $\mathbb{F}_p$ with a prime $p > n$ such that server $\mathcal{S}_q$, for $q \in [n]$, is identified with integer $q \in \mathbb{F}_p$.

By a d-polynomial we mean a polynomial $f(X) \in \mathbb{F}[X]$ of degree *at most* d. To share a secret $x \in \mathbb{F}$ with degree d, a uniformly random d-polynomial $f(X) \in \mathbb{F}[X]$ with $f(0) = x$ is chosen, and $\mathcal{S}_q$ is given the share $x_q = f(q)$. We denote such a procedure by $\mathsf{Share}_d(x)$. It is well known that such a d-polynomial information theoretically hides x from any subset of at most d share holders.

In the following we call a vector $\vec{x} = (x_1, \ldots, x_n)$ a consistent d-sharing of s if there exist coefficients $a_1, \ldots, a_d$ such that the polynomial $f(X) = s + \sum_{i=1}^{d} a_i \cdot X^i$ evaluates $f(q) = x_q$ for all honest servers $\mathcal{S}_q$. For $q \in [n]$, we say that the polynomial $f(X)$ 'agrees with x_q' if $f(q) = x_q$. To reconstruct a secret x that is shared by at least $d + 1$ honest servers $\mathcal{S}_{q_1}, \ldots, \mathcal{S}_{q_{d+1}}$, the servers run the procedure $\mathsf{Reconstruct}(\{q_i, x_{q_i}\}_{i \in [d+1]})$, in which they broadcast their shares $x_{q_1}, \ldots, x_{q_{d+1}}$, which form the $d+1$ points $(q_1, x_{q_1}), \ldots, (q_{d+1}, x_{q_{d+1}})$. Then each server interpolates the d-polynomial $f(X)$ with $f(q_i) = x_{q_i}$ for all $i \in [d + 1]$ and outputs the secret $x = f(0)$.

In the rest of the paper we use $[x]$ to denote a consistent d-sharing of x if $d = t$ (i.e. when d equals the maximum number of corrupted servers t) and we use $\langle x \rangle$ if $d = 2t$.

3.1.1 Robust reconstruction.

The description of Reconstruct() above assumes that only honest parties participate and so a set of $d+1$ points can reconstruct a single d-polynomial, hence, a single possible secret to output. However, in practice, we do not know which servers are honest and which are not, thus, Reconstruct has to take into account the shares of all servers, even those of corrupted servers. Given n shares with at most t of them being malformed, Reconstruct should output a single d-polynomial that 'agrees' with $n-t=3t+1$ of them. Of course, it is possible to iterate over all subsets of $3t+1$ servers, interpolate a polynomial from their shares (which define points) and check whether the result is a d-polynomial. However, it would be too costly, as there are $\binom{n}{3t+1}$ such subsets. Fortunately, when $n \geq 4t+1$ it is possible to construct a unique polynomial of degree d, as long as $d \leq 2t$. Specifically, given $4t+1$ points, of which at most t are malformed, the Berlekamp-Welch algorithm outputs a d-polynomial passing through at least $3t+1$ of the points, in $\tilde{O}(n^3)$ time (based on Gaussian elimination)[4]. We re-define Reconstruct$(\{x_q\}_{q\in[n]})$ to take shares from *all* servers and find a unique polynomial that agrees with at least $3t+1$ servers.

In Blinder, there might be cases in which Reconstruct returns a polynomial that disagrees with some servers, then, we have to decide what to do. As we show below, in some cases we conclude that those servers cheated and so they are ignored in further computation while in other cases such a conclusion would be a mistake and so other action is taken.

3.1.2 Computation on sharings.

We introduce a shorthand to specifying computations on shares. By $[x^{(1)}],\ldots,[x^{(\ell)}]$ we mean that server $\mathcal{S}_q$ holds shares $x_q^{(1)},\ldots,x_q^{(\ell)}$. Consider any function $f:\mathbb{F}^\ell \to \mathbb{F}^m$. By $([y^{(1)}],\ldots,[y^{(m)}]) \leftarrow f([x^{(1)}],\ldots,[x^{(\ell)}])$ we mean that each server $\mathcal{S}_q$ locally computes $(y_q^{(1)},\ldots,y_q^{(m)}) \leftarrow f(x_q^{(1)},\ldots,x_q^{(\ell)})$, thus, for all $k \in [m]$ that defines shares $[y^{(k)}] = (y_1^{(k)},\ldots,y_n^{(k)})$. It is well known that if f is an affine function and the $[x^{(l)}]$ for $l \in [\ell]$ are consistent d-sharings, then $[y^{(k)}]$ for $k \in [m]$ are consistent d-sharings of $(y^{(1)},\ldots,y^{(m)}) \leftarrow f(x^{(1)},\ldots,x^{(\ell)})$. Furthermore, if $[x_1]$ and $[x_2]$ are consistent d-sharings, then $[x_1]\cdot[x_2]$ is a consistent $2d$-sharing of $y = x_1 \cdot x_2$. This is due to the fact that the parties implicitly multiply two polynomials of degree t, which result in a polynomial of degree $2t$. When $d=t$ we therefore write $\langle y\rangle = [x_1][x_2]$.

3.2 Basic Construction of Blinder

We are ready to describe the basic protocol. Let N be the number of messages submitted to the system in a given epoch (i.e. the number of clients). The servers distributively maintain a matrix A with rows rows and cols columns such that rows $\times$ cols $\geq N$ (i.e. it has sufficient room for all messages). Denote the entry in the i-th row and j-th column by $A_{(i,j)}$. Specifically, for every i,j the servers maintain the sharing $[A_{(i,j)}]$, meaning that $\mathcal{S}_q$ holds $A_{(i,j),q}$. We denote by A_q the whole matrix of shares held by $\mathcal{S}_q$.

[4] $\tilde{O}$ hides a poly-logarithmic complexity. A robust reconstruction could be done using the more efficient FFT-based algorithm of Soro and Lacan [SL09] in time $\tilde{O}(n)$.

For simplicity, in the following we assume that messages consist of a single field element, i.e. $L = 1$ and later discuss how to extend to any $L > 1$.

3.2.1 Submitting a message.

In this basic scheme, the communication of each client is $O(N)$. We show later in Section 4 how to reduce the communication overhead to $O(\sqrt{N})$. To submit message $m \in \mathbb{F}$, a client $\mathcal{C}$ picks random indices $i^\star \in [\mathsf{rows}]$ and $j^\star \in [\mathsf{cols}]$ and prepares a matrix M of size $\mathsf{rows} \times \mathsf{cols}$ such that:

$$M_{(i,j)} = \begin{cases} m & \text{if } (i,j) = (i^\star, j^\star) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Then, $\mathcal{C}$ calls $\mathsf{Share}_t(M_{(i,j)})$ for every (i,j) by which server $\mathcal{S}_q$ obtains $M_{(i,j),q}$. We call M the 'blinded' version of m as any subset of at most t servers learn nothing about neither the message m nor the indices $i^\star, j^\star$ from the sharings $[M_{(i,j)}]$.

During submission time, N clients prepare blinded messages, so that client $\mathcal{C}^k$ with message m^k prepares a matrix M^k as its blinded message and shares it entry-wise toward all servers. Denote S_q's share of the (i,j)-entry by $M^k_{(i,j),q}$ and denote by M^k_q the entire matrix of shares that $\mathcal{S}_q$ received from $\mathcal{C}^k$.

3.2.2 Processing and revealing all messages.

Once all messages are submitted, the servers reveal only the *aggregation* of all of them. Given shares of blinded messages $M^1_q, \ldots, M^N_q$ from all N clients, server $\mathcal{S}_q$ computes the sum of the shares. Namely, for every (i,j)

$$A_{(i,j),q} = \sum_{k=1}^{N} M^k_{(i,j),q} \tag{2}$$

Note that for every (i,j), $A_{(i,j),q}$ is a share of the sum of values that all clients put in the (i,j) entry of their blinded message. In other words, the servers compute the linear function $[A_{(i,j)}] = \sum_{k=1}^{N} [M^k_{(i,j)}]$ for every (i,j). Indeed, following 3.1.2, the sharing of $A_{(i,j)}$ is a t sharing.

Then, the servers run $A_{(i,j)} \leftarrow \mathsf{Reconstruct}(\{A_{(i,j),q}\}_{q\in[n]})$ for every (i,j) and output matrix $A = \{A_{(i,j)}\}_{i\in\mathsf{rows}, j\in\mathsf{cols}}$, which contains all clients' messages.

3.2.3 Handling collisions.

Suppose that *only one client* $\mathcal{C}^k$ picked some entry $(i^\star, j^\star)$ for its message m^k, then it follows that $M^k_{(i^\star,j^\star)} = m^k$ and $M^{k'}_{(i^\star,j^\star)} = 0$ for every $k' \neq k$. Thus, according to Eq.(2), the value $A_{(i^\star,j^\star)}$ reconstructed by the servers equals m^k, and the message of $\mathcal{C}^k$ is broadcast successfully. On the other hand, if two or more clients, $\mathcal{C}^{k_1}, \ldots, \mathcal{C}^{k_c}$ picked $(i^\star, j^\star)$ then $A_{(i^\star,j^\star)} = m^{k_1} + \ldots + m^{k_c}$, a case denoted as a 'collision'. To deal with collisions we apply the following technique borrowed from [CBM15]. Instead of only one matrix, each client $\mathcal{C}$ prepares two matrices: M as before, and a

new matrix $\hat{M}$ such that $M_{(i^\star,j^\star)} = m$, $\hat{M}_{(i^\star,j^\star)} = m^2$ and $M_{(i,j)} = \hat{M}_{(i,j)} = 0$ for all $(i,j) \neq (i^\star,j^\star)$. The servers now distributively maintain two matrices: the matrix A that aggregates blinded messages M^k and the matrix $\hat{A}$ that aggregates blinded messages $\hat{M}^k$. Now, if there are two clients $\mathcal{C}^{k_1}, \mathcal{C}^{k_2}$ who picked the same entry $(i^\star,j^\star)$ then we have $A_{(i^\star,j^\star)} = m^{k_1} + m^{k_2}$ and $\hat{A}_{(i^\star,j^\star)} = (m^{k_1})^2 + (m^{k_2})^2$. Using those two equations it is easily possible to find m^{k_1} and m^{k_2}: In a prime field $\mathbb{F}_p$ when $p \mod 4 = 3$, for a given $b \in \mathbb{F}_p$ we can solve $b = x^2$ by computing $x_1 = b^{(p+1)/4}$ and $x_2 = -x_1$. This incurs a cost of $\log p - 2$ finite field multiplications, by computing the repeated powers of b, i.e. $b, b^2, b^4, \ldots, b^{(p+1)/4}$.

This technique works as long as at most two clients wrote to the same entry and fails for three or more. Note that the probability that 3 or more clients picked the same entry $(i^\star,j^\star)$ is sufficiently small for our application. Specifically, [CBM15] shows that when fixing $\mathsf{rows} \times \mathsf{cols} \geq 2.7N$, the probability that 3 or more clients picked $(i^\star,j^\star)$ is less than 0.05, so at least 95% of the messages will be successfully broadcast. Of course, one can increase the success rate by increasing the blinded messages size, for instance, with each client submit blinded messages for m, m^2 and m^3 and then having the servers solve a cubic equation for an entry with a collision of three messages. Success rate reaches probability 1 when the clients send N messages $m, m^2, \ldots, m^N$ and having the servers solving equations of degree N. A similar extreme approach is taken by PowerMix [LYK+19] and incurs a very expensive computation overhead of $O(N^3)$; hence, it is practical to relatively small N's.

3.2.4 Formal description

Let us describe formally the full Blinder protocol. Note that the protocol refers to some sub-protocols that are presented in later sections.

Protocol $\mathsf{BlinderProtocol}()$

The protocol runs by n servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and N clients $\mathcal{C}^1, \ldots, \mathcal{C}^N$. The servers initialize distributed matrices A and $\hat{A}$ with $c_2 \cdot N$ entries each. That is, for each (i,j) we have $A_{i,j} = \hat{A}_{i,j} = 0$ and the servers maintain the sharings $[A_{i,j}]$ and $[\hat{A}_{i,j}]$. Initialize the counter $C_q = 0$ for every server $\mathcal{S}_q$. Initialize the counter $\mathsf{AcceptedClients} = 0$

Preprocess The servers run protocol $\mathsf{Preprocess}()$ from Section 5.1. This phase output sufficiently many random double sharings of the form $([r], \langle r \rangle)$ for a uniformly random $r \in \mathbb{F}$.

Private inputs The servers have no input. Client $\mathcal{C}^k$ has input $m^k \in \mathbb{F}$.

Blind message Client $\mathcal{C}^k$ prepares two matrices M and $\hat{M}$ with $N' = \sqrt{c_1 \cdot N}$ rows and columns as follows:

1. Uniformly sample $i^\star, j^\star \in [N']$
2. Set $M^k_{i^\star,j^\star} = m$ and $M^k_{i,j} = 0$ for all $(i,j) \neq (i^\star,j^\star)$.
3. Set $\hat{M}^k_{i^\star,j^\star} = m^2$ and $\hat{M}^k_{i,j} = 0$ for all $(i,j) \neq (i^\star,j^\star)$.

Optimistic input 1. For each $(i,j) \in [N']^2$, $\mathcal{C}^k$ runs $\mathsf{Share}_t(M^k_{i,j})$ and $\mathsf{Share}_t(\hat{M}^k_{i,j})$.

2. Given the sharings $[M^k_{i,j}]$ and $[\hat{M}^k_{i,j}]$ from $\mathcal{C}^k$, the servers run $(\mathsf{out}, \tilde{S}) \leftarrow \mathsf{FormatVerification}(M, \hat{M})$ (Section 3.4). If out = "Accept" then increment $\mathsf{AcceptedClients}$ and if $\mathsf{AcceptedClients} = N$ go to **Processing**. Otherwise, if out = "Reject", then increment C_q for every $\mathcal{S}_q \in \tilde{S}$. If $C_q > 2(1-\rho)N$ for some $\mathcal{S}_q$ then eliminate $\mathcal{S}_q$ from the rest of the execution.

Robust input 1. For each $(i,j) \in [N']^2$, $\mathcal{C}^k$ and the servers run $\mathsf{RobustInput}(M^k_{i,j})$ and $\mathsf{RobustInput}(\hat{M}^k_{i,j})$.

2. Given the sharings $[M^k_{i,j}]$ and $[\hat{M}^k_{i,j}]$ from $\mathcal{C}^k$, the servers run $(\mathsf{out}, \tilde{S}) \leftarrow \mathsf{FormatVerification}(M, \hat{M})$ (Section 3.4). If out = "Accept" then increment $\mathsf{AcceptedClients}$ and if $\mathsf{AcceptedClients} = N$ go to **Processing**. Otherwise, if out = "Reject", then for every $\mathcal{S}_q \in \tilde{S}$ eliminate $\mathcal{S}_q$ from the rest of the execution.

Processing The servers compute $[A_{(i,j)}] = \sum_{k=1}^{N} [M^k_{(i,j)}]$ for every (i,j).

Output 1. The servers run $A_{(i,j)} \leftarrow \mathsf{Reconstruct}(\{A_{(i,j),q}\}_{q\in[n]})$ and $A_{(i,j)} \leftarrow \mathsf{Reconstruct}(\{\hat{A}_{(i,j),q}\}_{q\in[n]})$ for every (i,j) and output matrices A and $\hat{A}$.

2. For each (i,j) if $(A_{i,j})^2 \neq \hat{A}_{i,j}$ then solve the system $m_1 + m_2 = A_{i,j}$; $(m_1)^2 + (m_2)^2 = \hat{A}_{i,j}$. If there is a solution then output m_1 and m_2.

Remark 1. Even though the robust input sub-protocol is more expensive than simply sharing the blind message $(M, \hat{M})$, a robust input will be run for at most $2(1-\rho)N$ times by adding the following rule to the protocol: Store a counter C_q for each server $\mathcal{S}_q$; whenever a reconstruction is not perfect, increment C_i for all those servers $\mathcal{S}_i$ who do not agree with the reconstructed polynomial; when C_i exceeds $(1-\rho)N$ then eliminate server $\mathcal{S}_i$ and ignore it for the reset of the execution. Note that the adversary who controls $(1-\rho)N$ clients and t servers cannot cause $C_i > (1-\rho)N$ for a honest $\mathcal{S}_i$ since only corrupted clients may cause C_i to increment. In contrast whenever a corrupted server uses a bad share its counter is incremented. Thus, there are at most $(1-\rho)N$ times in which the adversary may 'blame' honest parties, in which case we do not want to eliminate the servers. Then, there are additional $(1-\rho)N$ times in which the corrupted servers may cheat without being eliminated.

Remark 2. After the execution of a robust input ($\mathsf{RobustInput}$), the servers perform the verification check again, but this time, whenever a reconstruction is not perfect we know immediately that the server who do not agree with the polynomial were cheating and we eliminate them for the rest of the execution.

3.3 Format Verification: Preliminaries

A malicious client, or a coalition of malicious clients, might try to disrupt the operation of Blinder in various ways, by sending a malformed message or using a different distribution for the indices than required in 3.2.1. For example:

- A coalition of clients can pick the same $(i^\star, j^\star)$ for their messages, which might damage the reconstruction success rate analysis mentioned in 3.2.3.
- A malicious client might fill two or more entries, instead of one, in its blinded message M, and hope that since the servers do not actually learn the content of the blinded message they would not detect it. In the extreme case, two malicious clients who blow up the entire matrix with messages might cause a DoS attack on Blinder since our implementation successfully reconstructs up to two messages per entry (thus, all 'honest' messages are un-reconstructible).
- Even a single client who writes to a single entry may damage the matrix and decrease reconstruction success rate by writing m to $A_{(i^\star,j^\star)}$ and $m' \neq m^2$ to $\hat{A}_{(i^\star,j^\star)}$. This actually fills entry $(i^\star, j^\star)$ with 2 messages so a message of a honest client who writes to $(i^\star, j^\star)$ will not be extracted. This way, the adversary essentially doubles its power.
- Finally, a client may deal inconsistent shares to the servers such that when trying to reconstruct a message in some entry, even robust reconstruction will fail (e.g. if there are $2t$ shares that disagree with the polynomial).

To this end, before the servers aggregate a blinded message M^k from client $\mathcal{C}^k$, they perform a format verification sub-protocol that detects any kind of deviation from the message format dictated in 3.2.1.Before we describe the format verification protocol let us first present the relevant building blocks.

3.3.1 Coin flip.

We want the servers to collaboratively draw a value $r \in \mathbb{F}$ uniformly at random, which will be public and known to everyone[5]. We use a sub-protocol $\mathsf{RandomPublic}()$ for sampling a random value from $\mathbb{F}$. This can be done via a verifiable secret sharing (VSS)(e.g. [BGW88]): each $\mathcal{S}_q$ picks a uniformly random r_q and deals r_q among the parties using a VSS. Then, all parties reconstruct each r_i and output the value $r = \sum_{q=1}^{n} r_q$. When a large number of public random values is needed, the servers may generate a few and use them as a seed to a pseudorandom generator (PRG) in order to locally produce many more.

3.3.2 Random sharing.

In 3.1 we described how one can share a known value (secret) toward the servers. However, in some cases, it is required that the servers maintain a sharing $[r]$ where r is a uniformly random value from $\mathbb{F}$ and is kept secret from *all servers*. This can

[5] In fact, it will be immediately known to all servers. A client who wish to learn the result of a coin flip has to query at least $t + 1$ of the servers.

be achieved by having each server $\mathcal{S}_q$ picking a uniformly random secret $r_q \in \mathbb{F}$ and sharing r_q toward all servers, so the servers maintain $[r_q]$ for every $q \in [n]$. Then, the servers can locally compute $[r] = \sum_{q=1}^{n} [r_q]$. Note that even if only one server is honest, the final sharing $[r]$ is uniformly random and secret from all servers. An issue with that protocol is that the resulting sharing might not be consistent, hence, it is futile to use it in further computation. That is, a malicious server $\mathcal{S}_q$ may cheat by choosing a polynomial of a greater degree $t' > t$, which leads to the sharing $(r_q^{(1)}, \ldots, r_q^{(n)})$ on a polynomial of degree t'. If there are more than t shares that disagree with any polynomial of degree t then r_q could not be reconstructed, even by $t+1$ honest servers. Similarly, the value $r = \sum_{q=1}^{n} r_q$ could not be reconstructed. To solve that, the servers have to verify that the sharings dealt by each server are of degree at most t, without revealing the shared values. In Section 5.1 we describe a protocol that does exactly that and obtain random sharings $[r]$ that are perfectly consistent, that is, all parties' shares reside on a unique polynomial of degree t that hides r. We refer to such a protocol by $[r] \leftarrow \mathsf{RandomSecret}()$. So far we mentioned a t-sharing, however, the discussion above completely holds also to the case of generating a $2t$ sharing, which is denoted by $\langle r \rangle \leftarrow \mathsf{RandomSecret}()$.

In the rest of this section and in the next section we assume that outputs of $\mathsf{RandomSecret}()$ are sharings of degree at most t (or $2t$, depending on the context) as required.

3.3.3 Degree reduction and Output.

As mentioned in 3.1.2, the servers can locally apply any linear function to the sharings and obtain a valid share of the same degree. In addition, given two t-sharings $[x_1]$ and $[x_2]$ the servers can locally compute $\langle y \rangle = [x_1] \cdot [x_2]$ to obtain a sharing of degree $2t$. This raises two problems: (1) if the servers want to keep computing on the value y while y is secret. For example, multiplying y by another $2t$-shared secret z (i.e. $\langle z \rangle$) resulting in a $4t$-sharing of $y \cdot z$. But now, as discussed in 3.1.1, a *robust* reconstruction of $y \cdot z$ is no longer possible, even in case that only t servers cheat; (2) suppose y should be output to the servers; having the servers naively reconstruct $\langle y \rangle$ by having each server $\mathcal{S}_q$ broadcasting its share y_q would not be secure, as it might leak information on the multiplicands x_1 and x_2.

One approach for solving these problems proposed by Beaver [Bea91]. The idea is to preprocess a 'multiplication triple' $([a], [b], [c])$ with $c = ab$, such that, to multiply $[x]$ and $[y]$ the servers essentially do operations on t-sharings only. Specifically, the servers (locally) compute $[d] = [x] + [a]$ and $[e] = [y] + [b]$ and reconstruct the t-sharings d and e. Then, they locally compute $[x \cdot y] = [c] + e \cdot [y] + d \cdot [x] - ed$. However, this 'pushes' the dealing with $2t$ sharing to the offline (preprocessing) phase of the protocol, in order to obtain the sharing $[c] = [a \cdot b]$. Due to this reason, such a solution is sufficient for achieving fairness, since cheating in the preprocessing can be detected, which leads to an abort of the protocol. This way no input/output learnt by the adversary. On the other hand, if there is no cheat in the preprocessing then it is guaranteed that both the adversary and the honest parties will obtain the output of the computation. This is the approach taken by a recent AsynchroMix and PowerMix [LYK+19].

Nevertheless, in this work we aim to achieve *robustness*. So causing the protocol to

abort, even in the preprocessing phase where the adversary learns nothing, is unacceptable. This forces us to use stronger techniques, rather than the above multiplication triples, that are resilient to cheats even when dealing with $2t$-sharings. Furthermore, using the multiplication triplets approach does not allow for a faster MPC that Blinder relies on, as described in Section 4.

Our approach. We follow a protocol proposed by Damgard and Nielsen [DN07]. Specifically, [DN07] suggests using a simpler preprocessing, in which the servers generate *double random sharings*. A double random sharing allows re-randomization and reducing the degree of the polynomial that hides $y = x_1 \cdot x_2$ from $2t$ to t. Multiplication of $[x_1]$ and $[x_2]$ using the double random sharing technique works as follows: In the preprocessing phase the servers generate tuples of the form $([r], \langle r \rangle)$. Namely, a t and $2t$-sharings of the same secret random value $r \in \mathbb{F}$. Then, in the online phase, in order to obtain $[x_1 \cdot x_2]$ the servers first locally compute $\langle y \rangle = [x_1] \cdot [x_2]$, they reconstruct $e = \langle y \rangle - \langle r \rangle$ and then locally compute $e + [r] = [y - r + r] = [x_1 \cdot x_2]$. This technique solves both the problems described above as it produces a t-sharing of the product $x_1 \cdot x_2$ using a polynomial with fresh randomness. As show in Section 4, this technique enables a dramatic optimization to the operation of Blinder. We denote the above protocol by $[y] \leftarrow \mathsf{Mult}([x_1], [x_2])$. In Section 5.1 we show how the servers robustly generate tuples of the form $([r], \langle r \rangle)$.

Finally, we note that when the product $\langle y \rangle = [x_1] \cdot [x_2]$ is to be output rather than being part of further computation, then there is no need for degree reduction, but only for randomization. To this end, we require the servers to generate random $2t$-sharings of the value 0 (zero). Then, instead of reconstruct $\langle y \rangle$ directly, the servers reconstruct $\langle y \rangle + \langle 0 \rangle$. We denote that protocol by $y \leftarrow \mathsf{open}(\langle y \rangle)$.

3.4 Format Verification Protocol

To ease the presentation we treat M and $\hat{M}$ as vectors rather than matrices, e.g. we write $[M_1], \ldots, [M_N]$ to refer to the entries of M and write $i^\star$ to refer the index chosen by the client (instead of $(i^\star, j^\star)$). Upon receiving a sharing of a blinded message $(M, \hat{M})$ (see 3.2.1-3.2.3) the servers verify that the following holds.

1. **Random index of message.** We want the index of the message $(m, \hat{m})$ hidden in $(M, \hat{M})$ be uniformly random, in order to avoid complex success rate analysis (see 3.2.3).

2. **Consistent sharing.** The sharings of entries in M and $\hat{M}$ must be perfectly consistent (all shares reside on a single polynomial of degree t).

3. **Single non-zero entry.** The entries of M and $\hat{M}$ must be all zero, except one entry that contains the message m and m^2 in M and $\hat{M}$, respectively.

4. **Non-zero in the same entry.** The non-zero entries in M and $\hat{M}$ must be at the same index $i^\star$.

5. **Squared message.** If $M_{i^\star} = m$ then $\hat{M}_{i^\star} = m^2$. This is necessary for being able to recover from a collision (see 3.2.3).

We now turn to describe how the servers perform the above checks without revealing anything about the client's message, m, or its position, $i^\star$:

1. Instead of *verifying* item (1) we *enforce* it, that is, the servers make sure that the index of m within M is uniformly random, without actually knowing what is that index. Specifically, sample a public random value $r \in [N]$ and shift-left the vectors by r positions s.t. $M \leftarrow [M_{r+1}], \ldots, [M_N], [M_1], \ldots, [M_r]$ and $\hat{M} \leftarrow [\hat{M}_{r+1}], \ldots, [\hat{M}_N], [\hat{M}_1], \ldots, [\hat{M}_r]$. We stress that the servers do not know the final index of m even though they know r since the initial index of m secretly chosen by the client. We denote this operation by $\mathsf{ShiftLeft}(M, \hat{M}, r)$.

2. In order to check consistency of sharings of $M_1, \ldots, M_N$ (and similarly for $\hat{M}_1, \ldots, \hat{M}_N$) in item (2), one can simply apply a random linear combination on the sharings and attempt reconstructing the result. Specifically, the servers sample public random values $r_1, \ldots, r_N$ and a random secret value $[\tilde{r}]$. They compute
$$[\alpha] = [\tilde{r}] + \sum_{i=0}^{N} r_i \cdot [M_i]$$
and reconstruct α. There are two cases:

   - Reconstruction is perfect, i.e. all shares reside on the same t-polynomial, then we conclude that all sharings of $M_1, \ldots, M_N$ are of degree t with probability $1 - 1/|\mathbb{F}|$. Otherwise, if at least one sharing was of degree greater than $t' > t$, then the coefficient of $x^{t'}$ in the polynomial of that sharing was non-zero. This result in a uniformly random coefficient of $x^{t'}$ in the polynomial obtain when reconstructing $\tilde{r} + \sum_{i=0}^{N} r_i \cdot M_i$, which is zero with probability at most $1/|\mathbb{F}|$.
   - Reconstruction is not perfect. This may be caused by either a malicious client or a malicious server, however, it is impossible to tell which one. The servers inform the client that its submission was rejected and the client turn to submit its message again using a robust input, described in 5.2.

3. To verify item (3), Boyle et. al. [BGI16, BBC+20] present a linear sketch for the language of vectors of hamming weight one and the non-zero coordinate is from $\mathbb{F}$. That is, that is an equation that holds only if a given vector $w = (w_1, \ldots, w_\ell)$ has at most one non-zero entry[6]. The equation is $(\sum_{i=1}^{\ell} w_i \cdot r_i)^2 - m(\sum_{i=1}^{\ell} w_i \cdot r_i^2) = 0$ where w_i are the vector's entries, r_i are public random values generated *independently* of w and m is the value in the single non-zero entry. In [BGI16, BBC+20] (followed by Express [ECZB19]) the client has to input m along with the vector, however we observe that in our case the servers can simply obtain it by computing $m = \sum_{i=1}^{\ell} w_i$. So, given $[w_1], \ldots, [w_n]$

[6] In fact, the equation might hold even if the vector has more than one non-zero entry, but this occurs with probability at most $1/|\mathbb{F}|$

(where $[w_i] = [M_i]$), the servers locally compute $[m] = \sum_{i=1}^{N}[w_i]$ and

$$\langle\beta\rangle = (\sum_{i=1}^{N}[w_i]\cdot r_i)^2 - [m](\sum_{i=1}^{N}[w_i]\cdot r_i^2)$$

Finally, they reconstruct β and verify it equals zero. Note that β is shared via a $2t$-sharing because in the right side of the equation there are multiplications of two t-sharings.

4. To verify item (4), we combine M and $\hat{M}$ in a random fashion to a single vector and perform the above check on the combined one. Specifically, sample $r_1, \ldots, r_N$ and $\hat{r}_1, \ldots, \hat{r}_N$ and set $[w_i] \leftarrow r_i \cdot [M_i] + \hat{r}_i \cdot [\hat{M}_i]$ for all $i \in [N]$. Then, send $[w_1], \ldots, [w_N]$ to the verification in the above item. Clearly, this verifies that there is only one entry in M and $\hat{M}$ that is non-zero *and* that those entries are at the same index. Suppose to the contrary, then for each index of a non-zero entry in M and $\hat{M}$, we get a non-zero entry in w with probability $1 - 1/|\mathbb{F}|$. By a union bound, the probability to pass that verification is $2/|F|$.

5. Finally, to verify item (5), i.e. that $(M_{i^\star})^2 = \hat{M}_{i^\star}$, the servers can compute

$$\langle\gamma\rangle = \langle(M_{i^\star})^2 - \hat{M}_{i^\star}\rangle = (\sum_{i=1}^{N}[M_i])^2 - (\sum_{i=1}^{N}[\hat{M}_i])$$

reconstruct γ and check it equals zero. Obviously, if M and $\hat{M}$ are guaranteed to have at most one non-zero entry then passing this check indeed guarantees that $(M_{i^\star})^2 = \hat{M}_{i^\star}$.

The formal description of the format verification protocol follows:

Protocol $\mathsf{FormatVerification}(M, \hat{M})$

1. Run $r \leftarrow \mathsf{RandomPublic}()$ and $\mathsf{ShiftLeft}(M, \hat{M}, r)$.
2. Run $r_i \leftarrow \mathsf{RandomPublic}()$ for $i \in [N]$ and $[r] \leftarrow \mathsf{RandomSecret}()$. Then the servers locally compute $[\alpha] = [\tilde{r}] + \sum_{i=1}^{N} r_i \cdot [M_i]$ and $\alpha \leftarrow \mathsf{Reconstruct}([\alpha])$. Do the same with shares of $\hat{M}$ to obtain $\hat{\alpha}$. Reconstruct α and $\hat{\alpha}$. If reconstruction is perfect (all shares reside on a single polynomial) then continue, otherwise, output "Reject" and the set of servers $\tilde{S} \subset \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ that do not agree with the polynomial.
3. For $i \in [N]$ do as follows: Run $r_i \leftarrow \mathsf{RandomPublic}()$ and $\hat{r}_i \leftarrow \mathsf{RandomPublic}()$. Then compute $[w_i] = r_i \cdot [M_i] + \hat{r}_i \cdot [\hat{M}_i]$. In addition, compute $[m] = \sum_{i=1}^{N}[w_i]$.
4. Run $r_i \leftarrow \mathsf{RandomPublic}()$ for $i \in [N]$. Compute $\langle\beta\rangle = (\sum_{i=1}^{N}[w_i]\cdot r_i)^2 - [m](\sum_{i=1}^{N}[w_i]\cdot r_i^2)$ and reconstruct β. If reconstruction is perfect and $\beta = 0$ then continue, otherwise, output "Reject" and the set of servers $\tilde{S} \subset \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ that do not agree with the polynomial.

5. Compute $[m] = \sum_{i=1}^{N}[M_i]$ and $[\hat{m}] = \sum_{i=1}^{N}[\hat{M}_i]$. Then, compute $\langle\gamma\rangle = [m] \cdot [m] - [\hat{m}]$ and reconstruct γ. If reconstruction is perfect and $\gamma = 0$ then continue, otherwise, output "Reject" and the set of servers $\tilde{S} \subset \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ that do not agree with the polynomial.

6. Output "Accept" and $\emptyset$.

Remark. So far we considered $L = 1$, however, handling larger messages is straightforward: Given M and $\hat{M}$, each of size $N \cdot L$ field elements, the verification procedure first 'compress' those to vectors of size N and perform the same protocol FormatVerification on the compressed versions. Specifically, for the ith block of L field elements of M, i.e. sharings $[M_{i\cdot L+1}], \ldots, [M_{(i+1)L}]$ compute $Q_i \leftarrow \sum_{j=1}^{L} r_j \cdot [M_{i\cdot L+j}]$ and similarly $\hat{Q}_i \leftarrow \sum_{j=1}^{L} r_j \cdot [\hat{M}_{i\cdot L+j}]$ where $r_1, \ldots, r_L$ are generated using RandomPublic(). Then use vectors Q and $\hat{Q}$ in the FormatVerification protocol.

3.5 Efficiency

Let us consider the efficiency of the basic construction. Each client deals two matrices, M and $\hat{M}$, of $O(N)$ secrets, leading to a total of $O(N^2)$ client-server communication overhead. Then the format verification protocol incurs a $O(N)$ multiplications by constants and three reconstructions *per client* of values α, β and γ. Revealing all messages in A and $\hat{A}$ incurs $O(N)$ reconstructions as well. In total this is $O(N^2)$ of computational overhead and $O(N)$ of server-server communication.

Of course, the bottleneck in that protocol is the $O(N^2)$ client-server communication overhead, which makes the protocol impractical for a large number of clients (hundreds of thousands to millions). We significantly improve this in Section 4.

3.6 Security

Our protocol can be described as an enhanced arithmetic circuit with $2 \cdot N \cdot L$ inputs from $\mathbb{F}$. By 'enhanced' we mean that, in addition to addition and multiplication gates, it contains also random gates, input gates and output gates. Addition gates are performed locally. As for multiplication, random, input and output gates, those are defined by the sub-functionalities $\mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathrm{rand}}, \mathcal{F}_{\mathrm{input}}$ and $\mathcal{F}_{\mathrm{reconst}}$ in previous works [DN07, CGH+18]. The security of Blinder builds upon the existence of a simulator for those functionalities (gates). We prove the following theorem:

Theorem 3.1 *Protocol* BlinderProtocol *(Section 3.2.4) securely computes the ACB functionality (Definition 2.1) in the* $(\mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathrm{rand}}, \mathcal{F}_{\mathrm{input}}, \mathcal{F}_{\mathrm{reconst}})$*- hybrid model.*

Proof 3.1 *For correctness, consider a setting with honest servers and clients, since the message submitted by each client is destined to a uniformly random entry in the table* $(A, \hat{A})$*, the distribution of the outputs after running the protocol is identical to the distribution described in the functionality.*

All protocols, for functionalities ($\mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathrm{rand}}, \mathcal{F}_{\mathrm{input}}, \mathcal{F}_{\mathrm{reconst}}$ have a secure and robust implementation, thus, since we describe Blinder's protocol solely based on them, it implies the existence of a simulator. However, in contrast to [DN07, CGH+18] in which the servers themselves are also the clients, in Blinder, there is a distinction between those roles, which adds complication to the robustness and censorship resistance arguments.

In the protocol, the robustness property is required every time the servers are trying to reconstruct a shared value. Those cases appear in the construction of values α, β, γ for every client as well as at the very end of the protocol in which the servers reconstruct each entry of A and $\hat{A}$. In the former, corrupted servers might cause a message of a honest client to get discarded, but then that client uses a robust input sub-protocol, by which it could not get discarded and any cheating is identified, followed by elimination of the cheating servers, thus, this could not happen more than t times. In addition, as discussed above, the adversary may indeed slow down the execution by 'forcing' honest clients to use a robust input rather than simple secret sharing, but this could happen for at most $2(1-\rho)N$ times. Finally, providing a bad share when reconstructing the entries of A and $\hat{A}$ immediately identify the cheating servers, who are eliminated as before. Of course, this argument relies on the fact that the preprocessing (generating random double sharings) and the input sub protocols are robust as well. We present those sub protocols in Section 5.

We note that an honest client may cause the reconstruction of α and β to fail, by submitting inconsistent sharings in the first place. But in this case the worst that can happen is having the servers rightfully discard that client's message so it cause no harm to the aggregation process. Similarly, servers discard a submission for which the verification result with non-zero α and β as it does not adhere to the blinded message format.

4 Scaling Blinder

In this section we make use of a property observed in the work on general secure computation by Chida et. al. [CGH+18]. Namely, when we have vectors of shares $a = ([a_1], \ldots, [a_\ell])$ and $b = ([b_1], \ldots, [b_\ell])$, the naive way to compute $[c] = \sum_{i=1}^{\ell} [a_i][b_i]$ would be to perform $[c_i] \leftarrow \mathsf{Mult}([a_i], [b_i])$ (by the sub-protocol described in 3.3.3) and then sum up $[c] = [c_1] + \ldots, [c_\ell]$. This approach incurs the call to Mult for ℓ times, where each such a call costs $O(|\mathbb{F}|)$ communication per server. Alternatively, it is observed that locally multiplying $[a_i][b_i]$ results in $\langle c_i \rangle$, a $2t$-sharing of $c_i = a_i \cdot b_i$. Thus, we can sum up $\langle c \rangle = \langle c_1 \rangle + \ldots, \langle c_\ell \rangle$ already in the higher degree and then call the degree reduction on $\langle c \rangle$ only once, which costs $O(|\mathbb{F}|)$ communication for the entire operation. This 'sum of product' operation is used in [CGH+18] to make MPC protocols over small fields secure, but we adapt it to improve the efficiency of our protocol even when using a large field $\mathbb{F}$.

In addition, we utilize another technique from the PIR context. Namely, 'compressing' the blind message size of a client from $O(N)$ to $O(\sqrt{N})$. Then, the servers can de-compress the blind message by using the multiplication sub-protocol. Overall, we reduce the client-server communication to $O(\sqrt{N})$ without introducing any other

communication between the servers (i.e. it remains $O(N)$).

Finally, we improve the communication complexity of the format verification protocol from $O(1)$ per client to $O(1)$ *for all clients* in the optimistic case (assuming honest clients) and to $O(c \cdot \log N)$ where $c = (1-\rho)N$ in the pessimistic case.

4.1 Efficient construction of Blinder

We revise the protocol described in Section 3.2. For simplicity, suppose that N is a perfect square and $\mathsf{rows} = \mathsf{cols} = \sqrt{N}$.

4.1.1 Submitting a Message.

Instead a full matrix M of size $O(N)$, the blind message can be only two short vectors: a row vector $\boldsymbol{r}$ and a column vector $\boldsymbol{c}$, each of size $\sqrt{N}$, with elements from $\mathbb{F}$. Denote by $\boldsymbol{r}_i, \boldsymbol{c}_i$ the ith coordinate of $\boldsymbol{r}$ and $\boldsymbol{c}$, respectively.

A client $\mathcal{C}$ with a message m randomly picks $(i^\star, j^\star)$ and assigns $\boldsymbol{c}_{i^\star} = 1$,and $c_i = 0$ for every $i \neq i^\star$; likewise, $\boldsymbol{r}_{j^\star} = m$, and $\boldsymbol{r}_j = 0$ for every $j \neq j^\star$. Observe that $\boldsymbol{c} \times \boldsymbol{r}$ is exactly the matrix M from Eq.(1). Similarly, $\mathcal{C}$ prepares vectors $\hat{\boldsymbol{r}}$ and $\hat{\boldsymbol{c}}$ instead of the matrix $\hat{M}$, where $\hat{\boldsymbol{c}}_{i^\star} = 1$, and $\hat{\boldsymbol{c}}_i = 0$ for every $i \neq i^\star$; likewise $\hat{\boldsymbol{r}}_{j^\star} = m^2$, and $\hat{\boldsymbol{r}}_j = 0$ for every $j \neq j^\star$. Vectors $(\boldsymbol{r}, \boldsymbol{c})$ and $(\hat{\boldsymbol{r}}, \hat{\boldsymbol{c}})$ constitute the new blinded message. Notice that $\boldsymbol{c} = \hat{\boldsymbol{c}}$, so the client sends only 3 vectors $\boldsymbol{r}, \hat{\boldsymbol{r}}$ and $\boldsymbol{c}$, which are used to compute both $M = \boldsymbol{c} \times \boldsymbol{r}$ and $\hat{M} = \boldsymbol{c} \times \hat{\boldsymbol{r}}$. Nevertheless, in the following we still consider 4 vectors in order to make their purpose explicit. $\mathcal{C}$ shares those vectors toward all servers. Denote the share of the i-th coordinate of vector $\boldsymbol{x}$ sent to server $\mathcal{S}_q$ by $\boldsymbol{x}_{i,q}$, or use $\boldsymbol{x}^k_{i,q}$ to indicate that client $\mathcal{C}^k$ sends that share. To refer to $\mathcal{S}_q$'s share of the entire vector we simply write $\boldsymbol{x}_q$.

4.1.2 Processing and revealing all messages.

Given the shares $\boldsymbol{r}^k_q, \boldsymbol{c}^k_q$ and $\hat{\boldsymbol{r}}^k_q, \hat{\boldsymbol{c}}^k_q$ from client $\mathcal{C}^k$, server $\mathcal{S}_q$ first computes $M^k_{(i,j),q} = \boldsymbol{c}^k_i \cdot \boldsymbol{r}^k_j$ and $\hat{M}^k_{(i,j),q} = \hat{\boldsymbol{c}}^k_i \cdot \hat{\boldsymbol{r}}^k_j$ for every i, j, obtaining the matrices M^k_q and $\hat{M}^k_q$ for all $k \in [N]$. This means that the servers have a $2t$-shares for every entry (i, j) of the matrices M^k and $\hat{M}^k$ for every client $\mathcal{C}^k$, these are denoted $\langle M^k_{(i,j)} \rangle$ and $\langle \hat{M}^k_{(i,j)} \rangle$. Note, however, that the servers do not need to reduce the degree of the sharing for each client individually (something that would lead to $O(N^2)$ communication overhead). As mentioned above, the severs can reduce the degree only after summing up the $2t$-sharings *of all clients*. That is, for each (i, j) the servers compute $\langle A_{(i,j)} \rangle \leftarrow \sum_{k=1}^{N} \langle M^k_{(i,j)} \rangle$ and $\langle \hat{A}_{(i,j)} \rangle \leftarrow \sum_{k=1}^{N} \langle \hat{M}^k_{(i,j)} \rangle$. Finally, the servers reconstruct the sharings $\langle A_{(i,j)} \rangle$ and $\langle \hat{A}_{(i,j)} \rangle$ at each entry (i, j) to reveal the message(s) and their squares at $A_{(i,j)}$ and $\hat{A}_{(i,j)}$ respectively.

4.1.3 Format verification.

Naively, the servers can de-compress the blind message M and $\hat{M}$ and then perform the same format verification protocol described in 3.4. However, we observe that the exact

same check, at the same security guarantees, could be performed, more efficiently, *before* de-compression. Specifically, to force a random index of the message the servers can shift left each of the shorter vectors $\boldsymbol{r}, \hat{\boldsymbol{r}}$ by a random value and the vectors $\boldsymbol{c}, \hat{\boldsymbol{c}}$ by another independent random value, this completely re-randomizes the index of the message. In addition, note that the same verification check that is described in 3.4 should be taken here. That is, servers have to verify that the shares are consistent, that there is at most one non-zero entry in $\boldsymbol{r}, \hat{\boldsymbol{r}}, \boldsymbol{c}$ and $\hat{\boldsymbol{c}}$, that the non-zero entry in $\boldsymbol{r}$ and $\hat{\boldsymbol{r}}$ has the same index (likewise for $\boldsymbol{c}$ and $\hat{\boldsymbol{c}}$) and that the non-zero entry at $\hat{\boldsymbol{r}}$ is square of the non-zero entry in $\boldsymbol{r}$. Thus, it is sufficient to call $\mathsf{FormatVerification}(\boldsymbol{r}, \hat{\boldsymbol{r}})$ and $\mathsf{FormatVerification}(\boldsymbol{c}, \hat{\boldsymbol{c}})$ to complete the verification.

4.1.4 Efficiency.

There is a trade-off between computation and communication. That is, in Section 3.2 the servers received the complete matrices M and $\hat{M}$ from each client and all they had to do is to verify their format and then sum them all up. That approach required $O(N^2)$ client-server communication. In contrast, here we need only $O(N^{1.5})$ communication, but the servers have to perform $O(N)$ field multiplications to de-compress a client's blind message $(M, \hat{M})$ and $O(N^2)$ multiplications overall. Although the finite field arithmetic is fast, when number of clients is large (hundreds of thousands to millions) this becomes the bottleneck of Blinder.

4.2 Batching Format Verification

Using the same technique, the servers can perform the format verification for all clients in one shot (in an optimistic case). Recall that in the format verification the servers produce $[\alpha], \langle\beta\rangle$ and $\langle\gamma\rangle$. They reconstruct all of them and verify that all shares of each of them reside on a single polynomial (i.e. perfect reconstruction). In addition, they verify that β and γ equal zero. The reconstructions are the operations that cost in communication, therefore, the servers can delay it and verify all clients. Specifically, let $[\alpha_k], \langle\beta_k\rangle$ and $\langle\gamma_k\rangle$ be the sharings obtained by performing the verification for client $\mathcal{C}^k$. The servers aggregate all α's and reconstruct only $[\sum_{k=1}^{N} \alpha_k]$. In addition, they aggregate all β's and γ's and reconstruct $[\sum_{k=1}^{N} \beta_k + \gamma_k]$ and verify the result equals zero. The probability that the result equals zero if there are c malformed messages is at most $1/|\mathbb{F}|^c$ since each β_k and γ_k for a corrupted client will be zero with probability $1/|\mathbb{F}|$ and there are c of them.

Notice that even a single client can cause the result to be different than zero or cause the reconstruction fail or be imperfect, which leads to a verification failure. In that case the servers can perform a binary search to find k for which $\alpha_k, \beta_k, \gamma_k$ led to a verification failure. Finding k requires $O(\log N)$ communication (i.e. a reconstruction of a branch in a tree with N leaves). Thus, for c corrupted clients the overall communication will be $O(c \cdot \log N)$. In addition, finding all malformed messages incurs $\log N$ rounds.

4.3 Utilizing a GPU

In the above, we showed how to reduce serve to server communication from $O(N^2)$ to $O(N)$, which shifts the bottleneck of Blinder from communication to computation, since computation remains $O(N^2)$. In this section, we argue that the computation that Blinder performs can be accelerated by using a GPU.

The rational of using GPU in Blinder is that the $O(N^2)$ computational bottleneck consists of simple arithmetic (addition and multiplication) operations in a finite field. For such tasks, GPU's demonstrate a much higher throughput, i.e. integer operations per second. For instance, a benchmark in [WWB19], compares a 16 core Skylake CPU with 120GB memory with NVIDIA V100 GPU that contains 8 V100 packages (SXM2) and finds that the CPU is capable of 2 TFLOPS whereas the GPU is capable of 125 TFLOPS. A theoretical improvement of more than $50\times$; in practice this factor becomes even larger due to memory and threads management on a CPU.

The main bottleneck when working with a GPU is the capacity of the channel between the CPU and GPU. We characterize applications that fit a GPU deployment as follows: (1) The input size to the algorithm it has to run should be relatively small; (2) The algorithm itself has to be highly parallelizable, since a GPU has up to thousands of independent cores, each of which can perform some simple task; and (3) the output size should be small as well.

Consider for example the task of computing the cartesian product of N pairs $(\boldsymbol{c}_i, \boldsymbol{r}_i)$ of vectors and receiving back the matrices $\boldsymbol{c}_i \times \boldsymbol{r}_i$ for all i. It is quite easy to deploy this task to a GPU. Each core is given a single pair of vectors, and instructed to compute the matrix $\boldsymbol{c}_i \times \boldsymbol{r}_i$ and hand it back to the CPU. This however, only addresses points (1) and (2), but not (3), as the output size is $N \cdot |\boldsymbol{c}_i|^2$, which becomes a bottleneck when $|c_i|^2$ is large. In contrast, in Blinder we are not interested in the individual result of $\boldsymbol{c}_i \times \boldsymbol{r}_i$, but only in the *sum* of all the resulting matrices, which fits to point (3) as well.

We observe the following: Let $C = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N)$ be a matrix with $\boldsymbol{c}_k$ being its kth column and R be a matrix with $\boldsymbol{r}_k$ being its kth row, s.t. $\boldsymbol{c}_k, \boldsymbol{r}_k$ are part of client $\mathcal{C}^k$'s blind message. Then Blinder essentially computes *matrix multiplication!* We have that:

$$A = C \times R^\top$$

Where A is the matrix from 4.1.2. To efficiently divide that task across many cores, Blinder hands vectors $C' = \boldsymbol{c}_i, \ldots, \boldsymbol{c}_j$ and $R' = \boldsymbol{r}_k, \ldots, \boldsymbol{r}_\ell$ to one GPU core, then that core computes a small matrix multiplication that produce the final entries in matrix A at positions $(i, k), \ldots, (j, \ell)$. Thus, the GPU hands back exactly $(j - i) \cdot (\ell - k)$ entries to the CPU. This way, the overall output handed back from the GPU to the CPU is only of size $|\boldsymbol{c}_i| \times |\boldsymbol{r}_i|$ (rather than $N \cdot |\boldsymbol{c}_i| \times |\boldsymbol{r}_i|$) and so point (3) becomes a non-issue. Fortunately, there is a highly optimized code base for solving matrix multiplication over GPU, which makes Blinder a perfect fit for that hardware.

5 Robust Preprocessing and Input

5.1 Robust Preprocessing

As discussed in 3.3.2-3.3.3, the servers need to generate random double sharings in the preprocessing phase. A double random sharing is a pair $([r], \langle r \rangle)$ where r is sampled randomly from $\mathbb{F}$ and is unknown to any server. One way to generate those is by having each server $\mathcal{S}_q$ choosing random $r_q \in \mathbb{F}$ and dealing $[r_q]$ and $\langle r_q \rangle$. Then all servers conclude with the tuple $([r], \langle r \rangle) = (\sum_{q=1}^{n} [r_q], \sum_{q=1}^{n} \langle r_q \rangle)$ that is a combination of the randomness from all servers. We denote that procedure by $\mathsf{Combine}([r_1], \langle r_1 \rangle, \ldots, [r_n], \langle r_n \rangle)$. We remark that the above procedure produces a single random double sharing from the n sharings contributed by the servers; there exists a more efficient method, using a hyper-invertible matrix [DN07, BH08], to generate $n - t$ random sharings out of those n contributed by the server. Although, we leave that abstract under the $\mathsf{Combine}()$ procedure and proceed by focusing on how to ensure that the sharings that the servers contribute are consistent.

We have to ensure that the sharings that each server contributes are consistent and also that both secrets are the same r_q. Furthermore, as our aim is to guarantee robustness, the production of those sharings must be completed successfully. This is in contrast to other protocols like [BHKL18, CGH+18, LYK+19] (and others) that strive for fairness only or for an even weaker guarantee – MPC with abort. While those protocols simply *abort* if any cheating in the random sharing generation is detected, Blinder has to *recover* from such a scenario. Indeed, [DN07, BH08] recover from a cheating. Specifically, [DN07] does so by applying a complex accounting on the number of 'disputes' each server is involved in, and [BH08] does so by a complex, t-rounds, 'player elimination' technique [BH08].

In this section we suggest a simpler recovering protocol that guarantees that all random sharings are *perfectly consistent*. That is, if $([r], \langle r \rangle)$ is a double random sharing produced by the protocol than it is guaranteed that the shares of *all servers* reside on the same t or $2t$-polynomial. This is in contrast to the online phase in which we may be tolerant to cases where up to t shares *do not* reside on that polynomial.

Suppose that in the online phase the servers need ℓ random double sharings. Our protocol, as previous ones, first instruct each server to choose and deal $([r_i], \langle r_i \rangle)$ for $i = 1, \ldots, \ell + 1$. Then, the servers run $\mathsf{RandomPublic}()$ to produce ℓ public random values $s_1, \ldots, s_\ell$ and locally obtain the sharings $[a] = \sum_{i=1}^{\ell+1} s_i \cdot [r_i]$ and $\langle b \rangle = \sum_{i=1}^{\ell+1} s_i \cdot \langle r_i \rangle$. The servers reconstruct the secrets a and b to verify that $a = b$ and both sharings are perfectly consistent.

Now, our protocol relies on the following observation: in the reconstruction, server $\mathcal{S}_q$ opens its shares a_q and b_q and then the servers run the reconstruction algorithm to find a and b, that is, a single polynomial that 'agrees' with at least $2t + 1$ shares. There are the following cases:

- Perfect reconstruction – all shares are on the same polynomial. This is the best case, the servers continue the protocol.
- Reconstruction succeeds – but at most t shares disagree with the polynomial. In this case there are two options: (1) the dealer was corrupted and dealt t bad

shares; (2) the dealer was honest and at most t servers reported bad shares.

- Reconstruction fails – more than t shares are out of polynomial. This is also a good case, because such a case is not possible without a corrupted dealer.

In the first and third cases above the situation is clear. In the first case all servers behave honestly so the servers can continue the protocol while in the third case it is clear that (at least) the dealer is cheating, in which case we eliminate the dealer and continue the protocol without it.

In the second case we cannot tell whether the dealer or the servers with bad shares were cheating, but we are sure that t out of those $t+1$ servers (dealer plus at most t servers that disagree with the polynomial) are corrupted. We treat that case by eliminating both the dealer and one of the servers with bad shares, this means that we eliminate two parties, knowing that at least one of them is corrupted. This action does not harm the robustness capabilities of the rest of the protocol. Suppose that in the preprocessing phase we eliminate $2k$ servers due to the second case. Then we remain with $4t+1-2k = 2t+1+2k'$ servers, for some $k' = t-k$. This means that we are still able to run the algorithm for robust reconstruction. That is, given $2t+1+2k'$ shares of a $2t$- (or t) polynomial, of which up to k' are 'bad', the robust reconstruction algorithm discussed in 3.1.1 still works.

A formal description of the protocol follows:

Protocol $\mathsf{Preprocess}(\ell)$

1. Initialize $S = [n]$, the set of servers that will remain to the online phase of the protocol.
2. For $q \in [n]$ do (in parallel):
   (a) $\mathcal{S}_q$ chooses $r_1, \ldots, r_\ell$ and deals sharings $[r_1], \ldots, [r_{\ell+1}]$ and $\langle r_1 \rangle, \ldots, \langle r_{\ell+1} \rangle$.
   (b) All servers run $\mathsf{RandomPublic}()$ to generate $s_1, \ldots, s_{\ell+1}$.
   (c) The servers (locally) compute $[a] = \sum_{i=1}^{\ell+1} s_i \cdot [r_i]$ and $\langle b \rangle = \sum_{i=1}^{\ell+1} s_i \cdot \langle r_i \rangle$.
   (d) Server $\mathcal{S}_q$ broadcasts a_q and b_q and the servers attempt to reconstruct a and b. There are 3 cases:
      - Reconstruction succeeds – all a_i (and resp. b_i) reside on the same polynomial. If $a = b$ then continue the protocol, otherwise, if $a \neq b$, remove q from S.
      - Reconstruction succeeds – but $k \leq t$ shares disagree with the polynomial that hides a and b. If $a \neq b$ then remove q from s. Otherwise, if $a = b$, denote the set of indices of disagreeing servers by $C \subset [n]$. The servers randomly pick $c \in C$ (possibly using $\mathsf{RandomPublic}()$) and remove c and q from S and continue.
      - Reconstruction fails – meaning $k > t$ shares do not reside on the polynomial that hides a and b (the polynomial actually could not be found). Then remove q from S and continue.
   (e) For each $q \in S$, let $[r_{q,1}], \ldots, [r_{q,\ell}]$ and $\langle r_{q,1} \rangle, \ldots, \langle r_{q,\ell} \rangle$ be the first ℓ sharings that $\mathcal{S}_q$ dealt.
   (f) Let $n' = |S| = 2t+1+2k'$. Denote the indices in S by $q_1, \ldots, q_{n'}$.

(g) For each $i \in [\ell]$ the servers output

$$\mathsf{Combine}([r_{q_1,i}], \langle r_{q_1,i} \rangle, \ldots, [r_{q_{n'},i}], \langle r_{q_{n'},i} \rangle)$$

5.2 Robust Input

Recall that in the FormatVerification protocol (Section 3.4) the servers output "Accept" only if reconstruction of α, β and γ is perfect. That is, all shares of α reside on the same polynomial (and similarly for β and γ). We note that the 'optimistic input' step raises issues that prevent it from being robust. When the sharings are 'just' consistent (but not perfectly consistent) then it is impossible to tell who cheated; it could be the client who dealt incorrect shares to the servers or the servers who indeed received correct shares, but used incorrect shares in the protocol. This leads to the following attack: Two malicious clients could cause Blinder to halt: the first client deals incorrect shares to a set of t servers and the second client deals incorrect shares to a *disjoint* set of t servers. Since there are at most t incorrect shares in the verification of each client, the resulting sharings $[\alpha], \langle b \rangle$ and $\langle \gamma \rangle$ are indeed consistent and both messages are accepted. When aggregating sharings, as prescribed in the **Processing** step of BlinderProtocol, the resulting sharings are no longer inconsistent, because there will be $2t$ incorrect shares, which means that the sharings are non-reconstructible any more. An alternative would be to reject the client's message for the reason of $\langle \alpha \rangle$ and $\langle \beta \rangle$ not being perfectly consistent. But this, in turn may lead to a censorship attack, i.e., malicious servers can cheat in the shares they use in the verification check, by that they block a client. As discussed in the introduction, this is not desired.

To solve that dilemma, we use a standard technique for *robust input* [DN07, BH08]. In order to input a value x the client does not actually share x, rather, it obtains shares of a random value, $[r]$, from the servers, reconstruct it to obtain r in the clear and then broadcast $x' = x - r$ to all servers. By seeing x' the servers know nothing about x because it is masked by a random value. Then the servers use the value $[r] + x' = [r + x'] = [r + x - r] = [x]$ as the client's shared input.

6 Implementation & Evaluation

We have developed the networking, offline and online phases of Blinder including the generation of random double sharing, the format verification check, processing of blinded messages and final message openings (along with solving quadratic equations over finite field, which is required for extracting colliding messages from the matrices). Our implementation is written mainly in C++, while the generation of blinded messages and client-server networking are written in Go.

We implemented and measured two variants: a variant that is based on CPU only and a variant based on GPU. Both variants use only commodity machines. For our CPU version we used m5.12xlarge machines with with 192GiB, up to 3.1 GHz Intel Xeon Platinum 8175 processors and AVX-512; all CPU experiments were performed with either 1,4 or 8 threads. For our GPU version we used p3.16xlarge machines with

488GiB, Intel Xeon E5 processor. The machines are equipped with three NVIDIA Tesla V100 GPU that has 5120 Cuda cores that support an overall of up to 7.8 TFLOPS of double precision. The GPU code is based on the Cuda library and NVIDIA's cutlass [cut14]. We note that the benefits of a GPU do not necessarily translate to other solutions, in particular, we find a low potential to improve Riposte via a GPU, see note in Appendix A.

The high-level architecture of Blinder divides time to 'epochs' such that N clients interact with Blinder in each epoch. In the Tth epoch the servers run a 'server' process that waits for N clients to submit their blinded messages. The servers store each client's submission in a separate file (using a unique UUID chosen by the client). In the meantime, the servers perform the de-compression of the matrices, as described in Sections 4.1.2-4.1.3, for all messages submitted in the $T-1$-th epoch, by reading the relevant files from storage, ordering them by the UUID so that servers will be synced on the identity of blinded message they are working on.

We extensively measured the performance of Blinder. In our experiments we were mainly concerned with the latency and the cost. By latency we refer to the time a client has to wait from the moment of submitting a blind message to the moment of publication of its message. We measure latency using either CPU or GPU. In addition we are interested in measuring parallelism by using a varying number of threads and on varying network settings of LAN and WAN. Our experiments involve a varying number of servers (n from 5 to 100), clients (N from 10^3 to 10^6) and message length (L from $32B$ to $10KB$) in order to suit the applications presented in the introduction. By cost we refer to the monetary cost over a cloud computing service. The interesting number is the cost of Blinder per server per epoch, divided by N in order to normalize the result.

In addition to an extensive evaluation, we also present a comparison to leading works on anonymous broadcast: Riposte [CBM15] – a system that guarantees neither fairness nor robustness and does not offer a mechanism for resisting censorship by a malicious server; and {Power,Asynchro}Mix [LYK+19] that guarantees fairness (robustness in the online phase) and is censorship resistant.

Field type. The underlying field $\mathbb{F}_p$ we chose is the Mersenne field modulo the prime $p = 2^{31}-1$. We use this field since modular multiplication does not require performing divisions, even when the product is greater than the prime. We pick the field to be 31 bits as it best fits registers of both the CPU and GPU.

Matrix dimensions. The blinded message matrix M, as well as the aggregated matrix A are each of size $2.7 \cdot N \cdot L$. Note however, that the size of the vectors of the blinded message affects the overall blinded messages size, which forms the client to server communication complexity that we want to minimize. When both rows and cols are of size $\sqrt{2.7 \cdot N \cdot L}$ this is minimized.

Microbenchmark. Our aim is to break down the overall latency of Blinder across varying message sizes and number of clients, to the high level sub-tasks. In particular, we measure the times for (1) performing the ShiftLeft procedure in FormatVerification,

(2) obtaining α, β, γ in FormatVerification, (3) de-compressing the blind messages, (4) reconstructing all entries and finding the quadratic equations and (5) solving those equations. In addition, we measure the time it takes for N clients to submit their messages to n servers. The times are presented in Fig. 1. It is important to note that in Fig. 1 the y-axis is log scaled, in order to be able to express even short sub-tasks, so the lower parts of the bars are actually much shorter than the upper ones.

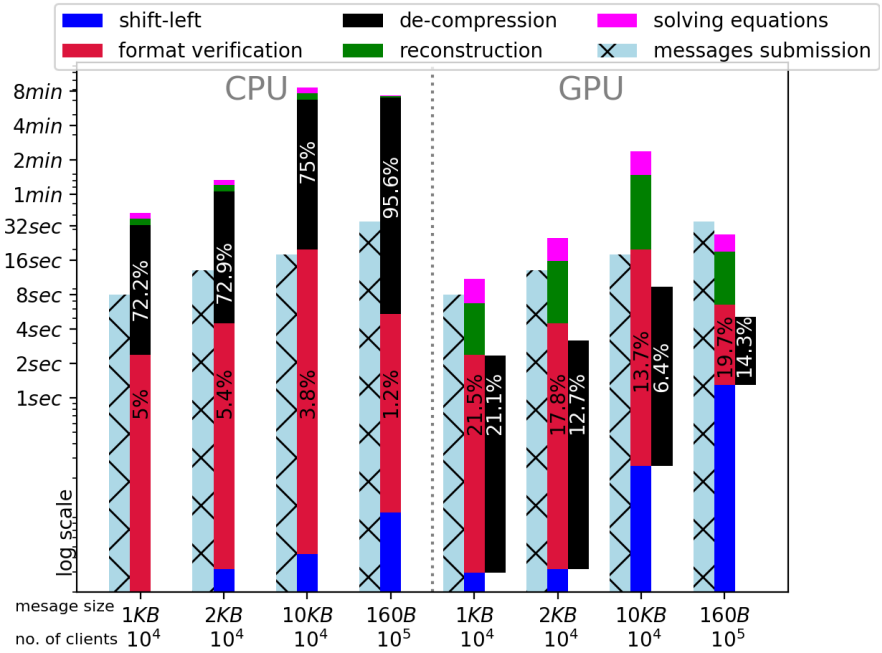

Figure 1: A breakdown of wall clock time of Blinder's sub-tasks.

Measurements were taken in a setting with 5 servers, all in the AWS N.Virginia data center. Since the messages received at the servers at the $T-1$-th epoch are used in the T-th epoch, it is preferable that the time it takes for all clients to submit their messages is less than the time it takes to perform all other tasks. As shown in the figure, this is indeed what happens, meaning that our write request submission phase is fast enough. We highlight two sub-tasks in the figure: the format verification time and the write query de-compression time. The de-compression time involves $O(L \cdot N^2)$ multiplications where L is the message size (number of field elements). Thus, it is expected to take a significant part of the overall time. Indeed, in the CPU-only version it takes more than 70% of the time. However, when this task is performed by the GPU in parallel to the format verification sub-task, it is no longer the bottleneck and takes only about $7-22\%$ of the time – a dramatic improvement.

Evaluation and comparison. We now present an evaluation of Blinder along with comparison to Riposte [CBM15], AsynchroMix and PowerMix [LYK+19]. We ran Riposte on the same machines and network settings as Blinder. As for AsynchroMix

and PowerMix, we used times reported in their paper, since their recommended setup is over Docker, which would not constitute a fair comparison to ours anyway, since we run Blinder and Riposte directly on the machines.

Scalability. We separate the scalability evaluation to two. First, we show that Blinder can scale to large number of servers, n, and large number of clients, N. To do so, we execute it over up to 100 servers, and configure the servers to accept up to 1 million messages. Second, we show that Blinder can scale to support large message size. This is necessary for anonymous P2P payment systems like Zcash and Monero (as mentioned in the Introduction). To this end, we configure the system to accept messages with length ranging from 32 bytes up to 10 kilobytes.

We start by showing the scalability of the number of clients and servers. The resulting latencies are presented in Fig. 2, note that the figure is log-scaled in both axes

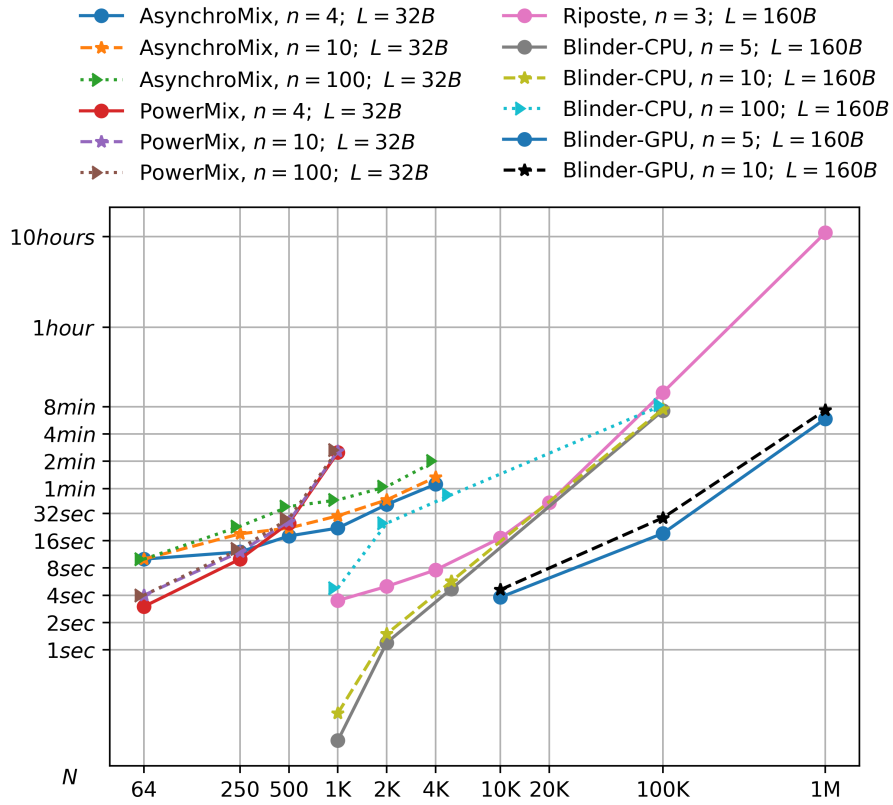

Figure 2: Latency of Blinder (CPU and GPU versions), Riposte, AsynchroMix and PowerMix when scaling number of clients and servers.

We recognize the fact that AsynchroMix and PowerMix are benchmarked under weaker machines than those we used for Blinder and Riposte, yet we estimate their performance to improve by no more than one order of magnitude.

Fig. 2 can be clustered to three parts: the top-left part includes plots of AsynchroMix and PowerMix, which cannot support a large number of clients, and therefore are

evaluated with up to 4096 clients. On the other hand, these protocols scale well with the number of servers and we managed to run with up to 100 servers. The center part of the figure shows plots of the CPU implementation of Blinder and Riposte, showing that extending the DPF approach to more than 2 servers is indeed practical, and in some cases even faster than the 2-servers setting of Riposte (e.g. running Blinder-CPU with 5 and 10 servers is faster than Riposte). Furthermore, just like [LYK+19], Blinder can run efficiently by up to 100 servers. Finally, the right-most part of the figure shows Blinder-GPU version with 5 and 10 servers, respectively. **The figure shows that with 10^5 clients, Blinder-GPU is $36\times$ faster, and with 10^6 clients it is more than $100\times$ faster than Riposte** (we remark that due to time limitation we did not run Riposte with 10^6 clients, and the comparison is based on what reported in [KCDF17], i.e. that it takes more than 11 hours).

Let us turn to discuss the scalability of the message length. AsynchroMix and PowerMix were not implemented with messages larger than 32 bytes and therefore these are omitted from the following comparison. We evaluate Blinder and Riposte with messages of length up to 10 and 5 kilobytes, respectively. The resulting latencies are found in Fig. 3. Let us examine the suitability of Blinder to a system like Zcash.

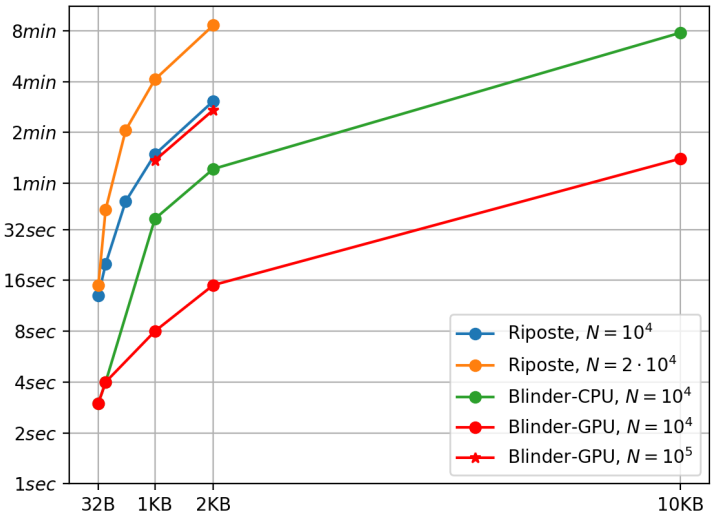

Figure 3: Latency of Blinder (CPU and GPU versions) and Riposte with varying message sizes.

The block rate is 2.5 minutes and the rate of shielded transactions is 6 per second [Bit20], or 150 per block. A standard shielded transaction (including a zk SNARK) is of 1-2 kilobytes [BCG+14] and the current number of users is much less than 10 thousands. The figure shows that supporting 10 thousands users within a block rate is possible already with the CPU variant, whereas the GPU variant can support even hundreds of thousands of users in that rate. Furthermore, considering more

complex and heavyweight transactions, the GPU variant can support transactions of size 10 kilobytes in the required rate. In contrast, Riposte could support 2 kilobytes transactions with that rate, but for up to 10 thousands users.

Multi-core utilization. To evaluate how well the CPU-based Blinder with 5 servers in N.Virginia, and Riposte with 3 servers utilize hardware, we ran them with different numbers of clients, N, and different message lengths, L. We ran both of them with 1 and 8 threads (and Blinder also with 4 threads). The run times appear in Table 2. The aim is to present the speedup of those systems when deployed with multi-core machines. As shown in the table, the third row of Riposte and second row of Blinder (in bold) have exactly the same setting ($N = 10^5$ and $L = 160B$), their speed ups when run with 8 threads are close (4.6x for Riposte vs 4.17x for Blinder), however Blinder is 1.51-1.67x faster. We note that utilizing X threads typically does not speed up a software by factor X, as there are management overheads and idle times when waiting for the slowest thread.

			number of threads				
	N	L	1	4	speed up	8	speed up
Riposte	10K	160B	69	-	-	17	4.05x
	50K	160B	896	-	-	200	4.48x
	100K	**160B**	**3162**	**-**	**-**	**686**	**4.60x**
Blinder	100K	32B	295	115	2.56x	82	3.59x
	100K	**160B**	**1887**	**589**	**3.20x**	**452**	**4.17x**

Table 2: Multi core utilization of Blinder and Riposte.

Effect of bandwidth. In this experiment we evaluate how Blinder scales over different network settings: LAN, WAN1 and WAN2. Our hypothesis is that since the bottleneck of the protocol is the task of de-compressing the blind messages, then 'slowing down' the network would not have much impact. In the LAN setting all machines are deployed at N.Virginia, US, with 0.287 RTT and a bandwidth of 4.97Gb/sec. In the WAN1 setting machines are separated between N.Virginia, US and Oregon, US, with 75.9 RTT and a bandwith of 171 Mbits/sec. In the WAN2 setting machines are separated between N.Virginia, US and Sydney, Australia, with 203.54 RTT and a bandwith of 58.1 Mbits/sec. We measured Blinder's CPU version with $N = 10^5$ clients. The number of servers increases from 5 to 10 to 20. The lower part of the figure shows the latency when the message length $L = 32B$, and the upper part of the figures shows this when $L = 160B$. The figure proves that our hypothesis is correct. Over a fast network (LAN) and $L = 32B$ the difference in latency is minimal, i.e. increasing n from 5 to 10 and from 10 to 20 increases latency by only 3% and 1.2% respectively. Over a slower network (WAN2), on the other hand, this increases latency by 55% and 32%, respectively. We also plot the latency of Riposte with 3 servers, $L = 160B$, over LAN. Riposte's latency over WAN is out of the border of this figure, due to the fact that their protocol requires much more communication in order to verify the format of the messages.

Monetary cost. Fig. 5 presents the estimated monetary cost, in US cents, over Amazon AWS, for both CPU- and GPU-based Blinder, supporting message size of $32B$ and $160B$. This is calculated according to the CPU and GPU machine cost per hour, which are 2.304\$ and 24.48\$, respectively. The figure shows the cost per a single

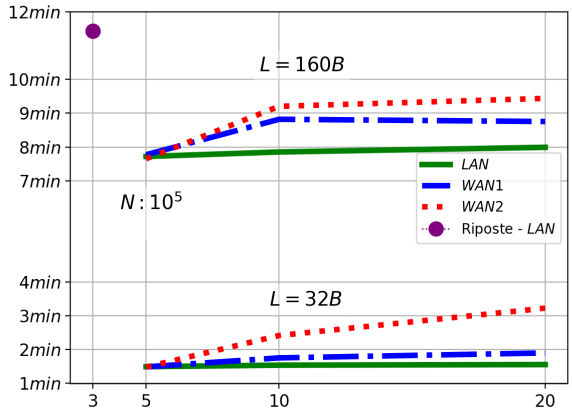

Figure 4: Latency of Blinder (CPU and GPU versions) and Riposte over LAN, WAN1 and WAN2 settings (described in text).

epoch, single server and divided by the anonymity set size (hence, the cost per client). To obtain the total epoch price, one has to multiply the reported cost by $n \cdot N$.

We note that, counter-intuitively, the cost per client when the anonymity set size is $N = 10^5$ is lower than when the anonymity set size is $N = 10^4$. This is because our GPU-based implementation is optimized for larger workloads (e.g. for $N \geq 10^5$).

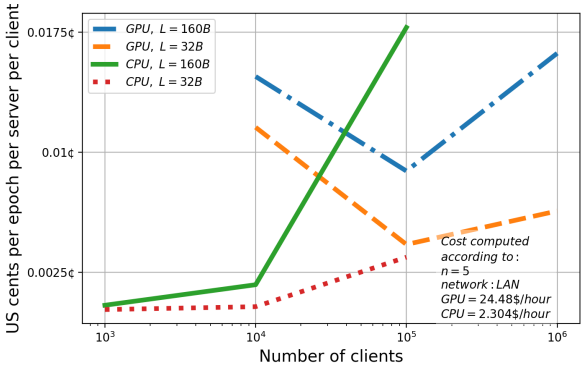

Figure 5: Monetary cost of Blinder.

7 Conclusions

This work addresses the question of whether we can design a system for anonymous committed broadcast over a synchronous network, which is resilient to a malicious

adversary controlling servers and clients, prevents a malicious server from censoring honest clients, and is scalable in all metrics: the number of servers, number of clients and the message size. We answer this question in the affirmative and present the first system, called Blinder, that has all these properties. Blinder confirms our hypothesis that an information theoretically secure protocol performs better than a protocol that relies on computational assumptions. We come to this conclusion since the 'amount' of work in Blinder and in Riposte [CBM15] is very close, whereas the type of work that is done in both systems is different: in Blinder the work is dominated by simple finite field arithmetic, and in Riposte the work is dominated by AES. Even though AES is implemented in Riposte by a hardware instruction, Blinder performs much better in both the CPU and GPU versions.

Acknowledgements. This work has been partially funded by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Ministers Office, and by a grant from the Israel Science Foundation. We thank Udi Wieder, Meital Levi, Moriya Farbstein, Lior Koskas, Shahar Zadok, Assi Barak and Oren Tropp for valuable discussion and their contribution to the implementation and the experiments.

References

[ADFM17] Ahmed A. Abdelrahman, Hisham Dahshan, Mohamed M. Fouad, and Ahmed M. Mousa. High performance cuda aes implementation: A quantitative performance analysis approach. In *Computing Conference*, page 1, 2017.

[AKTZ17] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX*, pages 1217–1234, 2017.

[AS16] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *USENIX*, pages 551–569, 2016.

[Bal04] J.M. Balkin. Digital speech and democratic culture: A theory of freedom of expression for the information society. *New York University Law Review*, 79:1–58, 04 2004.

[BBC+20] Dan Boneh, Elette Boyle, Henry Corrigan Gibbs, Niv Gilboa, and Yuval Ishai. Private communication. 2020.

[BBGN19] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. The privacy blanket of the shuffle model. In *Advances in Cryptology - CRYPTO 2019, Part II*, pages 638–667, 2019.

[BCC+15] Stevens Le Blond, David R. Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. *Computer Communication Review*, 45(5):639–652, 2015.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE SP*, pages 459–474, 2014.

[BCZ+13] Stevens Le Blond, David R. Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *ACM SIGCOMM*, pages 303–314, 2013.

[BDG15] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A private presence service. *PoPETs*, 2015(2):4–24, 2015.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[BEM+17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, pages 441–459, 2017.

[BFK00] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web mixes: A system for anonymous and unobservable internet access. In *Workshop on Design Issues in Anonymity and Unobservability, Berkeley, July 25-26, 2000, Proceedings*, pages 115–129, 2000.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, pages 1292–1303, 2016.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, pages 213–230, 2008.

[BHKL18] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale P2P mpc-as-a-service and low-bandwidth MPC for weak participants. In *CCS*, pages 695–712, 2018.

[Bit20] BitDegree. *Zcash vs. Monero*, 2020.

[BL] Oliver Berthold and Heinrich Langos. Dummy traffic against long term intersection attacks. In *PET*.

[CB95] David A. Cooper and Kenneth P. Birman. Preserving privacy in a network of mobile computers. In *S&P*, pages 26–38, 1995.

[CBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *SOSP*, pages 321–338, 2015.

[CF10] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *ACMCCS*, pages 340–350, 2010.

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *ACM STOC*, pages 304–313, 1997.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, pages 34–64, 2018.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[CJK+16] David Chaum, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, and Alan T. Sherman. cmix: Anonymization byhigh-performance scalable mixing. *ePrint*, 2016:8, 2016.

[CSM+20] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas E. Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. *CoRR*, abs/2001.08250, 2020.

[CSU+19] Albert Cheu, Adam D. Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Advances in Cryptology - EUROCRYPT 2019, Part I*, pages 375–403, 2019.

[cut14] cutlass. Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2014.

[CZJJ12] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: website fingerprinting attacks and defenses. In *ACMCCS*, pages 605–616, 2012.

[DDM] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *(S&P*.

[DGK+19] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *CoRR*, abs/1904.05234, 2019.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX*, pages 303–320, 2004.

[DN07] Ivan Damgard and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[ECZB19] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. *CoRR*, abs/1911.09215, 2019.

[EFM+19] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, pages 2468–2479, 2019.

[Fou19] Zcash Foundation. *Zcash Privacy and Security Recommendation*, 2019.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658, 2014.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[GT96] Ceki Gülcü and Gene Tsudik. Mixing email with babel. In *1996 Symposium on Network and Distributed System Security, (S)NDSS '96, San Diego, CA, USA, February 22-23, 1996*, pages 2–16, 1996.

[GWF13] Henry Corrigan Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *USENIX*, pages 147–162, 2013.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji, and KatsuChidami Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, pages 202–216, 2012.

[HVC10] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Trans. Inf. Syst. Secur.*, 13(2):13:1–13:28, 2010.

[KAL+15] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *USENIX*, pages 287–302, 2015.

[KAPR06] Dogan Kesdogan, Dakshi Agrawal, Dang Vinh Pham, and Dieter Rautenbach. Fundamental limits on the anonymity provided by the MIX technique. In *S&P*, pages 86–99, 2006.

[KCDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, pages 406–422, 2017.

[KEB98] Dogan Kesdogan, Jan Egner, and Roland Büschkes. Stop-and-go-mixes providing probabilistic anonymity in an open system. In *Information Hiding*, pages 83–98, 1998.

[KLDF16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *PoPETs*, 2016(2):115–134, 2016.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *S&P*, pages 839–858, 2016.

[KOR+04] Lea Kissner, Alina Oprea, Michael K. Reiter, Dawn Xiaodong Song, and Ke Yang. Private keyword-based push and pull with applications to anonymous communication. In *ACNS*, pages 16–30, 2004.

[Kre01] Seth Kreimer. Technologies of protest: Insurgent social movements and the first amendment in the era of the internet. *University of Pennsylvania Law Review*, 150:119, 11 2001.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACMCCS*, pages 254–269, 2016.

[LRWW04] Brian Neil Levine, Michael K. Reiter, Chenxi Wang, and Matthew K. Wright. Timing attacks in low-latency mix systems (extended abstract). In *Financial Cryptography*, pages 251–265, 2004.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Rahul Mahadev, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronousmpc and its application to anonymous communication. *eprint archive Report 2019/883*, 2019.

[LZ16] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *USENIX*, pages 571–586, 2016.

[MD04] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *PET*, pages 17–34, 2004.

[MD05] Steven J. Murdoch and George Danezis. Low-cost traffic analysis of tor. In *IEEE (S&P 2005)*, pages 183–195, 2005.

[MOT+11] Prateek Mittal, Femi G. Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. Pir-tor: Scalable anonymous communication using private information retrieval. In *USENIX*, 2011.

[MWB13] Prateek Mittal, Matthew K. Wright, and Nikita Borisov. Pisces: Anonymous communication using social networks. In *NDSS*, pages 1–18, 2013.

[MZ07] Steven J. Murdoch and Piotr Zielinski. Sampled traffic analysis by internet-exchange-level adversaries. In *PET*, pages 167–183, 2007.

[NS03] Lan Nguyen and Reihaneh Safavi-Naini. Breaking and mending resilient mix-nets. In *PET*, pages 66–80, 2003.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303, 1997.

[Pfi94] Birgit Pfitzmann. Breaking efficient anonymous channel. In *EUROCRYPT*, pages 332–340, 1994.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security*, pages 1199–1216, 2017.

[PNZE11] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *ACMWPES*, pages 103–114, 2011.

[PP89] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct rsa-implementation of mixes. In *EUROCRYPT*, pages 373–381, 1989.

[Ray00] Jean-Francois Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *PET*, pages 10–29, 2000.

[RSG98] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, 1998.

[SAKD17] Fatemeh Shirazi, Elena Andreeva, Markulf Kohlweiss, and Claudia Díaz. Multiparty routing: Secure routing for mixnets. *CoRR*, abs/1708.03387, 2017.

[SCM05] Len Sassaman, Bram Cohen, and Nick Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES*, pages 1–9, 2005.

[SDS02] Andrei Serjantov, Roger Dingledine, and Paul F. Syverson. From a trickle to a flood: Active attacks on several mix types. In *Information Hiding,*, pages 36–52, 2002.

[SGRE04] Emin Gün Sirer, Sharad Goel, Mark Robson, and Dogan Engin. Eluding carnivores: file sharing with strong anonymity. In *Proceedings of the 11st ACM SIGOPS European Workshop, Leuven, Belgium, September 19-22, 2004*, page 19, 2004.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SL09] Alexandre Soro and Jérôme Lacan. Fnt-based reed-solomon erasure codes. *CoRR*, abs/0907.1788, 2009.

[SW06] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *ESORICS*, pages 18–33, 2006.

[TFKL99] Al Teich, Mark Frankel, Rob Kling, and Ya-Ching Lee. Anonymous communication policies for the internet: Results and recommendations of the aaas conference. *Inf. Soc.*, 15:71–77, 04 1999.

[vdHLZZ15] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152, 2015.

[Vol99] Eugene Volokh. Freedom of speech, information privacy, and the troubling implications of a right to stop people from speaking about you. *New York University Law Review*, 1999.

[vS13] Nicolas van Saberhagen. *Monero*, 2013.

[Wal19] Jonathan Wallac. Nameless in cyberspac e anonymity on the internet. *Cato Institute*, 09 2019.

[WCFJ12] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *USENIX OSDI*, pages 179–182, 2012.

[WG13] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *ACMWPES*, pages 201–212, 2013.

[Wik03] Douglas Wikström. Five practical attacks for "optimistic mixing for exit-polls". In *Selected Areas in Cryptography Workshop, SAC*, pages 160–175, 2003.

[WWB19] Yu Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and CPU platforms for deep learning. *CoRR*, abs/1907.10701, 2019.

A A Note on the Suitability of GPU to Riposte

The Nvidia v100 GPU we use is equivalent to GTX1080 in its integer operations throughput, which can potentially perform up to 2840 Giga int32 operations per second (GIOPS). GTX1080 can execute AES operations on a stream of data with a rate of 50-250 Gigabit per second, equivalent to 6-40 GB/s [ADFM17]. The GPU, as mentioned supports more than $300\times$ than the CPU (i.e. $2840 \times 32/250$). However, Blinder requires arithmetics over a finite field, rather than merely int32 operations. In our implementation we use Mersenne31 as our finite field (where field elements are represented using 31 bits), such that operations over the field are almost equivalent to about three int32 operations. This renders the potential of GPU to accelerate Blinder to be $\sim 100\times$

higher than its potential to accelerate Riposte. In fact, we find in our experiments that the GPU version of Blinder is about $100\times$ faster than Riposte, meaning that a GPU version of Riposte would not accelerate it.

B Notation

We present here a compilation of the various notations we use throughout the paper. Note that notation in the appendix may be different and even collide.

$[1, x]$	the set $\{1, \ldots, x\}$
$\mathcal{C}^k$	the k-th client
$\mathcal{S}_q$	the q-th server
N	number of supported messages/clients
ρ	the assumed fraction on number of honest clients
n	number of servers
t	number of corrupted servers
q	iterator for servers
k	iterator for clients
$[s]$	a sharing of secret s, meaning that server $\mathcal{S}_q$ holds the share s_q for $q \in [1, n]$.
$A_{(i,j)}$	entry at row i and column j in the bulletin board
$\mathsf{rows} \times \mathsf{cols}$	dimensions of A: rows rows and cols columns
$M^k_{(\cdot,\cdot)}$	the blinded message prepared by $\mathcal{C}^k$ in the naive implementation of Blinder.
$M^k_{(\cdot,\cdot),q}$	the shares of $M^k_{(\cdot,\cdot)}$ held by $\mathcal{S}_q$
$\boldsymbol{r}$	a row vector (of size cols)
$\boldsymbol{c}$	a column vector (of size rows)
DoS	Denial of Service