Anonymity Preserving Byzantine Vector Consensus

Christian Cachin[1], Daniel Collins[2,3](✉), Tyler Crain[2], and Vincent Gramoli[2,3]

[1] University of Bern, Bern, Switzerland
cachin@inf.unibe.ch
[2] University of Sydney, Sydney, Australia
tycrain@gmail.com, vincent.gramoli@sydney.edu.au
[3] EPFL, Lausanne, Switzerland
daniel.collins@epfl.ch

Abstract. Collecting anonymous opinions has applications from whistleblowing to complex voting, where participants rank candidates by order of preferences. Unfortunately, as far as we know there is no efficient distributed solution to this problem. Previous solutions either require trusted third parties, are inefficient or sacrifice anonymity.
In this paper, we propose a distributed solution called the *Anonymised Vector Consensus Protocol* (AVCP) that reduces the problem of agreeing on a set of anonymous votes to the binary Byzantine consensus problem. The key idea to preserve anonymity of voters—despite some of them acting maliciously—is to detect double votes through traceable ring signatures. AVCP is resilient-optimal as it tolerates up to a third of Byzantine participants. We prove our algorithm correct and show that it preserves anonymity with at most a linear communication overhead and constant message overhead when compared to a recent consensus baseline. Finally, we demonstrate empirically that the protocol is practical by deploying it on 100 machines geo-distributed in three continents: America, Asia and Europe. Anonymous decisions are reached within 10 seconds with a conservative choice of traceable ring signatures.

1 Introduction and related work

Consider a distributed survey where a group of mutually distrusting participants wish to exchange their opinions about some issue. For example, participants may wish to communicate over the Internet to rank candidates in order of preference to change the governance of a blockchain. Without making additional trust assumptions [1, 19, 39], one promising approach is to run a Byzantine consensus algorithm [41], or more generally a vector consensus algorithm [17, 24, 47] to allow for arbitrary votes. In vector consensus, a set of participants decide on a common vector of values, each value being proposed by one process. Unlike interactive consistency [41], a protocol solving vector consensus can be executed without fully synchronous communication channels, and as such is preferable for use over the Internet. Unfortunately, vector consensus protocols tie each participant's opinion to its identity to ensure one opinion is not overrepresented in

the decision and to avoid double voting. There is thus an inherent difficulty in solving vector consensus while preserving anonymity.

In this paper, we introduce the *anonymity-preserving vector consensus* problem that prevents an adversary from discovering the identity of non-faulty participants that propose values in the consensus and we introduce a solution called Anonymised Vector Consensus Protocol (AVCP). To prevent the leader in some Byzantine consensus algorithms [13] from influencing the outcome of the vote by discarding proposals, AVCP reduces the problem to binary consensus that is executed without the need for a traditional leader [18].

We provide a mechanism to prevent Byzantine processes from double voting whilst also decoupling their ballots from their identity. In particular, we adopt *traceable ring signatures* [30,36], which enable participants to anonymously prove their membership in a set of participants, exposing a signer's identity if and only if they sign two different votes. This can disincentivise participants from proposing multiple votes to the consensus. Alternatively, we could use linkable ring signatures [42], but would not ensure that Byzantine processes are held accountable when double-signing. We could also have used blind signatures [12, 14], but this would have required an additional trusted authority.

We also identify interesting conditions to ensure anonymous communication. Importantly, participants must propagate their signatures anonymously to hide their identity. To this end, we could construct anonymous channels directly. However, these protocols either require additional trusted parties, are not robust or incur $O(n)$ message delays to terminate [15,33,34,38]. Thus, we assume access to anonymous channels, such as through publicly-deployed [23, 58] networks which often require sustained network observation or large amounts of resources to deanonymise with high probability [32,45]. Anonymity is ensured then as processes do not reveal their identity via ring signatures and communicate over anonymous channels. If correlation-based attacks on certain anonymous networks are deemed viable [43] and for efficiency's sake, processes can anonymously broadcast their proposal and then continue the protocol execution over regular channels. However, anonymous channels alone cannot ensure anonymity when combined with regular channels as an adversary may infer identities through latency [2] and message transmission ordering [49]. For example, an adversary may relate late message arrivals from a single process over both channels to deduce the identity of a slow participant. This is why, to ensure anonymity, the timing and order of messages exchanged over anonymous channels should be statistically independent (or computationally indistinguishable with a computational adversary) from that of those exchanged over regular channels. In practice, one may attempt to ensure that there is a low correlation by ensuring that message delays over anonymous and regular channels are sufficiently randomised.

We construct our solution iteratively first by defining the *anonymity-preserving all-to-all reliable broadcast problem* that may be of independent interest. Here, anonymity and comparable properties to reliable broadcast [7] with n broadcasters are ensured. By constructing a solution to this problem with Bracha's classical reliable broadcast [5], AVCP can terminate after three regular

and one anonymous message delay. With this approach, our experimental results are promising—with 100 geo-distributed nodes, AVCP generally terminates in less than ten seconds. We remark that, to ensure confidentiality of proposals until after termination, threshold encryption [21, 53], in which a minimum set of participants must cooperate to decrypt any message, can be used at the cost of an additional message delay.

Related constructions. We consider techniques without additional trusted parties. Homomorphic tallying [19] encodes opinions as values that are summed in encrypted form prior to decryption. Such schemes that rely on a public bulletin board for posting votes [35] could use Byzantine consensus [41] and make timing assumptions on vote submission to perform an election. Unfortunately, homomorphic tallying is impractical when the pool of candidates is large and impossible when arbitrary. Nevertheless, using a multiplicative homomorphic scheme [26] requires exponential work in the amount of candidates at tallying time, and additive homomorphic encryption like Paillier's scheme [48] incurs large (RSA-size) parameters and more costly operations. Fully homomorphic encryption [8,31] is suitable in theory, but at present is untenable for computing complex circuits efficiently. Self-tallying schemes [35], which use homomorphic tallying, are not appropriate as at some point all participants must be correct, which is untenable with arbitrary (Byzantine) behaviour. Constructions involving mix-nets [16] allow for arbitrary ballot structure. However, decryption mix-nets are not a priori robust to a single Byzantine or crash failure [16], and re-encryption mix-nets which use proofs of shuffle are generally slow in tallying [1]. With arbitrary encryptions, larger elections can take minutes to hours [40] to tally. Even with precomputation, $O(t)$ processes still need to perform proofs of shuffle [46] in sequence, which incurs untenable latency even without process faults. With faults and/or asynchrony, we could run $O(t)$ consensus instances in sequence but at an additional cost. DC-nets and subsequent variations [15, 34] are not sufficiently robust and generally require $O(n)$ message delays for an all-to-all broadcast. On multi-party computation, general techniques like Yao's garbled-circuits [56] incur untenable overhead given enough complexity in the structure of ballots. Private-set intersection [28, 57] can be efficient for elections that require unanimous agreement, but do not generalise arbitrarily.

Roadmap. The paper is structured as follows. Section 2 provides preliminary definitions and specifies the model. Section 3 presents protocols and definitions required for our consensus protocol. Section 4 presents our anonymity-preserving vector consensus solution. Section 5 benchmarks AVCP on up to 100 geo-distributed located on three continents. To conclude, Section 6 discusses the use of anonymous and regular channels in practice and future work.

2 Model

We assume the existence of a set of processes $P = \{p_1, \ldots, p_n\}$ (where $|P| = n$, and the ith process is p_i), an adversary A who corrupts $t < \frac{n}{3}$ processes in

P, and a trusted dealer D who generates the initial state of each process. For simplicity of exposition, we assume that cryptographic primitives are unbreakable. With concrete primitives that provide computational guarantees, each party could be modelled as being able to execute a number of instructions each message step bounded by a polynomial in a security parameter k [11]. In this case, the transformation of our proofs presented in the companion technical report [9] is straight forward given the hardness assumptions required by the underlying cryptographic schemes.

2.1 Network

We assume that P consists of asynchronous, sequential processes that communicate over reliable, point-to-point channels in an asynchronous network. An *asynchronous* process is one that executes instructions at its own pace. An *asynchronous* network is one where message delays are unbounded. A *reliable* network is such that any message sent by a correct process will eventually be delivered by the intended recipient. We assume that processes can also communicate using reliable one-way *anonymous* channels, which we soon describe.

Each process is equipped with the primitive "send M to p_j", which sends the message M (possibly a tuple) to process $p_j \in P$. For simplicity, we assume that a process can send a message to itself. A process receives a message M by invoking the primitive "receive M". Each process may invoke "broadcast M", which is short-hand for "**for each** $p_i \in P$ **do** send M to p_i **end for**". Analogously, processes may invoke "anon_send M to p_j" and "anon_broadcast M" over anonymous channels.

Since reaching consensus deterministically is impossible in failure-prone asynchronous message-passing systems [27], we assume that *partial synchrony* holds among processes in P in Section 4. That is, we assume there exists a point in time in protocol execution, the global stabilisation time (GST), unknown to all processes, after which the speed of each process and all message transfer delays are upper bounded by a finite constant [25].

2.2 Adversary

We assume that the adversary A schedules message delivery over the regular channels, restricted to the assumptions of our model (e.g., the reliability of the channels). For each send call made, A determines when the corresponding receive call is invoked. A portion of processes—exactly $t < \frac{n}{3}$ members of P—are initially corrupted by A and may exhibit Byzantine faults [41] over the lifetime of a protocol's execution. That is, they may deviate from the predefined protocol in an arbitrary way. We assume A can see all computations and messages sent and received by *corrupted* processes. A *non-faulty* process is one that is not corrupted by A and therefore follows the prescribed protocol. A can only observe anon_send and anon_receive calls made by corrupted processes.

A cannot see the (local) computations that non-faulty processes perform. We do not restrict the amount or speed of computation that A can perform.

2.3 Anonymity assumption

Consider the following experiment. Suppose that $p_i \in P$ is non-faulty and invokes "anon_send m to p_j", where $p_j \in P$ is possibly corrupted, and p_j invokes anon_receive with respect to m. No process can directly invoke send or invoke receive in response to a send call at any time. p_j is allowed to use anon_send to message corrupted processes if it is corrupted, and can invoke anon_recv with respect to messages sent by corrupted processes. Each process is unable to make oracle calls (described below), but is allowed to perform an arbitrary number of local computations. p_j then outputs a single guess, $g \in \{1, \ldots, n\}$ as to the identity of p_i. Then for any run of the experiment, $Pr(i = g) \leq \frac{1}{n-t}$.

As A can corrupt t processes, the anonymity set [20], i.e., the set of processes p_i is indistinguishable from, comprises $n - t$ non-faulty processes. Our definition captures the anonymity of the anonymous channels, but does not consider the effects of regular message passing and timing on anonymity. As such, we can use techniques to establish anonymous channels in practice with varying levels of anonymity with these factors considered.

2.4 Traceable ring signatures

Informally, a ring signature [30,51] proves that a signer has knowledge of a private key corresponding to one of many public keys of their choice without revealing the corresponding public key. Hereafter, we consider *traceable ring signatures* (or *TRSs*), which are ring signatures that expose the identity of a signer who signs two different messages. To increase flexibility, we can consider traceability with respect to a particular string called an *issue* [55], allowing signers to maintain anonymity if they sign multiple times, provided they do so each time with respect to a different issue.

We now present relevant definitions of the ring signatures which are analogous to those of Fujisaki and Suzuki [30]. Let $ID \in \{0,1\}^*$, which we denote as a *tag*. We assume that all processes may query an idealised distributed oracle, which implements the following *four* operations:

1. $\sigma \leftarrow \mathsf{Sign}(i, ID, m)$, which takes the integer $i \in \{1, \ldots, n\}$, tag $ID \in \{0,1\}^*$ and message $m \in \{0,1\}^*$, and outputs the signature $\sigma \in \{0,1\}^*$. We restrict Sign such that only process $p_i \in P$ may invoke Sign with first argument i.
2. $b \leftarrow \mathsf{VerifySig}(ID, m, \sigma)$, which takes the tag ID, message $m \in \{0,1\}^*$, and signature $\sigma \in \{0,1\}^*$, and outputs a bit $b \in \{0,1\}$. All parties may query VerifySig.
3. $out \leftarrow \mathsf{Trace}(ID, m, \sigma, m', \sigma')$, which takes the tag $ID \in \{0,1\}^*$, messages $m, m' \in \{0,1\}^*$ and signatures $\sigma, \sigma' \in \{0,1\}^*$, and outputs $out \in \{0,1\}^* \cup$

$\{1, \ldots, n\}$ (possibly corresponding to a process p_i). All parties may query Trace.

4. $x \leftarrow \mathsf{FindIndex}(ID, m, \sigma)$ takes a tag $ID \in \{0,1\}^*$, a message $m \in \{0,1\}^*$, and a signature $\sigma \in \{0,1\}^*$, and outputs a value $x \in \{1, \ldots, n\}$. FindIndex may not be called by any party, and exists only for protocol definitions.

The distributed oracle satisfies the following relations:

- $\mathsf{VerifySig}(ID, m, \sigma) = 1 \iff \exists\ p_i \in P$ which invoked $\sigma \leftarrow \mathsf{Sign}(i, ID, m)$.
- Trace is as below $\iff \sigma \leftarrow \mathsf{Sign}(i, ID, m)$ and $\sigma' \leftarrow \mathsf{Sign}(i', ID, m')$ where:

$$\mathsf{Trace}(ID, m, \sigma, m', \sigma') = \begin{cases} \text{“indep”} & \text{if } i \neq i', \\ \text{“linked”} & \text{else if } m = m', \\ i & \text{otherwise } (i = i' \wedge m \neq m'). \end{cases}$$

- If adversary D is given an arbitrary set of signatures S and must identify the signer p_i of a signature $\sigma \in S$ by guessing i, $Pr(i = g) \leq \frac{1}{n-t}$ for any D.

The concrete scheme proposed by Fujisaki and Suzuki [30] computationally satisfies these properties in the random oracle model [3] provided the Decisional Diffie-Hellman problem is intractable. In our protocols, where the ring comprises the n processes of P, the resulting signatures are of size $O(kn)$, where k is the security parameter. To simplify the presentation, we assume that its properties hold unconditionally in the following.

3 Communication primitives

3.1 Traceable broadcast

Suppose a process p wishes to anonymously send a message to a given set of processes P. By invoking anonymous communication channels, p can achieve this, but processes in $P \setminus \{p\}$ are unable to verify that p resides in P, and so cannot meaningfully participate in the protocol execution. By using (traceable) ring signatures, p can verify its membership in P over anonymous channels without revealing its identity. To this end, we outline a simple mechanism to replace the invocation of send and receive primitives (over regular channels) with calls to ring signature and anonymous messaging primitives (namely anon_send and anon_receive).

Let $(ID, TAG, \mathsf{label}, m)$ be a tuple for which $p_i \in P$ has invoked send with respect to. Instead of invoking send, p_i first calls $\sigma \leftarrow \mathsf{Sign}(i, T, m)$, where T is a tag uniquely defined by TAG and label. Then, p_i invokes anon_send M, where $M = (ID, TAG, \mathsf{label}, m, \sigma)$. Upon an anon_receive M call by $p_j \in P$, p_j verifies that $\mathsf{VerifySig}(T, m, \sigma) = 1$ and that they have not previously received (m', σ') such that $\mathsf{Trace}(T, m, \sigma, m', \sigma') \neq$ “indep”. Given this check passes, p_j invokes receive $(ID, TAG, \mathsf{label}, m)$. In our protocols, processes always broadcast messages, so the transformation is used with respect to broadcast calls.

By the properties of the anonymous channels and the signatures, it follows that anonymity as defined in the previous section holds with additional adversarial access to the distributed oracles. We present the proof in the companion technical report [9]. Hereafter, we assume that calls to primitives send and receive are handled by the procedure presented in this subsection unless explicitly stated otherwise.

3.2 Binary consensus

Broadly, the binary consensus problem involves a set of processes reaching an agreement on a binary value $b \in \{0, 1\}$. We first recall the definitions that define the binary Byzantine consensus (BBC) problem as stated in [18]. In the following, we assume that every non-faulty process proposes a value, and remark that only the values in the set $\{0, 1\}$ can be decided by a non-faulty process.

1. **BBC-Termination:** Every non-faulty process eventually decides on a value.
2. **BBC-Agreement:** No two non-faulty processes decide on different values.
3. **BBC-Validity:** If all non-faulty processes propose the same value, no other value can be decided.

For simplicity, we present the safe, non-terminating variant of the binary consensus routine from [18] in Algorithm 1. As assumed in the model (Section 2), the terminating variant relies on partial synchrony between processes in P. The protocols execute in asynchronous rounds.

State. A process keeps track of a binary value $est \in \{0, 1\}$, corresponding to a process' current estimate of the decided value, arrays $bin\_values[1..]$, in which each element is a set $S \subseteq \{0, 1\}$, a round number r (initialised to 0), an auxiliary binary value b, and lists of (binary) values $values_r$, $r = 1, 2, \ldots$, each of which are initially empty.

Messages. Messages of the form (EST, r, b) and (AUX, r, b), where $r \geq 1$ and $b \in \{0, 1\}$, are sent and processed by non-faulty processes. Note that we have omitted the dependency on a label *label* and identifier *ID* for simplicity of exposition.

BV-broadcast. To exchange EST messages, the protocol relies on an all-to-all communication abstraction, BV-broadcast [44], which is presented in Algorithm 1. When a process adds a value $v \in \{0, 1\}$ to its array $bin\_values[r]$ for some $r \geq 1$, we say that v was BV-delivered.

Functions. Let $b \in \{0, 1\}$. In addition to BV-broadcast and the communication primitives in our model, a process can invoke bin_propose(b) to begin executing an instance of binary consensus with input b, and decide(b) to decide the value b. In a given instance of binary consensus, these two functions may be called exactly once. In addition, the function $list$.append(v) appends the value v to the list $list$.

Algorithm 1 Safe binary consensus routine

```
1: bin_propose(v):
2:   est ← v; r ← 0;
3:   repeat:
4:     r ← r + 1;
5:     BV-broadcast(EST, r, est)
6:     wait until (bin_values[r] ≠ ∅)
7:     broadcast (AUX, r, bin_values[r])
8:     wait until (|values_r| ≥ n − t) ∧ (val ∈ bin_values[r] for all val ∈ values_r)
9:     b ← r (mod 2)
10:    if val = w for all val ∈ values_r where w ∈ {0, 1} then
11:      est ← w;
12:      if w = b then
13:        decide(v) if not yet invoked decide()
14:    else
15:      est ← b
16: upon intial receipt of (AUX, r, b) for some b ∈ {0, 1} from process p_j
17:   values_r.append(b)

18: BV-broadcast(EST, r, v_i):
19:   broadcast (EST, r, v_i)
20: upon receipt of (EST, r, v)
21:   if (EST, r, v) received from (t + 1) processes and not yet broadcast then
22:     broadcast (EST, r, v)
23:   if (EST, r, v) received from (2t + 1) processes then
24:     bin_values[r] ← bin_values[r] ∪ {v}
```

To summarise Algorithm 1, the BV-broadcast component ensures that only a value proposed by a correct process may be decided, and the auxiliary broadcast component ensures that enough processes have received a potentially decidable value to ensure agreement. The interested reader can verify the correctness of the protocol and read a thorough description of how it operates in [18], where the details of the corresponding terminating protocol also reside.

3.3 Anonymity-preserving all-to-all reliable broadcast

To reach eventual agreement in the presence of Byzantine processes without revealing who proposes what, we introduce the *anonymity-preserving all-to-all reliable broadcast* problem that preserves the anonymity of each honest sender which is reliably broadcasting. In this primitive, all processes are assumed to (anonymously) broadcast a message, and all processes deliver messages over time. It ensures that all honest processes always receive the same message from one (possibly faulty) sender while hiding the identity of any non-faulty sender.

Let $ID \in \{0,1\}^*$ be an *identifier*, a string that uniquely identifies an instance of anonymity-preserving all-to-all reliable broadcast, hereafter referred

to as AARB-broadcast. Let m be a message, and σ be the output of the call $\mathsf{Sign}(i, T, m)$ for some $i \in \{1, \ldots, n\}$, where $T = f(ID, \mathsf{label})$ for some function f as in traceable broadcast. Each process is equipped with two operations, "AARBP" and "AARB-deliver". $\mathsf{AARBP}[ID](m)$ is invoked once with respect to ID and any message m, denoting the beginning of a process' execution of AARBP with respect to ID. $\mathsf{AARB\text{-}deliver}[ID](m, \sigma)$ is invoked between $n-t$ and n times over the protocol's execution. When a process invokes $\mathsf{AARB\text{-}deliver}[ID](m, \sigma)$, they are said to "AARB-deliver" (m, σ) with respect to ID. Then, given $t < \frac{n}{3}$, we define a protocol that implements AARB-broadcast with respect to ID as satisfying the following *six* properties:

1. **AARB-Signing:** If a non-faulty process p_i AARB-delivers a message with respect to ID, then it must be of the form (m, σ), where a process $p_i \in P$ invoked $\mathsf{Sign}(i, T, m)$ and obtained σ as output.
2. **AARB-Validity:** Suppose that a non-faulty process AARB-delivers (m, σ) with respect to ID. Let $i = \mathsf{FindIndex}(T, m, \sigma)$ denote the output of an idealised call to FindIndex. Then if p_i is non-faulty, p_i must have anonymously broadcast (m, σ).
3. **AARB-Unicity:** Consider any point in time in which a non-faulty process p has AARB-delivered more than one tuple with respect to ID. Let $delivered = \{(m_1, \sigma_1), \ldots, (m_l, \sigma_l)\}$, where $|delivered| = l$, denote the set of these tuples. For each $i \in \{1, \ldots, l\}$, let $out_i = \mathsf{FindIndex}(T, m_i, \sigma_i)$ denote the output of an idealised call to FindIndex. Then for all distinct pairs of tuples $\{(m_i, \sigma_i), (m_j, \sigma_j)\}$, $out_i \neq out_j$.
4. **AARB-Termination-1:** If a process p_i is non-faulty and invokes $\mathsf{AARBP}[ID](m)$, all the non-faulty processes eventually AARB-deliver (m, σ) with respect to ID, where σ is the output of the call $\mathsf{Sign}(i, T, m)$.
5. **AARB-Termination-2:** If a non-faulty process AARB-delivers (m, σ) with respect to ID, then all the non-faulty processes eventually AARB-deliver (m, σ) with respect to ID.

Firstly, we require AARB-Signing to ensure that the other properties are meaningful. Since messages are anonymously broadcast, properties refer to the index of the signing process determined by an idealised call to FindIndex. In spirit, AARB-Validity ensures if a non-faulty process AARB-delivers a message that was signed by a non-faulty process p_i, then p_i must have invoked AARBP. Similarly, AARB-Unicity ensures that a non-faulty process will AARB-deliver at most one message signed by each process. We note that AARB-Termination-1 is insufficient for consensus: without AARB-Termination-2, different processes may AARB-deliver different messages produced by the *same* process if it is faulty, as in the two-step algorithm implementing no-duplicity broadcast [6, 50]. Finally, we state the anonymity property:

6. **AARB-Anonymity:** Suppose that non-faulty process p_i invokes $\mathsf{AARBP}[ID](m)$ for some m and a given ID, and has previously invoked an arbitrary number of $\mathsf{AARBP}[ID_j](m_j)$ calls where $ID \neq ID_j$ for all such j. Suppose that an adversary A is required to output a guess $g \in \{1, \ldots, n\}$,

corresponding to the identity of p_i after performing an arbitrary number of computations, allowed oracle calls and invocations of networking primitives. Then for any A and run, $Pr(i = g) \leq \frac{1}{n-t}$.

Informally, AARB-Anonymity guarantees that the source of an anonymously broadcast message by a non-faulty process is unknown to the adversary, in that it is indistinguishable from $n - t$ (non-faulty) processes. AARBP can be implemented by composing n instances of Bracha's reliable broadcast algorithm, which we describe and prove correct in the companion technical report [9].

4 Anonymity-preserving vector consensus

In this section, we introduce the *anonymity-preserving vector consensus* problem and present and discuss the protocol Anonymised Vector Consensus Protocol (AVCP) that solves it. The anonymity-preserving vector consensus problem brings anonymity to the vector consensus problem [24] where non-faulty processes reach an agreement upon a vector containing at least $n - t$ proposed values. More precisely, anonymised vector consensus asserts that the identity of a process who proposes must be indistinguishable from that of all non-faulty processes. As in AARBP, instances of AVCP are identified uniquely by a given value ID. Each process is equipped with two operations. Firstly, "$\mathsf{AVCP}[ID](m)$" begins execution of an instance of AVCP with respect to ID and proposal m. Secondly, "$\mathsf{AVC\text{-}decide}[ID](V)$" signals the output of V from the instance of AVCP identified by ID, and is invoked exactly once per identifier. We define a protocol that solves anonymity-preserving vector consensus with respect to these operations as satisfying the following four properties. Firstly, we require an anonymity property, defined analogously to that of AARB-broadcast:

1. **AVC-Anonymity:** Suppose that non-faulty process p_i invokes $\mathsf{AVCP}[ID](m)$ for some m and a given ID, and has previously invoked an arbitrary number of $\mathsf{AVCP}[ID_j](m_j)$ calls where $ID \neq ID_j$ for all such j. Suppose that an adversary A is required to output a guess $g \in \{1, \ldots, n\}$, corresponding to the identity of p_i after performing an arbitrary number of computations, allowed oracle calls and invocations of networking primitives. Then for any A and run, $Pr(i = g) \leq \frac{1}{n-t}$.

It also requires the original agreement and termination properties of vector consensus:

2. **AVC-Agreement:** All non-faulty processes that invoke $\mathsf{AVC\text{-}decide}[ID](V)$ do so with respect to the same vector V for a given ID.
3. **AVC-Termination:** Every non-faulty process eventually invokes $\mathsf{AVC\text{-}decide}[ID](V)$ for some vector V and a given ID.

It also requires a validity property that depends on a pre-determined, deterministic validity predicate $\mathsf{valid}()$ [10,18] which we assume is common to all processes. We assume that all non-faulty processes propose a value that satisfies $\mathsf{valid}()$.

4. **AVC-Validity:** Consider each non-faulty process that invokes AVC-decide$[ID](V)$ for some V and a given ID. Each value $v \in V$ must satisfy valid(), and $|V| \geq n - t$. Further, at least $|V| - t$ values correspond to the proposals of distinct non-faulty processes.

4.1 AVCP, the Anonymised Vector Consensus Protocol

We present a reduction to binary consensus which may converge in four message steps, as in the reduction to binary consensus of Democratic Byzantine Fault Tolerance (DBFT) [18], at least one of which must be performed over anonymous channels. We present the proof of correctness in the companion technical report [9]. We note that comparable problems [22], including agreement on a core set over an asynchronous network [4], rely on such a reduction but with a probabilistic solution. As in DBFT, we solve consensus deterministically by reliably broadcasting proposals which are then decided upon using n instances of binary consensus. The protocol is divided into two components. Firstly, the reduction component (Algorithm 2) reduces anonymity-preserving vector consensus to binary consensus. Here, one instance of AARBP and n instances of binary consensus are executed. But, since proposals are made anonymously, processes cannot associate proposals with binary consensus instances a priori. Consequently, processes start with n unlabelled binary consensus instances, and label them over time with the hash digest of proposals they deliver (of the form $h \in \{0,1\}^*$). To cope with messages sent and received in unlabelled binary consensus instances, we require a handler component (Algorithm 3) that replaces function calls made in binary consensus instances.

Functions. In addition to the communication primitives detailed in Section 2 and the two primitives "AVCP" and "AVC-decide", the following primitives may be called:

- "$inst$.bin_propose(v)", where $inst$ is an instance of binary consensus and $v \in \{0,1\}$, which begins execution of $inst$ with initial value v.
- "AARBP" and "AARB-deliver", as in Section 3.
- "valid()" as described above.
- "m.keys()" (resp. "m.values()"), which returns the keys (resp. values) of a map m.
- "$item$.key()", which returns the key of $item$ in a map m which is determined by context.
- "s.pop()", which removes and returns a value from set s.
- "$H(v)$", a cryptographic hash function which maps an element $v \in \{0,1\}^*$ to $h \in \{0,1\}^*$.

State. Each process tracks the following variables:

- $ID \in \{0,1\}^*$, a common identifier for a given instance of AVCP.

- *proposals*[], which maps labels of the form $l \in \{0,1\}^*$ to AARB-delivered messages of the form $(m, \sigma) \in (\{0,1\}^*, \{0,1\}^*)$ that may be decided, and is initially empty.
- *decision_count*, tracking the number of binary consensus instances for which a decision has been reached, initialised to 0.
- *decided_ones*, the set of proposals for which 1 was decided in the corresponding binary consensus instance, initialised to $\emptyset$.
- *labelled*[], which maps labels, which are the hash digest $h \in \{0,1\}^*$ of AARB-delivered proposals, to binary consensus instances, and is initially empty.
- *unlabelled*, a set of binary consensus instances with no current label, which is initially of cardinality n.
- *ones*[][], which maps two keys, EST and AUX, to maps with integer keys $r \geq 1$ which map to a set of labels, all of which are initially empty.
- *counts*[][], which maps two keys, EST and AUX, to maps with integer keys $r \geq 1$ which map to an integer $n \in \{0, \ldots, n\}$, all of which are initialised to 0.

Messages. In addition to messages propagated in AARBP, non-faulty processes process messages of the form $(ID, TAG, r, label, b)$, where $TAG \in \{EST, AUX\}$, $r \geq 1$, $label \in \{0,1\}^*$ and $b \in \{0,1\}$. A process buffers a message $(ID, TAG, r, label, b)$ until *label* labels an instance of binary consensus *inst*, at which point the message is considered receipt in *inst*. The handler, described below, ensures that all messages sent by non-faulty processes eventually correspond to a label in their set of labelled consensus instances (i.e., contained in *labelled*.keys()). Similarly, a non-faulty process can only broadcast such a message after labelling the corresponding instance of binary consensus. Processes also process messages of the form $(ID, TAG, r, ones)$, where $TAG \in \{EST\_ONES, AUX\_ONES\}$, $r \geq 1$, and *ones* is a set of strings corresponding to binary consensus instance labels.

Reduction. In the reduction, presented in Algorithm 2, n (initially unlabelled) instances of binary consensus are used, each corresponding to a value that one process in P may propose. Each (non-faulty) process invokes AARBP with respect to ID and their value m' (line 2), anonymously broadcasting (m', σ') inside the AARBP instance. On AARB-delivery of some message (m, σ), an unlabelled instance of binary consensus is deposited into *labelled*, whose key (label) is set to $H(m \mid\mid \sigma)$ (line 10). Proposals that fulfil valid() are stored in *proposals* (line 12), and *inst*.bin_propose(1) is invoked with respect to the newly labelled instance $inst = labelled[H(m \mid\mid \sigma)]$ if not yet done (line 13). Upon termination of each instance (line 14), provided 1 was decided, the corresponding proposal is added to *decided_ones* (line 16). For either decision value, *decision_count* is incremented (line 17). Once 1 has been decided in $n - t$ instances of binary consensus, processes will propose 0 in all instances that they have not yet proposed in (line 6). Note that upon AARB-delivery of valid messages after this point, bin_propose(1) is not invoked at line 13. Upon the termination of all n instances of binary consensus (after line 7), all non-faulty processes decide their set of values for which 1 was decided in the corresponding instance of binary consensus (line 8).

Algorithm 2 AVCP (1 of 2): Reduction to binary consensus

```
1: AVCP[ID](m'):
2:    AARBP[ID](m')                          ▷ anonymised reliable broadcast of proposal
3:    wait until |decided_ones| ≥ n − t      ▷ wait until n − t instances terminate with 1
4:    for each inst ∈ unlabelled ∪ labelled.values() such that
5:    inst.bin_propose() not yet invoked do
6:       Invoke inst.bin_propose(0)          ▷ propose 0 in all binary consensus not yet invoked
7:    wait until decision_count = n          ▷ wait until all n instances of binary consensus terminate
8:    AVC-decide[ID](decided_ones)

9:  upon invocation of AARB-deliver[ID](m, σ)
10:   labelled[H(m || σ)] ← unlabelled.pop()
11:   if valid(m, σ) then                    ▷ deterministic, common validity function
12:      proposals[H(m || σ)] ← (m, σ)
13:      Invoke labelled[H(m || σ)].bin_propose(1) if not yet invoked

14: upon inst deciding a value v ∈ {0, 1}, where inst ∈ labelled.values() ∪ unlabelled
15:   if v = 1 then ▷ store proposals for which 1 was decided in the corresponding binary consensus
16:      decided_ones ← decided_ones ∪ {proposals[inst.key()]}
17:   decision_count ← decision_count + 1
```

Handler. As proposals are anonymously broadcast, binary consensus instances cannot be associated with process identifiers a priori, and so are labelled by AARB-delivered messages. Thus, we require the handler, which overrides two of the three broadcast calls in the non-terminating variant of the binary consensus of [18] (Algorithm 1).

We now describe the handler (Algorithm 3). Let $inst$ be an instance of binary consensus. On calling $inst$.bin_propose(b) ($b \in \{0, 1\}$) (and at the beginning of each round $r \geq 1$), processes invoke BV-broadcast (line 5 of Algorithm 1), immediately calling "broadcast (ID,EST, r, $label$, b)" (line 19 of Algorithm 1). If $b = 1$, (ID,EST, r, $label$, 1) is broadcast, and $label$ is added to the set $ones[EST][r]$ (line 21). Note that, given AARB-Termination-1, all messages sent by non-faulty processes of the form (ID,EST, r, $label$, 1) will be deposited in an instance $inst$ labelled by $label$. Then, as the binary consensus routine terminates when all non-faulty processes propose the same value, all processes will decide the value 1 in $n - t$ instances of binary consensus (i.e., will pass line 3), after which they execute bin_propose(0) in the remaining instances of binary consensus.

Since these instances may not be labelled when a process wishes to broadcast a value of the form (ID,EST, r, $label$, 0), we defer their broadcast until "broadcast (ID,EST, r, $label$, b)" is called in all n instances of binary consensus. At this point (line 23), (ID,EST_ONES, r, $ones[EST][r]$) is broadcast (line 24). A message of the form (ID,EST_ONES, r, $ones$) is interpreted as the receipt of zeros in all instances not labelled by elements in $ones$ (at lines 38 and 40). This can only be done once all elements of $ones$ label instances of binary consensus (i.e., after

Algorithm 3 AVCP (2 of 2): Handler of Algorithm 1

18: **upon** "broadcast $(ID, EST, r, label, b)$" in $inst \in labelled.\mathsf{values}() \cup unlabelled$
19: **if** b $= 1$ **then**
20: broadcast $(ID, EST, r, label, b)$
21: $ones[EST][r] \leftarrow ones[EST][r] \cup \{inst.\mathsf{key}()\}$
22: $counts[EST][r] \leftarrow counts[EST][r] + 1$
23: **if** $counts[EST][r] = n \wedge |ones[EST][r]| < n$ **then**
24: broadcast $(ID, EST\_ONES, r, ones[EST][r])$
25: **upon** "broadcast $(ID, AUX, r, label, b)$" in $inst \in labelled.\mathsf{values}() \cup unlabelled$
26: **if** b $= 1$ **then**
27: broadcast $(ID, AUX, r, label, b)$
28: $ones[AUX][r] \leftarrow ones[AUX][r] \cup \{inst.\mathsf{key}()\}$
29: $counts[AUX][r] \leftarrow counts[AUX][r] + 1$
30: **if** $counts[AUX][r] = n \wedge |ones[AUX][r]| < n$ **then**
31: broadcast $(ID, AUX\_ONES, r, ones[AUX][r])$
32: **upon** receipt of $(ID, TAG, r, ones)$ s.t. $TAG \in \{EST\_ONES, AUX\_ONES\}$
33: **wait until** $one \in labelled.\mathsf{keys}()\ \forall one \in ones$
34: **if** $TAG = EST\_ONES$ **then**
35: $TEMP \leftarrow EST$
36: **else** $TEMP \leftarrow AUX$
37: **for each** $l \in labelled.\mathsf{keys}()$ such that $l \notin ones$ **do**
38: deliver $(ID, TEMP, r, l, 0)$ in $labelled[l]$
39: **for each** $inst \in unlabelled$ **do**
40: deliver $(ID, TEMP, r, \perp, 0)$ in $inst$

line 33). Note that if $|ones[EST][r] = n|$, then there are no zeroes to be processed by receiving processes, and so the broadcast at line 24 can be skipped.

Handling "broadcast $(ID, AUX, r, label, b)$" calls (line 7 of Algorithm 1) is identical to the handling of initial "broadcast $(ID, EST, r, label, b)$" calls. Note that the third broadcast in the original algorithm, where $(ID, EST, r, label, b)$ is broadcast upon receipt from $t + 1$ processes if not yet done before (line 21 of Algorithm 1 (BV-Broadcast)), can only occur once the corresponding instance of binary consensus is labelled. Thus, it does not need to be handled. From here, we can see that messages in the handler are processed as if n instances of the original binary consensus algorithm were executing.

Table 1: Comparing the complexity of AVCP and DBFT [18] after GST [25]

Complexity	**AVCP**	**DBFT**
Best-case message complexity	$O(n^3)$	$O(n^3)$
Worst-case message complexity	$O(tn^3)$	$O(tn^3)$
Best-case complexity	$O((S+c)n^3)$	$O(n^3)$
Worst-case bit complexity	$O((S+c)tn^3)$	$O(tn^3)$

4.2 Complexity and optimizations

Let k be a security parameter, S the size of a signature and c the size of a message digest. In Table 1, we compare the message and communication complexities of AVCP against DBFT [18], which, as written, can be easily altered to solve vector consensus. We assume that AVCP is invoking the terminating variant of the binary consensus of [18]. When considering complexity, we only count messages in the binary consensus routines once the global stabilisation time (GST) has been reached [25]. Both best-free and worst-case message complexity are identical between the two protocols. We remark that there exist runs of AVCP where processes are faulty which has the best-case message complexity $O(n^3)$, such as when a process has crashed. AVCP mainly incurs greater communication complexity proportional to the size of the signatures, which can vary from size $O(k)$ [36, 54] to $O(kn)$ [29]. If processes make a single anonymous broadcast per run, the best-case and worst-case bit complexities of AVCP are lowered to $O(Sn^2 + cn^3)$ and $O(Sn^2 + ctn^3)$.

As is done in DBFT [18], we can combine the anonymity-preserving all-to-all reliable broadcast of a message m and the proposal of the binary value 1 in the first round of a binary consensus instance. To this end, a process may skip the BV-broadcast step in round 1, which may allow AVCP to converge in four message steps, at least one of which must be anonymous. It may be useful to invoke "broadcast $TAG[r](b)$", where $TAG \in \{EST, AUX\}$ (lines 20 and 27) when the instance of binary consensus is labelled, rather than simply when $b = 1$ (i.e., the condition preceding these calls). Since it may take some time for all n instances of binary consensus to synchronise, doing this may speed up convergence in the "faster" instances.

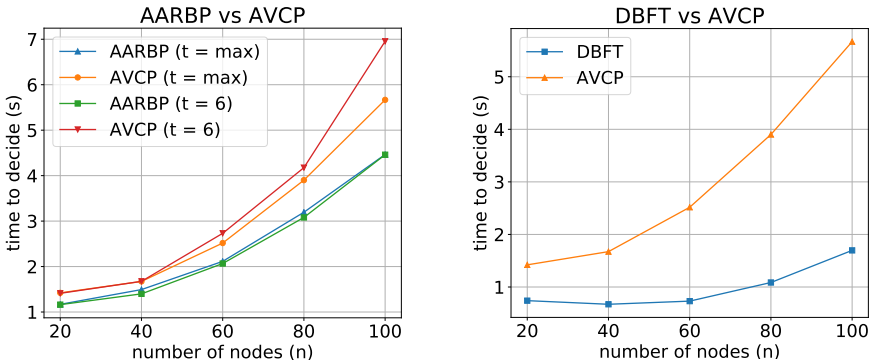
5 Experiments

In order to evaluate the practicality of our solutions, we implemented our distributed protocols and deployed them on Amazon EC2 instances. We refer to each EC2 instance used as a *node*, corresponding to a 'process' as in the protocol descriptions. For each value of n (the number of nodes) chosen, we ran experiments with an equal number of nodes from *four* regions: Oregon (us-west-2), Ohio (us-east-2), Singapore (ap-southeast-1) and Frankfurt (eu-central-1). The type of instance chosen was c4.xlarge, which provide 7.5 GiB of memory, and 4 vCPUs, i.e., 4 cores of an Intel Xeon E5-2666 v3 processor. We performed between 50 and 60 iterations for each value of n and t we benchmarked. We varied n incrementally, and varied t both with respect to the maximum fault-tolerance (i.e., $t = \lfloor \frac{n-1}{3} \rfloor$), and also fixed $t = 6$ for values of $n = 20, 40, \ldots$ All networking code, and the application logic, was written in Python (2.7). As we have implemented our cryptosystems in golang, we call our libraries from Python using ctypes[4]. To simulate reliable channels, nodes communicate over

[4] https://docs.python.org/2/library/ctypes.html

TCP. Nodes begin timing once all connections have been established (i.e., after handshaking).

Our protocol, Anonymised Vector Consensus Protocol (AVCP), was implemented on top of the existing DBFT [18] codebase, as was the case with our implementation of AARB-broadcast, i.e., AARBP. We do not use the fast-path optimisation described in Section 4, but we hash messages during reliable broadcast to reduce bandwidth consumption. We use the most conservative choice of ring signatures, $O(kn)$-sized traceable ring signatures [30], which require $O(n)$ operations for each signing and verification call, and $O(n^2)$ work for tracing overall. Each process makes use of a single anonymous broadcast in each run of the algorithm. To simulate the increased latency afforded by using publicly-deployed anonymous networks, processes invoke a local timeout for 750 ms before invoking anon_broadcast, which is a regular broadcast in our experiments.



(a) Comparing AARBP and AVCP (b) Comparing DBFT and AVCP

Fig. 1: Evaluating the cost of the reliable anonymous broadcast (AARBP) in our solution (AVCP) and the performance of our solution (AVCP) compared to an efficient Byzantine consensus baseline (DBFT) without anonymity preservation

Figure 1a compares the performance of AARBP with that of AVCP. In general, convergence time for AVCP is higher as we need at least three more message steps for a process to decide. Given that the fast-path optimisation is used, requiring 1 additional message step over AARBP in the good case, the difference in performance between AVCP and AARBP would indeed be smaller.

Comparing AVCP with $t = max$ and $t = 6$, we see that when $t = 6$, convergence is slower. Indeed, AVC-Validity states at least $n - t$ values fulfilling valid() are included in a process' vector given that they decide. Consequently, as t is smaller, $n - t$ is larger, and so nodes will process and decide more values. Although AARB-delivery may be faster for some messages, nodes generally have to perform more TRS verification/tracing operations. As nodes decide 1 in more instances of binary consensus, messages of the form $(ID, TAG, r, ones)$ are prop-

agated where $|ones|$ is generally larger, slowing down decision time primarily due to the size of the message.

Figure 1b compares the performance of DBFT to solve vector consensus against AVCP. Indeed, the difference in performance between AVCP and DBFT when $n = 20$ and $n = 40$ is primarily due to AVCP's 750 ms timeout. As expected when scaling up n further, cryptographic operations result in increasingly slower performance for AVCP.

Overall, AVCP performs reasonably well, reaching convergence when $n = 100$ between 5 and 7 seconds depending on t, which is practically reasonable, particularly when used to perform elections which are typically occasional.

6 Discussion

It is clear that anonymity is preserved if processes only use anonymous channels to communicate, provided that processes do not double-sign with ring signatures for each message type. For performance and to prevent long-term correlation attacks on anonymous networks like Tor [43], it may be of interest to use anonymous message passing to propose a value, and then to use regular channels for the rest of a given protocol execution. In this setting, the adversary can de-anonymise a non-faulty process by observing message transmission time [2] and the order in which messages are sent and received [49]. For example, a single non-faulty process may be relatively slow, and so the adversary may deduce that messages it delivers late over anonymous channels were sent by that process.

Achieving anonymity in this setting in practice depends on the latency guarantees of the anonymous channels, the speed of each process, and the latency guarantees of the regular channels. One possible strategy could be to use public networks like Tor [23] where message transmission time through the network can be measured.[5] Then, based on the behaviour of the anonymous channels, processes can vary the timing of their own messages by introducing random message delays [43] to minimise the correlation between messages over the different channels. It may also be useful for processes to synchronise between protocol executions. This prevents a process from being de-anonymised when they, for example, invoke anon_send in some instance when all other processes are executing in a different instance.

In terms of future work, it is of interest to evaluate anonymity in different formal models [37, 52] and with respect to various practical attack vectors [49]. It will be useful also to formalise anonymity under more practical assumptions so that the timing of anonymous and regular message passing do not correlate highly. In addition, a reduction to a randomized [11] binary consensus algorithm would remove the dependency on the weak coordinator used in each round of the binary consensus algorithm we rely on [18].

[5] https://metrics.torproject.org/

Acknowledgment

This research is supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled "Taipan: A Blockchain with Democratic Consensus and Validated Contracts" and Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things".

References

1. Ben Adida. Helios: Web-based open-audit voting. In *USENIX Security*, pages 335–348, 2008.
2. Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *International Workshop on Information Hiding*, pages 245–257, 2001.
3. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.
4. Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, pages 183–192, 1994.
5. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
6. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *PODC*, pages 12–26, 1983.
7. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *JACM*, 32(4):824–840, 1985.
8. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.
9. Christian Cachin, Daniel Collins, Tyler Crain, and Vincent Gramoli. Anonymity preserving Byzantine vector consensus. *CoRR*, abs/1902.10010, 2020.
10. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.
11. Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.
12. Jean Camp, Michael Harkavy, J. D. Tygar, and Bennet Yee. Anonymous atomic transactions. In *In Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, 1996.
13. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
14. David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203, 1983.
15. David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.
16. David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *CACM*, 24(2):84–90, 1981.
17. Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

18. Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In *NCA*, pages 1–8, 2018.
19. Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. Multi-authority secret-ballot elections with linear work. In *Eurocrypt*, pages 72–83, 1996.
20. George Danezis and Claudia Diaz. A survey of anonymous communication channels. Technical Report MSR-TR-2008-35, Microsoft Research, 2008.
21. Yvo Desmedt. Threshold cryptosystems. In *International Workshop on the Theory and Application of Cryptographic Techniques*, pages 1–14, 1992.
22. Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Nikos Chondros, and Mema Roussopoulos. Interactive consistency in practical, mostly-asynchronous systems. In *ICPADS*, pages 752–759, 2015.
23. Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security*, pages 21–21, 2004.
24. Assia Doudou and André Schiper. Muteness failure detectors for consensus with Byzantine processes. In *PODC*, page 315, 1997.
25. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.
26. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
27. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
28. Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Eurocrypt*, pages 1–19, 2004.
29. Eiichiro Fujisaki. Sub-linear size traceable ring signatures without random oracles. In *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, volume E95.A, pages 393–415, 04 2011.
30. Eiichiro Fujisaki and Koutarou Suzuki. Traceable ring signature. In *PKC*, pages 181–200, 2007.
31. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
32. Yossi Gilad and Amir Herzberg. Spying in the dark: Tcp and tor traffic analysis. In *Privacy Enhancing Technologies*, pages 100–119, 2012.
33. Philippe Golle, Markus Jakobsson, Ari Juels, and Paul Syverson. Universal re-encryption for mixnets. In *CT-RSA*, pages 163–178, 2004.
34. Philippe Golle and Ari Juels. Dining cryptographers revisited. In *Eurocrypt*, pages 456–473, 2004.
35. Jens Groth. Efficient maximal privacy in boardroom voting and anonymous broadcast. In *Financial Cryptography*, pages 90–104, 2004.
36. Ke Gu, Xinying Dong, and Linyu Wang. Efficient traceable ring signature scheme without pairings. *Advances in Mathematics of Communications*, page 0, 2019.
37. Joseph Y Halpern and Kevin R O'Neill. Anonymity and information hiding in multiagent systems. *Journal of Computer Security*, 13(3):483–514, 2005.
38. Markus Jakobsson. A practical mix. In *Eurocrypt*, pages 448–461, 1998.
39. Ari Juels, Dario Catalano, and Markus Jakobsson. *Coercion-Resistant Electronic Elections*, pages 37–63. Springer, 2010.
40. O. Kulyk, S. Neumann, M. Volkamer, C. Feier, and T. Koster. Electronic voting with fully distributed trust and maximized flexibility regarding ballot design. In *EVOTE*, pages 1–10, 2014.

41. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982.
42. Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Information Security and Privacy*, pages 325–335, 2004.
43. Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *International Workshop on Privacy Enhancing Technologies*, pages 17–34, 2004.
44. Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *JACM*, 62(4):31, 2015.
45. Steven J. Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *S&P*, pages 183–195, 2005.
46. C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS*, pages 116–125, 2001.
47. Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Solving vector consensus with a wormhole. *TPDS*, 16(12):1120–1131, 2005.
48. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, pages 223–238, 1999.
49. Jean-François Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, pages 10–29, 2001.
50. Michel Raynal. *Reliable Broadcast in the Presence of Byzantine Processes*, pages 61–73. Springer, 2018.
51. Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, pages 552–565, 2001.
52. Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In *International Workshop on Privacy Enhancing Technologies*, pages 41–53, 2002.
53. Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Journal of Cryptology*, 15(2):75–96, 2002.
54. Patrick P. Tsang and Victor K. Wei. Short linkable ring signatures for e-voting, e-cash and attestation. In *ISPEC*, pages 48–60, 2005.
55. Patrick P. Tsang, Victor K. Wei, Tony K. Chan, Man Ho Au, Joseph K. Liu, and Duncan S. Wong. Separable linkable threshold ring signatures. In *INDOCRYPT*, pages 384–398, 2005.
56. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.
57. Qingsong Ye, Huaxiong Wang, and Josef Pieprzyk. Distributed private matching and set operations. In *ISPEC*, pages 347–360, 2008.
58. Bassam Zantout and Ramzi Haraty. I2P data communication system. In *ICN*, pages 401–409, 2011.