BFT Protocol Forensics

Peiyao Sheng*
psheng2@illinois.edu
University of Illinois at Urbana Champaign, USA

Gerui Wang*
geruiw2@illinois.edu
University of Illinois at Urbana Champaign, USA

Kartik Nayak
kartik@cs.duke.edu
Duke University, USA

Sreeram Kannan
ksreeram@ece.uw.edu
University of Washington, USA

Pramod Viswanath
pramodv@illinois.edu
University of Illinois at Urbana Champaign, USA

ABSTRACT

Byzantine fault-tolerant (BFT) protocols allow a group of replicas to come to consensus even when some of the replicas are Byzantine faulty. There exist multiple BFT protocols to securely tolerate an optimal number of faults t under different network settings. However, if the number of faults f exceeds t then security could be violated. In this paper we mathematically formalize the study of *forensic support* of BFT protocols: we aim to identify (with cryptographic integrity) as many of the malicious replicas as possible and in as distributed manner as possible. Our main result is that forensic support of BFT protocols depends heavily on minor implementation details that do not affect the protocol's security or complexity. Focusing on popular BFT protocols (PBFT, HotStuff, Algorand) we exactly characterize their forensic support, showing that there exist minor variants of each protocol for which the forensic supports vary widely. We show strong forensic support capability of LibraBFT, the consensus protocol of Diem cryptocurrency; our lightweight forensic module implemented on a Diem client is open-sourced [4] and is under active consideration for deployment in Diem. Finally, we show that all secure BFT protocols designed for $2t + 1$ replicas communicating over a synchronous network forensic support is inherently nonexistent; this impossibility result holds for all BFT protocols and even if one has access to the states of all replicas (including Byzantine ones).

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

forensics; BFT protocols; blockchains

*The first two authors contributed equally to this work.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484566

ACM Reference Format:
Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. 2021. BFT Protocol Forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3460120.3484566

1 INTRODUCTION

Byzantine Fault Tolerant (BFT) protocols, guaranteeing distributed consensus among parties that follow the protocol, are a corner stone of distributed system theory. In the relatively recent context of blockchains, BFT protocols have received renewed attention; new and efficient (state machine replication (SMR)) BFT protocols specifically designed for blockchains have been constructed (e.g., Algorand [18], HotStuff [32], Streamlet [12]). The core theoretical security guarantee is that as long as a certain fraction of nodes are "honest", i.e., they follow the protocol, then these nodes achieve consensus with respect to (a time evolving) state machine regardless of the Byzantine actions of the remaining malicious nodes. When the malicious nodes are sufficiently numerous, e.g., strictly more than a 1/3 fraction of nodes in a partially synchronous network, they can "break security", i.e., band together to create different views at the honest participants.

In this paper, we are focused on "the day after" [16]: events *after* malicious replicas have successfully mounted a security breach. Specifically, we focus on identifying which of the participating replicas acted maliciously; we refer to this action as "forensics". Successful BFT protocol forensics meets two goals:

- identify as many of the nodes that acted maliciously as possible with an irrefutable cryptographic proof of culpability;
- identification is conducted as distributedly as possible, e.g., by the individual nodes themselves, with no/limited communication between each other after the security breach.

Main contributions. Our central finding is that the forensic possibilities crucially depend on minor implementation details of BFT protocols; the details themselves do not affect protocol security or performance (latency and communication complexity); we demonstrate our findings in the context of several popular BFT protocols for Byzantine Agreement (BA). We present our findings in the context of a mathematical and systematic formulation of the "forensic support" of BFT protocols. The forensic support of any BFT protocol is parameterized as a triplet (m, k, d), that represents the aforementioned goals of forensic analysis. The triplet (m, k, d) along with traditional BFT protocol parameters of (n, t, f) is summarized in

Symbol	Interpretation
n	total number of replicas
t	maximum number of faults for obtaining agreement and termination
f	actual number of faults
m	maximum number of Byzantine replicas under which forensic support can be provided
k	the number of different honest replicas' transcripts needed to guarantee a proof of culpability
d	the number of Byzantine replicas that can be held culpable in case of an agreement violation

Table 1: Summary of notations.

Protocols	Forensic Support	Parameters		
		m	k	d
PBFT-PK HotStuff-view VABA	Strong	$2t$	1	$t+1$
HotStuff-hash	Medium	$2t$	$t+1$	$t+1$
PBFT-MAC	None	$t+1$	$2t$	0
HotStuff-null				1
Algorand				0

Table 2: Summary of results; the forensic support values of d are the largest possible and $n = 3t + 1$ here.

Table 1. We emphasize that each of the protocol variants is safe and live when $f \leq t$ (here $n = 3t + 1$ for all the protocols considered) but their forensic supports are quite different.

Security attacks on BFT protocols can be far more subtle than simply double voting or overt equivocation (when the culpable replicas are readily identified), and the forensic analysis is correspondingly subtle. This paper brings to fore a new dimension for secure BFT protocol design beyond performance (low latency and low communication complexity): forensic support capabilities.

We analyze the forensic support of classical and state-of-the-art BFT protocols for Byzantine Agreement. Our main findings, all in the settings of safety violation (agreement violation), are the following, summarized in Table 2.

- **Parameter d.** We show that the number of culpable replicas d that can be identified is either 0 or as large as $t+1$. In other words, if at least one replica can be identified then we can also identify the largest possible number, $t + 1$ replicas; the only exception is for HotStuff-null, where in a successful safety attack, we can identity the culpability of one malicious replica.
- **Parameter m.** We show that the maximum number of Byzantine replicas m allowed for nontrivial forensic support (i.e., $d > 0$) cannot be more than $2t$. Furthermore, any forensic support feasible with m is also feasible with m being its largest value $2t$, i.e., if (m, k, d) forensic support is feasible, then $(2t, k, d)$ is also feasible.
- **Parameter k.** Clearly at least one replica's transcript is needed for forensic analysis, so $k = 1$ is the least possible value. This suffices for several of the BFT protocol variants. However for HotStuff-hash, k needs to be at least $t + 1$ for any nontrivial forensic analysis.
- **Strong forensic support.** The first three items above imply that the strongest possible forensic support is $(2t, 1, t+1)$. Further, the BFT protocols in Table 2 that achieve any nontrivial forensic support automatically achieve the strongest possible forensics (the only exception is HotStuff-hash, for which the forensic support we identified is the best possible).
- **Impossibility.** For certain variants of BFT protocols (PBFT-MAC, HotStuff-null, and Algorand), even with transcripts from all honest replicas, non-trivial forensics is simply not possible, i.e., $d = 0$ even if m is set to its smallest and k set to its largest possible values ($t + 1$ and $2t$ respectively); again, HotStuff-null allows the culpability of a single malicious replica.
- **Practical impact.** Forensic support is of immediate interest to practical blockchain systems; we conduct a forensic support analysis of LibraBFT, the consensus protocol in the new cryptocurrency Diem, and show in-built strong forensic support. We have implemented the corresponding forensic analysis algorithm inside of a Diem client and built an associated forensics dashboard; our reference implementation is available open-source [4] and is under active consideration for deployment in Diem.
- **BFT with $n = 2t + 1$.** For a secure BFT protocol operating on a synchronous network, the ideal setting is $n = 2t + 1$. For *every* such protocol we show that at most one culpable replica can be identified (i.e., d is at most 1) even if we have access to the state of all honest nodes, i.e, $k = t$.

Outline. We describe our results in the context of related work in §2; to the best of our knowledge, this is the first paper to systematically study BFT protocol forensics. The formal forensic support problem statement and security model is in §3. The forensic support of PBFT, HotStuff and Algorand (and variants) are explored in §4, §5, §6, respectively. Our forensic study of LibraBFT and implementation of the corresponding forensic protocol is the focus of §7. In appendix §A, we present the forensic support analysis for VABA, a state-of-the-art efficient BFT for asynchronous network conditions. The impossibility of forensic support for all BFT protocols operating in the classical $n = 2t + 1$ synchronous network setting is shown in §B. Our choice of the 5 protocols studied here (PBFT, HotStuff, Algorand, LibraBFT, VABA) is made with the goal of covering a variety of settings: (a) partially synchronous vs asynchronous; (b) authenticated vs non-authenticated; (c) player replaceable vs irreplaceable; (d) chained version vs single-shot version; (e) variants that communicate differing amounts of auxiliary information. Stitching the results across the 5 different protocols into a coherent theory of forensic support of abstract BFT protocols is an exciting direction of future work; this is discussed in §8.

2 RELATED WORK

BFT protocols. PBFT [10, 11] is the first practical BFT SMR protocol in the partially synchronous setting, with quadratic communication complexity of view change. HotStuff [32] is the first partially synchronous SMR protocol that enjoys both a linear communication of view change and optimistic responsiveness. Streamlet [12] is another SMR protocol known for its simplicity and textbook construction. Asynchronous Byzantine agreement is solved by [2] with asymptotically optimal communication complexity and round number. Synchronous protocols such as [1, 3] aim at optimal latency. Algorand [14, 18] designs a committee self-selection mechanism, and the Byzantine agreement protocol run by the committee decides the output for all replicas.

Beyond one-third faults. The seminal work of [17] shows that it is impossible to solve BA when the adversary corrupts one-third replicas for partially synchronous communication (the same bound holds for SMR). In BFT2F [22], a weaker notion of safety is defined, and a protocol is proposed such that when the adversary corrupts more than one-third replicas, the weaker notion of safety remains secure whereas the original safety might be violated. However, the weaker notion of safety does not protect the system against common attacks, e.g., double-spending attack in distributed payment systems. Flexible BFT [23] considers the case where clients have different beliefs in the number of faulty replicas and can act to confirm accordingly. Its protocol works when the sum of Byzantine faults and alive-but-corrupt faults, a newly defined type of faults, are beyond one-third. Two recent works [21, 31] propose BFT SMR protocols that can tolerate more than one-third Byzantine faults after some specific optimistic period. The goal of these works is to mask the effects of faults, even when they are beyond one-third, quite different from the goals of forensic analysis.

Distributed system forensics. Accountability has been discussed in seminal works [19, 20] for distributed systems in general. In the lens of BFT consensus protocols, accountability is defined as the ability for a replica to prove the culpability of a certain number of Byzantine replicas in [15]. Polygraph, a new BFT protocol with high communication complexity $O(n^4)$, is shown to attain this property in [15]. Reference [28] extends Polygraph to an SMR protocol, and devises a finality layer to "merge" the disagreement. Finality and accountability are also discussed in other recent works; examples include Casper [8], GRANDPA [29], and Ebb-and-flow [26]. Casper and GRANDPA identify accountability as a central problem and design their consensus protocols around this goal. Ebb-and-flow [26] observes that accountability is immediate in BFT protocols for safety violations via equivocating votes; however, as pointed out in [15], safety violations can happen during the view change process and this is the step where accountability is far more subtle.

Reference [15] argues that PBFT is not accountable and cannot be modified to be accountable without significant change/cost. We point out that the definition of accountability in [15] is rather narrow: two replicas with differing views have to themselves be able to identify culpability of malicious replicas. On the other hand, in forensic support, we study the number of honest replicas (not necessarily restricted to the specific two replicas which have identified a security breach) that can identify the culpable malicious replicas. Thus the definition of forensic support is more flexible than the one on accountability. Moreover, our work shows that we can achieve forensic support for protocols such as PBFT and HotStuff without incurring additional communication complexity other than (i) sending a proof of culpability to the client in case of a safety violation, and (ii) the need to use aggregate signatures instead of threshold signatures.

3 PROBLEM STATEMENT AND MODEL

The goal of state machine replication (SMR) is to build a replicated service that takes requests from clients and provides the clients with the interface of a single non-faulty server, i.e., each client receives the same totally ordered sequence of values. To achieve this, the replicated service uses multiple servers, also called replicas, some of which may be Byzantine, where a faulty replica can behave arbitrarily. A secure state machine replication protocol satisfies two guarantees. **Safety**: Any two honest replicas cannot output different sequences of values. **Liveness**: A value sent by a client will eventually be output by honest replicas.

SMR setting also has external validity, i.e., replicas only output non-duplicated values sent by clients. These values are eventually learned by the clients. Depending on the context, a replica may be interested in learning about outputs too. Hence, whenever we refer to a client for learning purposes, it can be an external entity or a server replica. A table of notations is in Table 1.

In this paper, for simplicity, we focus on the setting of outputting a *single value* instead of a sequence of values. The safety and liveness properties of SMR can then be expressed using the following definition:

Definition 3.1 (Validated Byzantine Agreement). A protocol solves validated Byzantine agreement among n replicas tolerating a maximum of t faults, if it satisfies the following properties:

(Agreement) Any two honest replicas output values v and v', then $v = v'$.

(Validity) If an honest replica outputs v, v is an externally valid value, i.e., v is signed by a client.

(Termination) All honest replicas start with externally valid values, and all messages sent among them have been delivered, then honest replicas will output a value.

Forensic support. Traditionally, consensus protocols provide guarantees only when $f \leq t$. When $f > t$, there can be a safety or liveness violation; this is the setting of study throughout this paper. Our goal is to provide *forensic support* whenever there is a safety violation (or agreement violation) and the number of Byzantine replicas in the system are not too high. In particular, if the actual number of Byzantine faults are bounded by m (for some $m > t$) and there is a safety violation, we can detect d Byzantine replicas using a *forensic protocol*. The protocol takes as input, the transcripts of honest parties, and outputs and *irrefutable* proof of d culprits. With the irrefutable proof, any party (not necessarily in the BFT system) can be convinced of the culprits' identities *without* any assumption on the number of honest replicas. However, if even with transcripts from all honest replicas, no forensic protocol can output such a proof, the consensus protocol has no forensic support (denoted as "None" in Table 2). Note that when we say a protocol

has no forensic support, we are referring to an impossibility w.r.t. providing irrefutable proof for d culprits (more precisely, in the context of Definition 3.2). In a general sense, there are other ways to provide forensics related to hardware, software, and network except for non-repudiation in the protocol.

To provide forensic support, we consider a setting where a client observes the existence of outputs for two conflicting (unequal) values.[1] By running a forensic protocol, the client sends (possibly a subset of) these conflicting outputs to all replicas and waits for their replies. Some of these replicas may be "witnesses" and may have (partial) information required to construct the irrefutable proof. After receiving responses from the replicas, the client constructs the proof. We denote by k the total number of transcripts from different honest replicas that are stored by the client to construct the proof.

Definition 3.2. $(\mathbf{m}, \mathbf{k}, \mathbf{d})$-**Forensic Support**. If $t < f \leq m$ and two honest replicas output conflicting values, then using the transcripts of all messages received from k honest replicas during the protocol, a client can provide an irrefutable proof of culpability of at least d Byzantine replicas.

Other assumptions. We consider forensic support for multiple protocols each with their own network assumptions. For PBFT and HotStuff, we assume a partially synchronous network [17]. For VABA [2] and Algorand [14], we suppose asynchronous and synchronous networks respectively.

We assume all messages are digitally signed except for one variant of PBFT (§4.3) that sometimes relies on the use of Message Authenticated Codes (MACs). Some protocols, e.g., HotStuff, VABA, use threshold signatures. For forensic purposes, we assume multi-signatures [7] instead (possibly worsening the communication complexity in the process). Whenever the number of signatures exceeds a threshold, the resulting aggregate signature is denoted by a pair $\sigma = (\sigma_{agg}, \epsilon)$, where ϵ is a bitmap indicating whose signatures are included in σ_{agg}. We define the intersection of two aggregated messages to be the set of replicas who sign both messages, i.e., $\sigma \cap \sigma' := \{i | \sigma.\epsilon[i] \wedge \sigma'.\epsilon[i] = 1\}$. An aggregate signature serves as a quorum certificate (QC) in our protocols, and we will use the two terms interchangeably. We assume a collision resistant cryptographic hash function.

4 FORENSIC SUPPORT FOR PBFT

PBFT is a classical partially synchronous consensus protocol that provides an optimal resilience of t Byzantine faults out of $n = 3t + 1$ replicas. However, when the actual number of faults f exceeds t, it does not provide any safety or liveness. In this section, we show that when $f > t$ and in case of a safety violation, the variant of the PBFT protocol (referred to as PBFT-PK) where all messages sent by parties are signed, has the *strongest forensic support*. Further, we show that for an alternative variant where parties sometimes only use MACs (referred to as PBFT-MAC), forensic support is impossible.

[1] We assume all the (honest) replica outputs are eventually learned by the client. In practice, the client may monitor the outputs by periodically communicating with all replicas.

4.1 Overview

We start with an overview focusing on a single-shot version of PBFT, i.e., a protocol for consensus on a single value. The protocol described here uses digital signatures to authenticate all messages and routes messages through leaders as shown in [27]; however we note that our arguments for PBFT-PK also apply to the original protocol in [11].

The protocol proceeds in a sequence of consecutive views denoted as view number $e = 1, 2, \cdots$. A higher view is a view with a larger view number. Each view has a unique leader. Each view of PBFT progresses as follows:

- **Pre-prepare.** The leader proposes a NewView message containing a proposal v and a status certificate M (explained later) to all replicas.
- **Prepare.** On receiving the first NewView message containing a valid value v in a view e, a replica sends Prepare for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *prepareQC*. The leader sends *prepareQC* to all replicas.
- **Commit.** On receiving a *prepareQC* in view e containing message v, a replica locks on (v, e) and sends Commit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *commitQC*. The leader sends *commitQC* to all replicas.
- **Reply.** On receiving *commitQC* from the leader, replicas output v and send a Reply (along with *commitQC*) to the client.

Once a replica locks on a value v in view e, we call (v, e) is the current lock of this replica. And a higher lock is a lock formed in a higher view. With lock (v, e), the replica only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when the status certificate M provides sufficient information stating that $2t + 1$ replicas are not locked on v. At the end of a view, every replica sends its lock to the leader of the next view. The next view leader collects $2t + 1$ such values as a status certificate M.

The safety of PBFT follows from two key quorum intersection arguments:

Uniqueness within a view. Within a view, safety is ensured by votes in either round. Since a replica only votes once for the first valid value it receives, by a quorum intersection argument, two conflicting values cannot both obtain *commitQC* when $f \leq t$.

Safety across views. Safety across views is ensured by the use of locks and the status certificate. First, observe that if a replica r outputs a value v in view e, then a quorum of replicas lock on (v, e). When $f \leq t$, this quorum includes a set H of at least $t + 1$ honest replicas. For any replica in H to update to a higher lock, they need a *prepareQC* in a higher view $e' > e$, which in turn requires a vote from at least one of these honest replicas in view e'. However, replicas in H will vote for a conflicting value v' in a higher view only if it is accompanied by a status certificate M containing $2t + 1$ locks that are not on value v. When $f \leq t$, honest replicas in M intersect with honest replicas in H at least one replica – this honest replica will not vote for a conflicting value v'.

4.2 Forensic Analysis for PBFT-PK

The agreement property for PBFT holds only when $f \leq t$. When the number of faults are higher, this agreement property (and even termination) can be violated. In this section, we show how to provide forensic support for PBFT when the agreement property is violated. We show that, if two honest replicas output conflicting values v and v' due to the presence of $t < f \leq m$ Byzantine replicas, our forensic protocol can detect $t + 1$ Byzantine replicas with an irrefutable proof. For each of the possible scenarios in which safety can be violated, the proof shows exactly what property of PBFT was not respected by the Byzantine replicas. The irrefutable proof explicitly uses messages signed by the Byzantine parties, and is thus only applicable to the variant PBFT-PK where all messages are signed.

Intuition. In order to build intuition, let us assume $n = 3t + 1$ and $f = t + 1$ and start with a simple scenario: two honest replicas output values v and v' in the same view. It must then be the case that a *commitQC* is formed for both v and v'. Due to a quorum intersection argument, it must be the case that all replicas in the intersection have voted for two conflicting values to break the uniqueness property. Thus, all the replicas in the intersection are culpable. For PBFT-PK, the *commitQC* (as well as *prepareQC*) for the two conflicting values act as the irrefutable proof for detecting $t + 1$ Byzantine replicas.

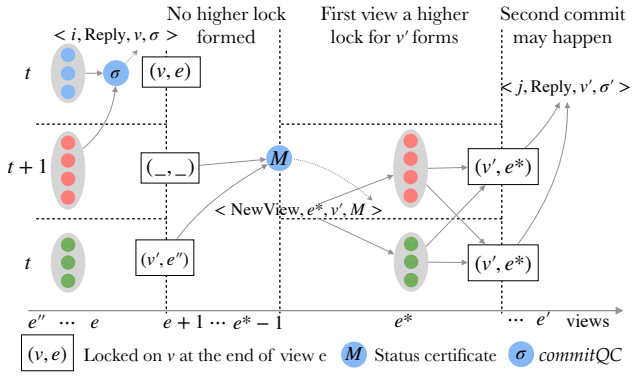

Figure 1: An example sequence of events in the PBFT-PK protocol that leads to replicas i and j outputting different values.

When two honest replicas output conflicting values in different views, there are many different sequences of events that could lead to such a disagreement. One such sequence is described in Figure 1. The replicas are split into three sets: the blue set and the green set are honest replicas each of size t and the red replicas are Byzantine replicas of size $t + 1$.

- In view e, replica i outputs v due to *commitQC* formed with the Commit from the honest blue set and the Byzantine red set. At the end of the view, replicas in the blue and red set hold locks (v, e) whereas the green set holds a lower lock for a different value.
- In the next few views, no higher locks are formed. Thus, the blue and the red set still hold locks (v, e).
- Suppose e^* is the first view where a higher lock is formed. At the start of this view, the leader receives locks from the honest green set who hold lower-ranked locks and the Byzantine red set who maliciously send lower-ranked locks. The set of locks received by the leader is denoted by M and suppose the highest lock was received for v'. The leader proposes v' along with M. This can make any honest replica "unlock" and vote for v' and form quorum certificates in this view.
- In some later view e', replica j outputs v'.

With this sequence of events, consider the following questions: (1) Is this an admissible sequence of events? (2) How do we find the culpable Byzantine replicas? What does the irrefutable proof consist of? (3) How many replica transcripts do we need to construct the proof?

To answer the first question, the only nontrivial part in our example is the existence of a view e^* where a higher lock is formed. However, such a view $e < e^* \leq e'$ must exist because replica j outputs in view e' and a higher lock must be formed at the latest in view e'.

For the second question, observe that both the red replicas as well as the green replicas sent locks lower than (v, e) to the leader in e^*. However, only the red replicas also sent Commit messages for value v in view e. Thus, by intersecting the set of Commit messages for value v in view e and the messages forming the status certificate sent to the leader of e^*, we can obtain a culpable set of $t + 1$ Byzantine replicas. So the proof for PBFT-PK consists of the *commitQC* in e and the status certificate in e^*, which indicates that the replicas sent a lower lock in view e^* despite having access to a higher lock in a lower view e.

For the third question, the NewView message containing the status certificate M in view e^* can act as the proof, so only one transcript needs to be stored.

Forensic protocol for PBFT-PK. Algorithm 1 describes the entire protocol to obtain forensic support atop PBFT-PK. For completeness, we also provide a complete description of the PBFT-PK protocol in Algorithm 5. Each replica keeps all received messages as transcripts and maintains a set Q containing all received NewView messages (line 2). If a client observes the replies of two conflicting values, it first checks if two values are output in the same view (line 9). If yes, then any two *commitQC* for two different output values can provide a culpability proof for at least $t + 1$ replicas (lines 19-22). Otherwise, the client sends a request for a possible proof between two output views e, e' (lines 13). Each replica looks through the set Q for the NewView message in the smallest view $e^* > e$ such that the status certificate M contains the highest lock (e'', v'') where $v'' \neq v$ and $e'' \leq e$ and sends it to the client (line 7). If inside M there are conflicting locks in the same view, the intersection of them proves at least $t + 1$ culprits (line 15), otherwise the intersection of M and the *commitQC* proves at least $t + 1$ culprits (line 18).

The following theorem sharply characterizes the forensic support capability of PBFT-PK. As long as $m \leq 2t$, the best possible forensic support is achieved (i.e., $k = 1$ and $d = t + 1$). Algorithm 1 can be used to irrefutably detect $t + 1$ Byzantine replicas. Conversely, if $m > 2t$ then no forensic support is possible (i.e., $k = n - f$ (messages from all honest nodes) and $d = 0$).

Algorithm 1 Forensic protocol for PBFT-PK Byzantine agreement

1: **as** a replica running PBFT-PK
2: $Q \leftarrow$ all NewView messages in transcript
3: **upon receiving** $\langle$Request-Proof, $e, v, e'\rangle$ from a client **do**
4: **for** $m \in Q$ **do**
5: $(v'', e'') \leftarrow$ the highest lock in $m.M$
6: **if** $(m.e \in (e, e']) \wedge (v'' \neq v) \wedge (e'' \leq e)$ **then**
7: send $\langle$NewView, $m\rangle$ to the client
8: **as** a client
9: **upon receiving** two conflicting Reply messages **do**
10: **if** the two messages are from different views **then**
11: $\langle$Reply, $e, v, \sigma\rangle \leftarrow$ the message from lower view
12: $e' \leftarrow$ the view number of Reply from higher view
13: broadcast $\langle$Request-Proof, $e, v, e'\rangle$
14: wait for: $\langle$NewView, $m\rangle$ s.t. $m.e \in (e, e'] \wedge (v'' \neq v) \wedge (e'' \leq e)$ where (v'', e'') is the highest lock in $m.M$.
15: **if** in $m.M$ there are two locks (e'', v_1, σ_1) and (e'', v_2, σ_2) s.t. $v_1 \neq v_2$ **then**
16: **output** $\sigma_1 \cap \sigma_2$
17: **else**
18: **output** the intersection of senders in $m.M$ and signers of σ.
19: **else**
20: $\langle$Reply, $e, v, \sigma\rangle \leftarrow$ first Reply message
21: $\langle$Reply, $e, v', \sigma'\rangle \leftarrow$ second Reply message
22: **output** $\sigma \cap \sigma'$

Theorem 4.1. *With $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, PBFT-PK provides $(2t,\ 1,\ t + 1)$-forensic support. Further $(2t + 1, n - f, d)$-forensic support is impossible with $d > 0$.*

Proof. We prove the forward part of the theorem below. The converse (impossibility) is proved in §C.1. Suppose the values v and v' are output in views e and e' respectively.

Case $e = e'$.
Culpability. The quorums *commitQC* for v and *commitQC* for v' intersect in $t + 1$ replicas. These $t + 1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can obtain the culpability proof based on two *commitQC*. No additional communication is needed in this case ($k = 0$).

Case $e \neq e'$.
Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since v is output in view e, it must be the case that $2t + 1$ replicas are locked on (v, e) at the end of view e (if they are honest). Now consider the first view $e < e^* \leq e'$ in which a higher lock (v'', e^*) is formed (not necessarily known to any honest party) where $v'' \neq v$ (possibly $v'' = v'$). Such a view must exist since v' is output in view $e' > e$ and a lock will be formed in at least view e'. Consider the status certificate M sent by the leader of view e^* in its NewView message. M must contain $2t + 1$ locks; each of these locks must be from view $e'' \leq e$, and a highest lock among them is (v'', e'').

We consider two cases based on whether the status certificate contains two different highest locks: (i) there exist two locks (v'', e'') and (v''', e'') s.t. $v'' \neq v'''$ in M. (ii) (v'', e'') is the only highest lock in M. For the first case, since two locks are formed in the same view, the two quorums forming the two locks in view e'' intersect in $t + 1$ replicas. These replicas are Byzantine since they voted for more than one value in view e.

For the second case, (v'', e'') is the only highest lock in the status certificate M. M intersects with the $2t + 1$ signers of *commitQC* in view e at $t + 1$ Byzantine replicas. These replicas are Byzantine because they had a lock on $v \neq v''$ in view $e \geq e''$ but sent a different lock to the leader of view $e^* > e$.
Witnesses. Client can obtain the proof by storing the NewView message containing the status certificate M in e^*. Only one witness is needed to provide the NewView message ($k = 1$). The status certificate M and the first *commitQC* act as the irrefutable proof. □

Communication complexity. In the first branch of the forensic protocol, Algorithm 1, the client needs to receive one message from $k = 1$ replica and the message size is $(2t + 1)(|v| + |sig|)$ where $|v|$ and $|sig|$ stand for the size of a value and an aggregate signature (line 14). In the second branch, the client doesn't need any message (line 19). Therefore the complexity of the client receiving messages is $O(n(|v| + |sig|))$. Notice that we exclude the communication for learning replica outputs (Reply messages) since that procedure happens *before* the forensic protocol.

4.3 Impossibility for PBFT-MAC

We now show an impossibility for a variant of PBFT proposed in [11, Section 5]. The arguments here also apply to the variant in [10]. Compared to §4.1, the only difference in this variant is (i) Prepare and Commit messages are authenticated using MACs instead of signatures, and (ii) these messages are broadcast instead of routing them through the leader.

Intuition. The key intuition behind the impossibility relies on the absence of digital signatures which were used to "log" the state of a replica when some replica i outputs a value. In particular, if we consider the example in Figure 1, while i receives $2t + 1$ Commit messages for value v, these messages are not signed. Thus, if $t + 1$ Byzantine replicas vote for a different value v', v' can be output by a different replica. The absence of a verifiable proof stating the set of replicas that sent a Commit to replica i prevents any forensic analysis. We formalize this intuition below.

Theorem 4.2. *With $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, $(t + 1, 2t, d)$-forensic support is impossible with $d > 0$ for PBFT-MAC.*

Proof. Suppose the protocol provides forensic support to detect $d \geq 1$ replicas with irrefutable proof. To prove this result, we construct two worlds where a different set of $t + 1$ replicas are Byzantine in each world but a forensic protocol cannot be correct in both worlds. We fix $f = t + 1$, although the arguments will apply for any $f > t$.

Let there be four replica partitions $P, Q, R, \{x\}$. $|P| = |Q| = |R| = t$, and x is an individual replica. In both worlds, the conflicting

outputs are presented in the same view e. Suppose the leader is a replica from set R.

World 1. Let P and x be Byzantine replicas in this world. The honest leader from set R in view e proposes v'. Parties in R, x and Q send Prepare and Commit messages (authenticated with MACs) for value v'. Due to partial synchrony, none of these messages arrive at P. At the end of view e, only R and one replica q in Q receive enough Commit messages and send replies to the client. So the client receives the first set of $t+1$ replies for value v', which contain the same quorum R, x, Q.

The Byzantine parties in P and x simulate a proposal from the leader for v, and the sending of Prepare and Commit messages within R, P and x. The simulation is possible due to the absence of a PKI. At the end of view e, P and x obtain enough Commit messages and send replies to the client. Thus, the client receives the second set of $t+1$ replies for value v, which contain the same quorum P, R, x.Client starts the forensic protocol.

During the forensic protocol, Byzantine P and x only present the votes for v, forged votes from R as their transcripts. Since $t+1$ parties have output each of v and v', there is a safety violation. Since the protocol has forensic support for $d \geq 1$, using these transcripts, the forensic protocol determines some subset of P and x are culpable.

World 2. Let R and q (one replica in Q) be Byzantine replicas in this world. The Byzantine leader in view e proposes v to P, R and x. They send Prepare and Commit messages (authenticated with MACs) for value v. These messages do not arrive at Q. At the end of view e, parties in P and x output v. So the client receives the first set of $t+1$ replies for value v, which contain the same quorum P, R, x.

Similarly, the leader sends v' to Q, R and x. The proposal does not arrive at x. Only Q and R send Prepare and Commit messages (authenticated with MACs) for v', these messages do not arrive at P. However, R and q forge Prepare and Commit messages from x. At the end of view e, R and q output v'. So the client receives the second set of $t+1$ replies for value v', which contain the same quorum R, x, Q. Client starts the forensic protocol.

During the forensic protocol, Byzantine R and q sends the same transcripts as in World 1 by dropping votes for v and forging votes from x. Again, since $t+1$ parties have output each of v and v', there is a safety violation. However, observe that the transcript presented to the forensic protocol is identical to that in World 1. Thus, the forensic protocol outputs some subset of P and x as culpable replicas. In World 2, this is incorrect since replicas in P and x are honest. This completes the proof.

□

5 FORENSIC SUPPORT FOR HOTSTUFF

HotStuff [32] is a partially synchronous consensus protocol that provides an optimal resiliency of $n = 3t + 1$. The HotStuff protocol is similar to PBFT but there are subtle differences which allow it to obtain a linear communication complexity for both its steady state and view change protocols (assuming the presence of threshold signatures). Looking ahead, these differences significantly change the way forensics is conducted if a safety violation happens.

5.1 Overview

We start with an overview of the protocol. For simplicity, we discuss a single-shot version of HotStuff. The protocol proceeds in a sequence of consecutive views where each view has a unique leader. Each view of HotStuff progresses as follows:[2]

- **Pre-prepare.** The leader proposes a NewView message containing a proposal v along with the *highQC* (the highest *prepareQC* known to it) and sends it to all replicas.
- **Prepare.** On receiving a NewView message containing a valid value v in a view e and a *highQC*, a replica sends Prepare for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote to the leader. The leader collects $2t + 1$ votes to form an aggregate signature *prepareQC* in view e. The leader sends the view e *prepareQC* to all replicas.
- **Precommit.** On receiving a *prepareQC* in view e containing message v, a replica updates its highest *prepareQC* to (v, e) and sends Precommit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *precommitQC*.
- **Commit.** On receiving *precommitQC* in view e containing message v from the leader, a replica locks on (v, e) and sends Commit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *commitQC*.
- **Reply.** On receiving *commitQC* from the leader, replicas output the value v and send a Reply (along with *commitQC*) to the client.

Once a replica locks on a given value v, it only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when it observes a *highQC* from a higher view in a NewView message. At the end of a view, every replica sends its highest *prepareQC* to the leader of the next view. The next view leader collects $2t + 1$ such values and picks the highest *prepareQC* as *highQC*. The safety and liveness of HotStuff when $f \leq t$ follows from the following:

Uniqueness within a view. Since replicas only vote once in each round, a *commitQC* can be formed for only one value when $f \leq t$.

Safety and liveness across views. Safety across views is ensured using locks and the voting rule for a NewView message. Whenever a replica outputs a value, at least $2t + 1$ other replicas are locked on the value in the view. Observe that compared to PBFT, there is no status certificate M in the NewView message to "unlock" a replica. Thus, a replica only votes for the value it is locked on. The only scenario in which it votes for a conflicting value v' is if the leader includes a *prepareQC* for v' from a higher view in NewView message. This indicates that at least $2t + 1$ replicas are not locked on v in a higher view, and hence it should be safe to vote for it. The latter constraint of voting for v' is not necessary for safety, but only for liveness of the protocol.

Variants of HotStuff. In this paper, we study three variants of HotStuff, identical for the purposes of consensus but provide varied forensic support. The distinction among them is only in the information carried in Prepare message. For all three versions, the message contains the message type Prepare, the current view number e and the proposed value v. In addition, Prepare in HotStuff-view contains e_{qc}, the view number of the *highQC* in the NewView

[2]The description of HotStuff protocol is slightly different from the basic algorithm described in [32, Algorithm 2] to be consistent with the description of PBFT in §4.1.

	HotStuff-view	HotStuff-hash	HotStuff-null
Info	e_{qc}	Hash(*highQC*)	$\emptyset$
m	$2t$	$2t$	$t+1$
k	1	$t+1$	$2t$
d	$t+1$	$t+1$	1

Table 3: Comparison of different variants of HotStuff, the PREPARE message is ⟨PREPARE, e, v, *Info*⟩

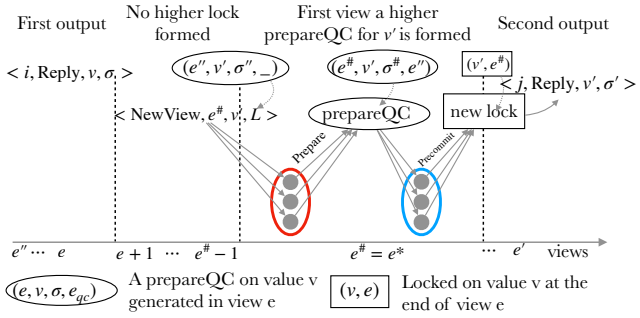

Figure 2: Depiction of events in the HotStuff-view protocol for the first view where a higher *prepareQC* for v' is formed.

message. HotStuff-hash contains the hash of *highQC* (cf. Table 3). HotStuff-hash is equivalent to the basic algorithm described in [32, Algorithm 2]. HotStuff-null does not add additional information.

5.2 Forensic Analysis for HotStuff

If two conflicting values are output in the same view, Byzantine replicas can be detected using *commitQC* and using ideas similar to that in PBFT. However, when the conflicting outputs of replicas i and j are across views e and e' for $e < e'$, the same ideas do not hold anymore. To understand this, observe that the two key ingredients for proving the culpability of Byzantine replicas in PBFT were (i) a *commitQC* for the value output in a lower view (denoted by σ for replica i's reply in Figure 1) and (ii) a status certificate from the first view higher than e containing the locks from $2t+1$ replicas (denoted by M for view $e^* > e$ in Figure 1). In HotStuff, a *commitQC* still exists. However, for communication efficiency reasons, HotStuff does not include a status certificate M in its NEWVIEW message. The status certificate in PBFT provides us with the following:

- **Identifying a potential set of culpable replicas.** Depending on the contents of M and knowing σ, we could identify a set of Byzantine replicas.
- **Determining whether the view is the first view where a higher lock for a conflicting value is formed.** By inspecting all locks in M, we can easily determine this. Ensuring first view with a higher lock is important; once a higher lock is formed, even honest replicas may update their locks and the proof of Byzantine behavior may not exist in the messages in subsequent views.

Let us try to understand this based on the first view $e^{\#}$ where a higher *prepareQC* is formed for $v' \neq v$ (see Figure 2). The set of replicas who sent PREPARE (the red ellipse) in $e^{\#}$ and formed a *prepareQC* are our potential set of Byzantine replicas. Why? If $e^{\#}$ is indeed the first view in which a higher *prepareQC* is formed, then all of these replicas voted for a NEWVIEW message containing a *highQC* from a lower or equal view e'' on a different value. If any of these replicas also held a lock (v, e) (by voting for replica i's output) then these replicas have output the culpable act of not respecting the voting rule.

The only remaining part is to ensure that this is indeed the first view where a higher conflicting *prepareQC* is formed. The way to prove this is also the key difference among three variants of HotStuff. For HotStuff-view, *prepareQC* contains e_{qc}, which explicitly states the view number of *highQC* in the NEWVIEW message they vote for. If $e_{qc} < e$, *prepareQC* provides an irrefutable proof for culpable behavior. For HotStuff-hash, the hash information contained in PREPARE provides the necessary link to the NEWVIEW message they vote, so once the linked NEWVIEW message is accessible, the *prepareQC* and NEWVIEW together serve as the proof for culpable behavior. However for HotStuff-null, even if we receive both *prepareQC* and NEWVIEW messages that are formed in the same view, no proof can be provided to show a connection between them. A Byzantine node vote for the first higher *prepareQC* can always refuse to provide the NEWVIEW message they receive.

Thus, to summarize, the red set of replicas in view $e^{\#}$ are a potential set of culpable nodes of size $t+1$. The irrefutable proof to hold them culpable constitutes two parts, (1) the first *prepareQC* containing their signed PREPARE messages, and (2) a proof to show this is indeed the first view. In the next two subsections we will introduce the forensic protocols for HotStuff-view and HotStuff-hash to provide different forensic supports, and how it is impossible for HotStuff-null to provide forensic support.

5.3 Forensic Protocols for HotStuff-view and HotStuff-hash

Forensic protocol for HotStuff-view. Algorithm 2 describes the protocol to obtain forensic support atop HotStuff-view. A complete description of the general HotStuff protocol is also provided in Algorithm 6. Each replica keeps all received messages as transcript and maintains a set P containing all received *prepareQC* from PRECOMMIT messages and *highQC* from NEWVIEW messages (line 2). If a client observes outputs of two conflicting values in the same view, it can determine the culprits using the two REPLY messages (line 15). Otherwise, the client sends a request to all replicas for a possible proof between two output views e, e' for $e < e'$ (line 12). Each replica looks through the set P for *prepareQC* formed in views $e < e^{\#} \leq e'$. If there exists a *prepareQC* whose value is different from the value v output in e and whose e_{qc} is less than or equal to e, it sends a reply with this *prepareQC* to the client (line 6). The client waits for a *prepareQC* (line 13) formed between two output views. For HotStuff-view, if it contains a different value from the first output value and an older view number $e_{qc} < e$, the intersection of this *prepareQC* and the *commitQC* from the REPLY message in the lower view proves at least $t+1$ culprits (line 14).

Algorithm 2 Forensic protocol for HotStuff-view

1: **as** a replica running HotStuff-view
2: $P \leftarrow$ all *prepareQC* in transcript ▷ including *prepareQC* in PRECOMMIT message and *highQC* in NEWVIEW message
3: **upon receiving** $\langle$REQUEST-PROOF, $e, v, e'\rangle$ from a client **do**
4: **for** $qc \in P$ **do**
5: **if** $(qc.v \neq v) \wedge (qc.e \in (e, e']) \wedge (qc.e_{qc} \leq e)$ **then**
6: send $\langle$PROOF-ACROSS-VIEW, $qc\rangle$ to the client
7: **as** a client
8: **upon receiving** two conflicting REPLY messages **do**
9: **if** two messages are from different views **then**
10: $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ the message from lower view
11: $e' \leftarrow$ the view number of message from higher view
12: broadcast $\langle$REQUEST-PROOF, $e, v, e'\rangle$
13: wait for $\langle$PROOF-ACROSS-VIEW, $qc\rangle$ s.t.
(1) $e < qc.e \leq e'$, and
(2) $(qc.v \neq v) \wedge (qc.e_{qc} \leq e)$
14: **output** $qc.\sigma \cap \sigma$
15: **else**
16: $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ first REPLY message
17: $\langle$REPLY, $e, v', \sigma'\rangle \leftarrow$ second REPLY message
18: **output** $\sigma \cap \sigma'$

Algorithm 3 Forensic protocol for HotStuff-hash

1: **as** a replica running HotStuff-hash
2: $P \leftarrow$ all *prepareQC* in transcript
3: $Q \leftarrow$ all NEWVIEW messages in transcript
4: **upon receiving** $\langle$REQUEST-PROOF, $e, v, e'\rangle$ from a client **do**
5: **for** $qc \in P$ **do**
6: **if** $(qc.v \neq v) \wedge (qc.e \in (e, e']) \wedge (qc.e_{qc} \leq e)$ **then**
7: send $\langle$PROOF-ACROSS-VIEW, $qc\rangle$ to the client
8: **for** $m \in Q$ **do**
9: **if** $(m.v \neq v) \wedge (m.e \in (e, e']) \wedge (m.highQC.e \leq e)$ **then**
10: send $\langle$NEWVIEW, $m\rangle$ to the client
11: **as** a client
12: $NV \leftarrow \{\}$
13: **upon receiving** two conflicting REPLY messages **do**
14: **if** two messages are from different views **then**
15: $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ the message from lower view
16: $e' \leftarrow$ the view number of message from higher view
17: broadcast $\langle$REQUEST-PROOF, $e, v, e'\rangle$
18: **upon receiving** $\langle$NEWVIEW, $m\rangle$ **do**
19: **if** $(m.v \neq v) \wedge (m.e \in (e, e']) \wedge (m.highQC.e \leq e)$ **then**
20: $NV \leftarrow NV \cup \{m\}$
21: wait for $\langle$PROOF-ACROSS-VIEW, $qc\rangle$ s.t.
(1) $e < qc.e \leq e'$, and
(2) $(qc.v \neq v) \wedge (\exists m \in NV, \text{Hash}(m) = qc.hash)$
22: **output** $qc.\sigma \cap \sigma$
23: **else**
24: $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ first REPLY message
25: $\langle$REPLY, $e, v', \sigma'\rangle \leftarrow$ second REPLY message
26: **output** $\sigma \cap \sigma'$

Forensic protocol for HotStuff-hash. Algorithm 3 describes the protocol to obtain forensic support atop HotStuff-hash, which is similar to the protocol for HotStuff-view. For replicas running HotStuff-hash, besides P, they also maintains the set Q for received NEWVIEW messages (line 3). When receiving a forensic request from clients, replicas look through P for *prepareQC* formed in views $e < e^{\#} \leq e'$ and send all *prepareQC* whose values are different from the value v to the client (line 7). Besides, they also look through Q for a NEWVIEW message formed in views $e < e^{\#} \leq e'$ and send all NEWVIEW proposing a value different from v and containing a *highQC* with view number $\leq e$ (line 10). For HotStuff-hash, when receiving such a NEWVIEW for different values, the message will be stored temporarily by the client until a *prepareQC* for the NEWVIEW message with a matching hash is received. The NEWVIEW and the *prepareQC* together form the desired proof; the intersection of the *prepareQC* and the *commitQC* provides at least $t + 1$ culprits.

Forensic proofs. The following theorems characterize the forensic support capability of HotStuff-view and HotStuff-hash. As long as $m \leq 2t$, HotStuff-view can achieve the best possible forensic support (i.e., $k = 1$ and $d = t + 1$). HotStuff-hash can achieve a medium forensic support (i.e., $k = t + 1$ and $d = t + 1$). Conversely, if $m > 2t$ then no forensic support is possible for both protocols (i.e., $k = n - f$ and $d = 0$).

THEOREM 5.1. *With $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, HotStuff-view provides $(2t,\ 1,\ t+1)$-forensic support. Further $(2t + 1, n - f, d)$-forensic support is impossible with $d > 0$.*

PROOF. We prove the forward part of the theorem below. The proof of converse (impossibility) is the same as §C.1. Suppose two conflicting values $v,\ v'$ are output in views e, e' respectively.

Case $e = e'$.
Culpability. The *commitQC* of v and *commitQC* of v' intersect in $t + 1$ replicas. These $t + 1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can obtain a proof based on the two REPLY messages, so additional witnesses are not necessary in this case.

Case $e \neq e'$.
Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since v is output in view e, it must be the case that $2t+1$ replicas are locked on (v, e) at the end of view e. Now consider the first view $e < e^* \leq e'$ in which a higher lock (v'', e^*) is formed where $v'' \neq v$ (possibly $v'' = v'$). Such a view must exist since v' is output in view $e' > e$ and a lock will be formed in at least view e'. For a lock to be formed, a higher *prepareQC* must be formed too. Consider the first view $e < e^{\#} \leq e'$ in which the corresponding *prepareQC* for v'' is formed. The leader in $e^{\#}$ broadcasts the NEWVIEW message containing a *highQC* on (v'', e''). Since this is the first time a higher *prepareQC* is formed and there is no higher *prepareQC* for v'' formed between view e and $e^{\#}$, we have $e'' \leq e$. The formation of the higher *prepareQC* indicates that $2t + 1$ replicas received the NEWVIEW message proposing v'' with *highQC* on (v'', e'') and consider it a valid proposal, i.e., the view number e'' is larger than their locks because the value is different.

Recall that the output value v indicates $2t+1$ replicas are locked on (v, e) at the end of view e. In this case, the $2t+1$ votes in *prepareQC* in view $e^{\#}$ intersect with the $2t+1$ votes in *commitQC* in view e at $t+1$ Byzantine replicas. These replicas should be Byzantine because they were locked on the value v in view e and vote for a value $v'' \neq v$ in a higher view $e^{\#}$ when the NewView message contained a *highQC* from a view $e'' \leq e$. Thus, they have violated the voting rule.

Witnesses. Client can obtain a proof by storing a *prepareQC* formed between e and e', whose value is different from v and whose $e_{qc} \leq e$. So only one witness is needed ($k = 1$), the *prepareQC* and the first *commitQC* act as the irrefutable proof. □

Theorem 5.2. *With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, HotStuff-hash provides $(2t,\ t+1,\ t+1)$-forensic support. Further $(2t+1, n-f, d)$-forensic support is impossible with $d > 0$.*

Proof. We prove the forward part of the theorem below. The proof of converse (impossibility) is the same as §C.1. Suppose two conflicting values $v,\ v'$ are output in views e, e' respectively.

Case $e = e'$. Same as Theorem 5.1.

Case $e \neq e'$.

Culpability. Same as Theorem 5.1.

Witnesses. Since *prepareQC* of HotStuff-hash only has the hash of *highQC*, the irrefutable proof contains the NewView message that includes the *highQC* and the corresponding *prepareQC* with the matching hash Hash(*highQC*). The client may need to store all NewView messages between e and e' whose value is different from v and the whose *highQC* is formed in $e_{qc} \leq e$, until receiving a *prepareQC* for some NewView message with a matching hash. In the best case, some replica sends both the NewView message and the corresponding *prepareQC* so the client only needs to store $k = 1$ replica's transcript. In the worst case, we can prove that any $t+1$ messages of transcript are enough to get the proof. Consider the honest replicas who receive the first *prepareQC* and the NewView message. $2t+1$ replicas have access to the *prepareQC* and $2t+1$ replicas have access to the NewView message. Among them at least $t+1$ replicas have access to both messages, and we assume they are all Byzantine. Then at least t honest replicas have the *prepareQC* and at least t honest replicas have the NewView message. The total number of honest replicas $n - f \leq 2t$. Thus among any $t+1$ honest replicas, at least one have NewView message and at least one have *prepareQC*. Therefore, $t+1$ transcripts from honest replicas ensure the access of both NewView message and *prepareQC* and thus guarantee the irrefutable proof.

□

Communication complexity. In line 13 of Algorithm 2, the client needs to receive one message from $k = 1$ replica and the message size is $(|v|+|sig|)$ where $|v|$ and $|sig|$ stand for the size of a value and an aggregate signature. Therefore the complexity of the client receiving messages is $O(|v|+|sig|)$ for HotStuff-view. As for HotStuff-hash, theorem 5.2 shows that in the worst case, the client needs to receive messages from $k = t+1$ replicas. Each of those replicas sends one message of size $O(|v|+|sig|+|hash|)$ where $|hash|$ stands for the size of a hash value. Therefore the complexity of the client receiving messages is $O(n(|v|+|sig|+|hash|))$ for HotStuff-hash.

5.4 Impossibility for HotStuff-null

Compared to the other two variants, in HotStuff-null, Prepare message and *prepareQC* are not linked to the NewView message. We show that this lack of information is sufficient to guarantee impossibility of forensics.

Intuition. When $f = t+1$, from the forensic protocols of HotStuff-view and HotStuff-hash, we know that given across-view $commitQC_1$ and $commitQC_2$ (ordered by view) and the first *prepareQC* higher than $commitQC_1$, the intersection of *prepareQC* and $commitQC_1$ contains at least $d = t+1$ Byzantine replicas. The intersection argument remains true for HotStuff-null, however, it is impossible for a client to decide whether *prepareQC* is the first one only with the transcripts sent by $2t$ honest replicas (when $f = t+1$). In an execution where there are two *prepareQC* in view e^* and e' respectively ($e^* < e'$), the Byzantine replicas (say, set P) may not respond with the *prepareQC* in e^*. The lack of information disallows a client from separating this world from another world P is indeed honest and sharing all the information available to them. We formalize this intuition in the theorem below.

Theorem 5.3. *With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, $(t+1,\ 2t,\ d)$-forensic support is impossible with $d > 1$ for HotStuff-null. Further, $(t+2,\ n-f,\ d)$-forensic support is impossible with $d > 0$.*

The theorem is proved in §C.2.

6 FORENSIC SUPPORT FOR ALGORAND

Algorand [14] is a synchronous consensus protocol which tolerates up to one-third fraction of Byzantine users. At its core, it uses a BFT protocol from [13, 24]. However, Algorand runs the protocol by selecting a small set of replicas, referred to as the committee, thereby achieving consensus with sub-quadratic communication. The protocol is also player replaceable, i.e., different steps of the protocol have different committees, thus tolerating an adaptive adversary. Each replica uses *cryptographic self-selection* using a verifiable random function (VRF) [25] to privately decide whether it is selected in a committee. The VRF is known only to the replica until it has sent a message to other parties thus revealing and proving its selection in the committee. In this section, we present an overview of the BFT protocol and then show why it is impossible to achieve forensic support for this protocol.

6.1 Overview

We start with an overview of the single-shot version of Algorand [14]. The protocol assumes synchronous communication, where messages are delivered within a known bounded time. The protocol proceeds in consecutive steps, each of which lasts for a fixed amount of time that guarantees message delivery. Each step has a self-selected committee, and a replica can compute its VRF, a value that decides whether it is selected in the committee, and it is known only by the replica itself until it has sent the value. All messages sent by a committee member is accompanied by a VRF thus allowing other replicas to verify its inclusion in the committee.

Parameters such as committee size κ are chosen such that the number of honest parties in the committee is greater than a threshold $t_H \geq 2\kappa/3$ with overwhelming probability.

The BFT protocol is divided into two sub-protocols: *Graded Consensus* and *BBA**. In *Graded Consensus*, which forms the first three steps of the protocol, each replica r inputs its value v_r. Each replica r outputs a tuple containing a value v_r^{out} and a grade g_r. In an execution where all replicas start with the same input v, $v_r^{\text{out}} = v$ and $g_r = 2$ for all replicas r. The replicas then enter the next sub-protocol, denoted *BBA**. If $g_r = 2$, replica r inputs value $b_r = 0$, otherwise it inputs $b_r = 1$. At the end of *BBA**, the replicas agree on the tuple $(0, v)$ or $(1, v_\perp)$.[3] The *BBA** sub-protocol also uses a random coin; for simplicity, we assume access to an ideal global random coin. Our forensic analysis in the next section only depends on *BBA** and thus, we only provide an overview for *BBA** here. The protocol proceeds in the following steps,

- Steps 1-3 are *Graded Consensus*. At the end of *Graded Consensus*, each replica inputs a value v_r and a binary value b_r to *BBA**.
- Step 4. Each replica in the committee broadcasts its vote for (b_r, v_r) along with its VRF.
- Step s ($s \geq 5, s \equiv 2 \mod 3$) is the Coin-Fixed-To-0 step of *BBA**. In this step, a replica checks *Ending Condition 0*: if it has received $\geq t_H$ valid votes on (b, v) from the previous step, where $b = 0$, it outputs (b, v) and ends its execution. Otherwise, it updates b_r as follows:

$$b_r \leftarrow \begin{cases} 1, & \text{if} \geq t_H \text{ votes on } b = 1 \\ 0, & \text{otherwise} \end{cases}$$

  If the replica is in the committee based on its VRF, it broadcasts its vote for (b_r, v_r) along with the VRF.
- Step s ($s \geq 6, s \equiv 0 \mod 3$). Symmetric to the previous step but for bit 1 instead of 0. Also, the votes need not be for the same v in the ending condition.
- Steps s ($s \geq 7, s \equiv 1 \mod 3$) is the Coin-Genuinely-Flipped step of *BBA**. In this step, it updates its variable b_r as follows:

$$b_r \leftarrow \begin{cases} 0, \text{if} \geq t_H \text{ votes on } b = 0 \\ 1, \text{if} \geq t_H \text{ votes on } b = 1 \\ \text{random coin of step } s, & \text{otherwise} \end{cases}$$

  If the replica is in the committee based on its VRF, it broadcasts its vote for (b_r, v_r) along with the VRF.

Safety of *BBA* within a step.* If all honest replicas reach an agreement before any step, the agreement will hold after the step. If the agreement is on binary value 0 (1 resp.) then the opposite Ending Condition 1 (0 resp.) will not be satisfied during the step. This is because synchronous communication ensures the delivery of at least t_H votes on the agreed value and there are not enough malicious votes on the other value.

Safety of *BBA* across steps.* For the step Coin-Fixed-To-0 (1 resp.), if any honest replica ends due to Ending Condition 0 (1 resp.), all honest replicas will agree on binary value 0 and value v (1 and $v_\perp$ resp.) at the end of the step, because there could only be less than t_H votes on binary value 1 (0 resp.). Hence, together with safety within a step, binary value 1 and value $v_\perp$ (0 and v resp.) will never be output.

[3] $v_\perp$ is considered external valid in Algorand.

6.2 Impossibility of Forensics

When the Byzantine fraction in the system is greater than one-third, with constant probability, a randomly chosen committee of size $\kappa < n$ will have $t_H < 2\kappa/3$. In such a situation, we can have a safety violation. Observe that since only $\kappa < n$ committee members send messages in a round, the number of culpable replicas may be bounded by $O(\kappa)$. However, we will show an execution where no Byzantine replica can be held culpable.

Intuition. The safety condition for *BBA** relies on the following: if some honest replica commits to a value b, say $b = 0$, in a step and terminates, then all honest replicas will set $b = 0$ as their local value. In all subsequent steps, there will be sufficient ($> 2/3$ fraction) votes for $b = 0$ due to which replicas will never set their local value $b = 1$. Thus, independently of what Byzantine replicas send during the protocol execution, honest replicas will only commit on $b = 0$. On the other hand, if replicas do not receive $> 2/3$ fraction of votes for $b = 0$, they may switch their local value to $b = 1$ in the Coin-Fixed-To-1 or Coin-Genuinely-Flipped step. This can result in a safety violation. When the Byzantine fraction is greater than one-third, after some replicas have committed 0, the Byzantine replicas can achieve the above condition by selectively not sending votes to other replicas (say set Q), thereby making them switch their local value to $b = 1$. There is no way for an external client to distinguish this world from another world where the set Q is Byzantine and states that it did not receive these votes. We formalize this intuition in the theorem below. Observe that our arguments work for the *BBA** protocol with or without player-replaceability.

Theorem 6.1. *When the Byzantine fraction exceeds 1/3, if two honest replicas output conflicting values, $(t+1,\ 2t,\ d)$-forensic support is impossible with $d > 0$ for Algorand.*

The theorem is proved in §C.3.

7 LIBRABFT AND DIEM

In this paper, we have focused on forensics for single-shot consensus. Chained BFT protocols are natural candidates for consensus on a sequence with applications to blockchains. LibraBFT is a chained version of Hotstuff and is the core consensus protocol in Diem, a new cryptocurrency supported by Facebook [5]. In this section, we show that LibraBFT has the strongest forensic support possible (as in Hotstuff-view). Further, we implement the corresponding forensic analysis protocol as a module on top of an open source Diem client. We highlight the system innovations of our implementation and associated forensic dashboard; this has served as a reference implementation presently under active consideration for deployment (due to anonymity imperativs we are unable to document this more concretely).

Diem **blockchain.** Diem Blockchain uses LibraBFT [30], a variant of HotStuff protocol for consensus. In LibraBFT, the replicas are called validators, who receive transactions from clients and propose blocks of transactions in a sequence of rounds.

	HotStuff-hash	LibraBFT
Prepare	⟨Prepare, e, v, Hash($highQC$)⟩	⟨Prepare, e, Hash($b = (v, highQC)$)⟩
k	$t+1$	1
m	$2t$	$n-2$
extra condition	-	must receive the preimage of hash

Table 4: Comparison of HotStuff-hash and LibraBFT

LibraBFT forensics. The culpability analysis for LibraBFT is similar to Theorem 5.2. However, for the witnesses, the blockchain property of LibraBFT makes sure that any replica (validator) has access to the full blockchain and thus provides $(n-2, 1, t+1)$-forensic support. The formal result is below (proof in §C).

Theorem 7.1. *For $n = 3t+1$, when $f > t$, if two honest replicas output conflicting blocks, LibraBFT provides $(n-2,\ 1,\ t+1)$-forensic support.*

The aforementioned three variants of HotStuff in §5 are described under the VBA setting to reach consensus on a single value, and the value can be directly included in the vote message (cf. Table 4, v is contained in the Prepare message). In this setting, once a replica receives the *commitQC* for the value, it will output the value and send a reply to the client, even if the *commitQC* is the only message it receives in the current view so far. So when two honest replicas output conflicting values, it is possible that the client receives only the commit messages and extra communication is needed. And when $m > 2t$, Byzantine replicas are able to form QCs by themselves so that no other honest replicas can get access to the first *prepareQC*. Thus the bound on m for HotStuff-view and HotStuff-hash is $2t$.

However, the setting is slightly different in practice, when the value v is no longer a single value, but actually a block with more fields and a list of transactions/commands, as in LibraBFT. In single-shot consensus, a block includes the transactions (value) and the *highQC*. In this case, the block is too heavy to include directly in a vote message, so the replicas add the hash of the block to the vote message (see Table 4, Hash($b = (v, highQC)$) is contained in the Prepare message). And since only the NewView message has the block's preimage, replicas cannot vote/output until receiving the original blocks. Thus when two honest replicas output conflicting values, the client can obtain the full blockchain from one of them ($k = 1$) and all *prepareQC* are part of the blocks. In this case, even if $m > 2t$ the client can still enjoy non-trivial forensic support.

Forensic module. Our prototype consists of two components, a database Forensic Storage used to store quorum certificates received by validators, which can be accessed by clients through JSON-RPC requests or consensus API; and an independent Detector run by clients to analyze the forensic information.

- **Forensic Storage** maintains a map from the view number to quorum certificates and its persistent storage. It is responsible for storing forensic information and allows access by other components, including clients (via JSON-RPC requests or consensus API).

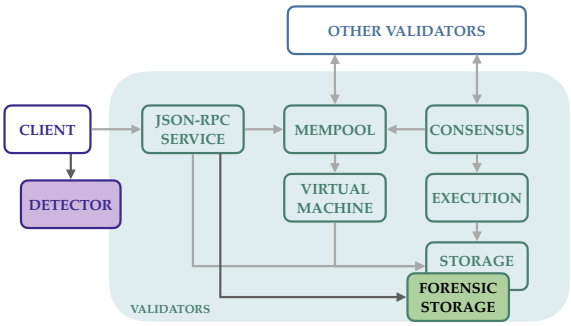

Figure 3: Forensic module integrated with Diem.

- **Detector** is run by clients manually to send requests periodically to connected validators. It collates information received from validators, using it as the input to the forensic analysis protocol.

Testing using Twins [6]. To test the correctness of forensic protocols, we build a testbed to simulate Byzantine attacks and construct different types of safety violations. Ideally, for modularity purposes, our testbed should not require us to modify the underlying consensus protocol to obtain Byzantine behavior. We leverage Twins [6], an approach to emulate Byzantine behaviors by running two instances of a node (i.e. replica) with the same identity. Consider a simple example setting with four nodes (denoted by $node0 \sim 3$), where $node0$ and $node1$ are Byzantine so they have twins called $twin0$ and $twin1$. The network is split into two partitions, the first partition P_1 includes nodes $\{node0, node1, node2\}$ and the second partition P_2 includes nodes $\{twin0, twin1, node3\}$. Nodes in one partition can only receive the messages sent from the same partition. The double voting attack can be simulated when Byzantine leader proposes different valid blocks in the same view, and within each partition, all nodes will vote for the proposed block. The network partition is used to drop all messages sent from a set of nodes. However, it can only help construct the safety violation within the view. To construct more complicated attacks, we further improve the framework and introduce another operation called "detailed drop", which drops selected messages with specific types.

Visualization. The Detector accepts the registration of different views to get notified once the data is updated. We built a dashboard to display the information received by the detector and the analysis results output by the forensic protocol. Figure 4 shows a snapshot of the dashboard which displays information about the network topology, hashes of latest blocks received at different validators, conflicting blocks, detected culprit keys and raw logs. Interactions with end-users, including Diem core-devs, has guided our design of the dashboard.

8 CONCLUSION

In this paper, we have embarked on a systematic study of the forensic properties of BFT protocols, focusing on 4 canonical examples: PBFT (classical), Hotstuff and VABA (state-of-the-art protocols on partially synchronous and asynchronous network settings) and Algorand (popular protocol that is adaptable to proof of stake

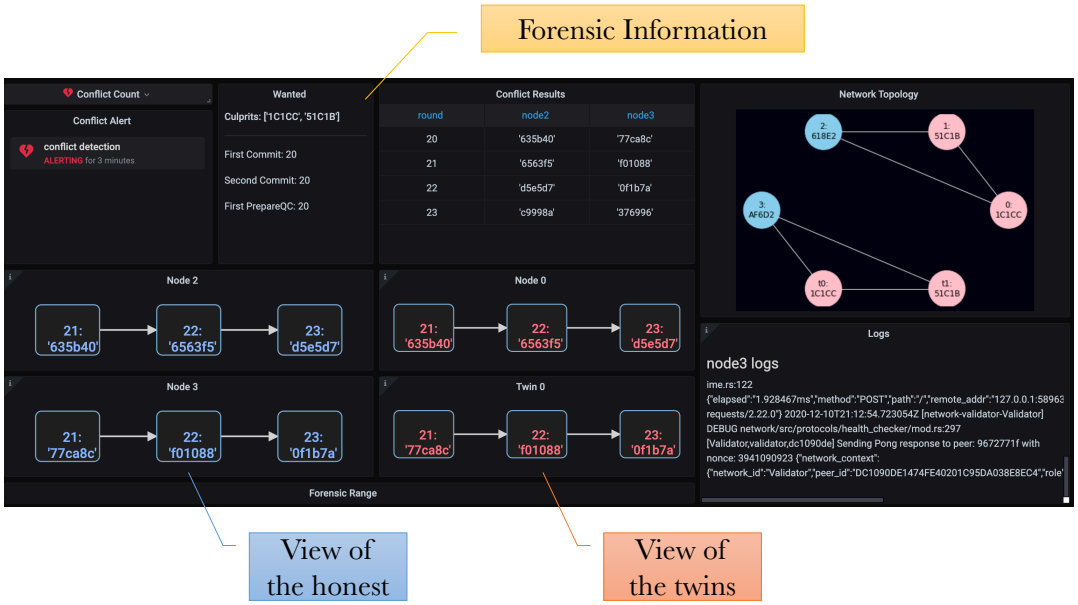

Figure 4: Forensic module dashboard.

blockchains). Our results show that minor variations in the BFT protocols can have outsized impact on their forensic support.

We exactly characterize the forensic support of each protocol, parameterized by the triplet (m, k, d). The forensic support characterizations are remarkably similar across the protocols: if any non-trivial support is possible (i.e., at least one culpable replica can be implicated; $d > 0$), then the largest possible forensic support, $(2t, 1, t + 1)$, is also possible; the one exception to this result is the Hotstuff-hash variant. Although the proof of forensic support is conducted for each protocol and its variant individually, we observe common trends:

- For each of the protocols with strong forensic support, as a part of the protocol execution, there exist witnesses who hold signed messages from Byzantine parties indicating that they have not followed some rule in the protocol.
- On the other hand, for protocols with no forensic support, the Byzantine parties are able to break safety without leaving any evidence, although the mechanism to achieve this is different for each of PBFT-MAC, Algorand, and HotStuff-null. With PBFT-MAC, Byzantine parties are able to construct arbitrary transcripts due to the absence of signatures. Hence, message transcripts cannot be used as evidence. With Algorand, they can utilize a rule which relies on the absence of messages (under synchrony) to set an incorrect protocol state without leaving a trail. With HotStuff-null, due to the lack of links between messages across views, Byzantine parties can present fake message transcripts and thus, pretend to be honest.

Conceptually, the burning question is whether these common ingredients can be stitched together to form an overarching theory of forensic support for abstract families of secure BFT protocols: First, from an impossibility standpoint, is there a relationship between the need to use synchrony or the absence of PKI in a protocol and absence of forensic support? Second, for the positive results, can one argue strong forensic support for an "information-complete" variant of any BFT protocol? This is an active area of research.

From a practical stand point, forensic analysis for existing blockchain protocols is of great interest; our forensic protocol for LibraBFT and its reference implementation has made strong inroads towards practical deployment. However, one shortcoming of the approach in this paper is that forensic analysis is conducted only upon *fatal* safety breaches. It is of great interest to conduct forensics with other forms of attacks: liveness attacks, censorship, small number of misbehaving replicas that impact performance. We note that liveness attacks do not afford, at the outset, undeniable complicity of malicious replicas and an important research direction is in formalizing weaker notions of culpability proofs (perhaps including assumptions involving external forms of trust). This is an active area of research.

9 ACKNOWLEDGEMENT

This research was partly supported by US Army Research Office Grant W911NF-18-1-0332, National Science Foundation CCF-1705007 and the XDC network. Kartik Nayak was supported in part by Novi and VMware gift research grant.

We thank the Diem team for implementation advice and Jovan Komatovic for helpful discussions about a forensic attack on HotStuff.

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2019. Sync hotstuff: Simple and practical synchronous state machine replication. *IACR Cryptology ePrint Archive* 2019 (2019), 270.

[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 337–346.

[3] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2020. Optimal good-case latency for byzantine broadcast and state machine replication. *arXiv preprint arXiv:2003.13155* (2020).

[4] Anonymized. 2020. Forensic Module for Diem. https://github.com/BFTForensics/DiemForensics

[5] Diem Association. 2020. Diem. https://github.com/diem/diem

[6] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. 2020. Twins: White-Glove Approach for BFT Testing. *arXiv preprint arXiv:2004.10617* (2020).

[7] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 435–464.

[8] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *arXiv preprint arXiv:1710.09437* (2017).

[9] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

[10] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[11] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[12] Benjamin Y Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. *IACR Cryptol. ePrint Arch.* 2020 (2020), 88.

[13] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. 2018. ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement. *IACR Cryptol. ePrint Arch.* 2018 (2018), 377.

[14] Jing Chen and Silvio Micali. 2019. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science* 777 (2019), 155–183.

[15] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2019. Polygraph: Accountable Byzantine Agreement. *IACR Cryptol. ePrint Arch.* 2019 (2019), 587.

[16] Wikipedia contributors. 2020. "The Day After — Wikipedia, The Free Encyclopedia". https://en.wikipedia.org/wiki/The_Day_After Online; accessed 23 September 2020.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.

[19] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

[20] Andreas Haeberlen and Petr Kuznetsov. 2009. The fault detection problem. In *International Conference On Principles Of Distributed Systems*. Springer, 99–114.

[21] Daniel Kane, Andreas Fackler, Adam Gągol, Damian Straszak, and Vlad Zamfir. 2021. Highway: Efficient Consensus with Flexible Finality. *arXiv preprint arXiv:2101.02159* (2021).

[22] Jinyuan Li and David Maziéres. 2007. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems.. In *NSDI*.

[23] Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2019. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1041–1053.

[24] Silvio Micali. 2018. Byzantine agreement, made trivial.

[25] Silvio Micali, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*. IEEE, 120–130.

[26] Joachim Neu, Ertem Nusret Tas, and David Tse. 2020. Ebb-and-Flow Protocols: A Resolution of the Availability-Finality Dilemma. *arXiv preprint arXiv:2009.04987* (2020).

[27] HariGovind V Ramasamy and Christian Cachin. 2005. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference on Principles of Distributed Systems*. Springer, 88–102.

[28] Alejandro Ranchal-Pedrosa and Vincent Gramoli. 2020. Blockchain Is Dead, Long Live Blockchain! Accountable State Machine Replication for Longlasting Blockchain. *arXiv preprint arXiv:2007.10541* (2020).

[29] Alistair Stewart and Eleftherios Kokoris-Kogia. 2020. GRANDPA: a Byzantine finality gadget. *arXiv preprint arXiv:2007.01560* (2020).

[30] The LibraBFT Team. 2020. State Machine Replication in the Diem Blockchain. https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/

[31] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2021. Strengthened Fault Tolerance in Byzantine Fault Tolerant Replication. *arXiv preprint arXiv:2101.03715* (2021).

[32] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

A VABA HAS STRONG FORENSIC SUPPORT

Validated Asynchronous Byzantine Agreement (VABA) is a state-of-the-art protocol [2] in the asynchronous setting with asymptotically optimal $O(n^2)$ communication complexity and expected $O(1)$ latency for $n \geq 3t + 1$.

A.1 Overview

At a high-level, the VABA protocol adapts HotStuff to the asynchronous setting. There are three phases in the protocol:

- **Proposal promotion.** In this stage, each of the n replicas run n parallel HotStuff-like instances, where replica i acts as the leader within instance i.

Algorithm 4 Forensic protocol for VABA

as a replica running VABA
  for $e \geq 1$ **initialize**:
    for $j \in [n]$ **do**
      $ledger[e][j] \leftarrow \{\}$
      $coin[e] \leftarrow \{\}$
  upon receiving $\langle i, \text{NewView}, e, v, L\rangle$ in view e in Proposal-Promotion instance i **do**
    $(e', v', \sigma, e_{qc}) \leftarrow L$ ▷ Note that L has selectors e, v, σ, e_{qc}
    $ledger[e'][i] \leftarrow ledger[e'][i] \cup \{(v', \sigma, e_{qc})\}$
  upon receiving $\langle i, \text{Prepare}, e, v, \sigma, e_{qc}\rangle$ in view e in Proposal-Promotion instance i **do**
    $ledger[e][i] \leftarrow ledger[e][i] \cup \{(v, \sigma, e_{qc})\}$
  if $Leader[e]$ is elected **then**
    discard $ledger[e][j]$ for $j \neq Leader[e]$
    $coin[e] \leftarrow$ inputs to *threshold-coin* for electing $Leader[e]$
  upon receiving $\langle \text{ViewChange}, e, prepareQC, precommitQC, commitQC\rangle$ in view e **do**
    $(e', v', \sigma, e_{qc}) \leftarrow prepareQC$ ▷ Note that $prepareQC$ has selectors e, v, σ, e_{qc}
    $ledger[e'][Leader[e']] \leftarrow ledger[e'][Leader[e']] \cup \{(v', \sigma, e_{qc})\}$
  upon receiving $\langle \text{Request-Proof-of-Leader}, e, e'\rangle$ from a client **do**
    for all $e \leq e^* \leq e'$ **do**
      send $\langle \text{Proof-of-Leader}, e^*, Leader[e^*], coin[e^*]\rangle$ to client
  upon receiving $\langle \text{Request-Proof}, e, v, \sigma, e'\rangle$ with a collection of $LeaderMsg$ from a client **do**
    for all $e < e^{\#} \leq e'$ **do**
      if $Leader[e^{\#}]$ is not elected yet **then**
        $\langle \text{Proof-of-Leader}, e^{\#}, leader, coin\rangle \leftarrow LeaderMsg$ of view $e^{\#}$
        check $leader$ is the leader generated by $coin$ in view $e^{\#}$ (otherwise don't reply to the client)
        $Leader[e^{\#}] \leftarrow leader$
      for $qc \in ledger[e^{\#}][Leader[e^{\#}]]$ **do**
        if $(qc.v \neq v) \wedge (qc.e_{qc} \leq e)$ **then**
          send $\langle \text{Proof-across-View}, e^{\#}, Leader[e^{\#}], qc\rangle$ to the client

Algorithm 4 Forensic protocol for VABA

29: **as** a client
30: **upon receiving** two conflicting Reply messages **do**
31: $e \leftarrow$ the view number of Reply from lower view
32: $e' \leftarrow$ the view number of Reply from higher view
33: **for** all $e \le e^* \le e'$ **initialize**:
34: $Leader[e^*] \leftarrow \{\}$
35: $LeaderMsg[e^*] \leftarrow \{\}$
36: send $\langle$Request-Proof-of-Leader, $e, e'\rangle$ to the replica of Reply message from higher view
37: **for** all $e \le e^* \le e'$ **do**
38: wait for $\langle$Proof-of-Leader, $e^*, leader, coin\rangle$ s.t. $leader$ is the leader generated by $coin$ in view e^* (otherwise the Reply message is not considered valid)
39: $Leader[e^*] \leftarrow leader$
40: $LeaderMsg[e^*] \leftarrow \langle$Proof-of-Leader, $e^*, leader, coin\rangle$
41: **if** the two Reply messages are from different views **then**
42: $\langle i, \text{Reply}, e, v, \sigma\rangle \leftarrow$ the message from lower view
43: $\langle i', \text{Reply}, e', v', \sigma'\rangle \leftarrow$ the message from higher view
44: check $i = Leader[e]$ and $i' = Leader[e']$ (otherwise the Reply message is not considered valid)
45: broadcast $\langle$Request-Proof, $e, v, \sigma, e'\rangle$ with $LeaderMsg[e^*]$ for all $e < e^* \le e'$
46: wait for $\langle$Proof-across-View, $e^\#, leader, qc\rangle$ s.t.
(1) $e < e^\# \le e'$, and
(2) $(qc.v \neq v) \wedge (qc.e_{qc} \le e)$, and
(3) $leader = Leader[e^\#]$
47: **output** $qc.\sigma \cap \sigma$
48: **else**
49: $\langle i, \text{Reply}, e, v, \sigma\rangle \leftarrow$ first Reply message
50: $\langle i', \text{Reply}, e, v', \sigma'\rangle \leftarrow$ second Reply message
51: check $i = i' = Leader[e]$ (otherwise the Reply message is not considered valid)
52: **output** $\sigma \cap \sigma'$

- **Leader election.** After finishing the previous stage, replicas run a leader election protocol using a *threshold-coin* primitive [9] to randomly elect the leader of this view, denoted as $Leader[e]$ where e is a view number. At the end of the view, replicas adopt the "progress" from $Leader[e]$'s proposal promotion instance, and discard values from other instances.
- **View change.** Replicas broadcast quorum certificates from $Leader[e]$'s proposal promotion instance and update local variables and/or output value accordingly.

Within a proposal promotion stage, the guarantees provided are the same as that of HotStuff, and hence we do not repeat it here. The leader election phase elects a unique leader at random – this stage guarantees (i) with $\ge 2/3$ probability, an honest leader is elected, and (ii) an adaptive adversary cannot stall progress (since a leader is elected in hindsight). Finally, in the view-change phase, every replica broadcasts the elected leader's quorum certificates to all replicas.

A.2 Forensic Support for VABA

The key difference between forensic support for HotStuff and VABA is the presence of the leader election stage – every replica/client needs to know *which* replica was elected as the leader in each view. Importantly, the *threshold-coin* primitive ensures that there is a unique leader elected for each view. Thus, the forensic analysis boils down to performing an analysis similar to the HotStuff protocol, except that the leader of a view is described by the leader election phase.

We present the full forensic protocol in Algorithm 4 for completeness. We make the following changes to VABA:

- **Storing information for forensics.** Each replica maintains a list of *ledgers* for all instances, containing all received *prepareQC* from Prepare messages, NewView messages, and ViewChange messages (lines 8,10,16). When the leader of a view is elected, a replica keeps the *ledger* from the leader's instance and discards others (line 12). A replica also stores the random coins from the leader election phase for client verification (line 13).
- **Bringing proposal promotion closer to HotStuff-view.** There are minor differences in the proposal promotion phase of VABA [2] to the description in HotStuff (§5). We make this phase similar to that in the description of our HotStuff protocol with forensic support. In particular: (i) the *LOCK* variable stores both the view number and the value (denoted by $LOCK.e$ and $LOCK.v$), (ii) the voting rule in a proposal promotion phase is: vote if KEY has view and value equal to $LOCK$, except when KEY's view is strictly higher than $LOCK.e$, (iii) assume a replica's own ViewChange message arrives first so that others' ViewChange messages do not overwrite local variables KEY and $LOCK$, and (iv) add e_{qc} into Prepare.

A client first verifies leader election (lines 36-40). Then, it follows steps similar to the HotStuff forensic protocol (lines 41-52) except that there are added checks pertaining to leader elections (lines 44,46,51).

We prove the forensic support in Theorem A.1.

Theorem A.1. *For $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, VABA protocol provides $(2t,\ 1,\ t{+}1)$-forensic support. Further $(2t + 1,\ n - f,\ d)$-forensic support is impossible with $d > 0$.*

Proof. We prove the forward part of the theorem below. The proof of converse (impossibility) is the same as §C.1.

The leader of a view is determined by the threshold coin-tossing primitive *threshold-coin* and Byzantine replicas cannot forge the result of a leader election by the robustness property of the threshold coin. Suppose two conflicting outputs happen in view e, e' with $e \le e'$. The replica who outputs in view e' has access to the proof of leader election of all views $\le e'$. Therefore, a client can verify the leader election when it receives messages from this replica. Even if other replicas have not received messages corresponding to the elections in views $\le e'$, the client can send the proof of leader to them. The remaining forensic support proof follows from Theorem 5.1 in a straightforward manner, where any witness will

receive the proof of leader from the client (if leader is not elected) and send the proof of culprits to the client. □

Communication complexity. The client needs to first receive all leader election results from view e to view e', and each result is of size $|coin|$ (the size of the coin in the *threshold-coin* primitive). Then, the client shares leader election results with all replicas. This step incurs receiving message complexity $O(l|coin|)$ where $l = e' - e$. Next, the client needs to receive one message from $k = 1$ replica and the message size is $(|v| + |sig|)$. Therefore the complexity for the client receiving messages is $O(|v| + |sig| + l|coin|)$. However, the procedure of sharing leader election is irrelevant to forensic support, and we could assign it to replicas. (This procedure is included in the forensic protocol because we do not want to change the consensus protocol itself.) In that case, the client needs to receive just one leader election result, so the receiving message complexity is $O(|v| + |sig| + |coin|)$.

B IMPOSSIBILITY OF FORENSIC SUPPORT FOR $n = 2t + 1$

A validated Byzantine agreement protocol allows replicas to obtain agreement, validity, and termination so far as the actual number of faults $f \leq t$ where t is a Byzantine threshold set by the consensus protocol. A protocol that also provides forensic support with parameters m and d allows the detection of d Byzantine replicas when $\leq m$ out of n replicas are Byzantine faulty. In particular, in §4 and §5, we observed that when $t = \lfloor n/3 \rfloor,\ m = 2t$, and $k = 1$, we can obtain $(2t, 1, d)$-forensic support for $d = t + 1$. This section presents the limits on the number of Byzantine replicas detected (d), given the total number of Byzantine faulty replicas available in the system (m). In particular, we show that if the total number of Byzantine faults are too high, in case of a disagreement, the number of corrupt (Byzantine) replicas that can be deemed undeniably culpable will be too few.

Intuition. To gain intuition, let us consider a specific setting with $n = 2t + 1$, $m = n - t = t + 1$, and $d > 1$. Thus, such a protocol provides us with agreement, validity, and termination if the Byzantine replicas are in a minority. If they are in the majority, the protocol transcript provides undeniable guilt of more than one Byzantine fault. We show that such a protocol does not exist. Why? Suppose we split the replicas into three groups P, Q, and R of sizes t, t, and 1 respectively. First, observe that any protocol cannot expect Byzantine replicas to participate in satisfying agreement, validity, and termination. Hence, if the replicas in Q are Byzantine, replicas in $P \cup R$ may not receive any messages from Q. However, if, in addition, the replica R is also corrupt, then $R \cup Q$ can separately simulate another world where P are Byzantine and not sending messages, and $Q \cup R$ output a different value. Even if an external client obtains a transcript of the entire protocol execution (i.e., transcripts of $k = n - f$ honest replicas and f Byzantine replicas), the only replica that is undeniably culpable is R since it participated in both worlds. For all other replicas, neither P nor Q have sufficient information to prove the other set's culpability. Thus, an external client will not be able to detect more than one Byzantine fault correctly. Our lower bound generalizes this intuition to hold for $n > 2t$, $m = n - t$, $k = n - f$, and $d > n - 2t$.

Theorem B.1. *For any validated Byzantine agreement protocol with $t < n/2$, when $f > t$, if two honest replicas output conflicting values, $(n - t,\ n - f,\ d)$-forensic support is impossible with $d > n - 2t$.*

Proof. Suppose there exists a protocol that achieves agreement, validity, termination, and forensic support with parameters $n,\ t < n/2,\ m = n - t,\ k = n - f$ and $d > n - 2t$. Through a sequence of worlds, and through an indistinguishability argument we will show the existence of a world where a client incorrectly holds at least one honest replica as culpable. Consider the replicas to be split into three groups $P,\ Q$, and R with t, t, and $n - 2t$ replicas respectively. We consider the following sequence of worlds:

World 1. [t Byzantine faults, satisfying agreement, validity, and termination]
Setup. Replicas in P and R are honest while replicas in Q have crashed. P and R start with a single externally valid input v_1. All messages between honest replicas arrive instantaneously.
Output. Since there are $|Q| = t$ faults, due to agreement, validity and termination properties, replicas in P and R output v_1. Suppose replicas in P and R together produce a transcript T_1 of all the messages they have received.

World 2. [t Byzantine faults, satisfying agreement, validity, and termination]
Setup. Replicas in Q and R are honest while replicas in P have crashed. Q and R start with an externally valid input v_2. All messages between honest replicas arrive instantaneously.
Output. Since t replicas are Byzantine faulty, due to agreement, validity and termination properties, replicas in Q and R output v_2. Suppose replicas in Q and R together produce a transcript T_2 of all the messages they have received.

World 3. [$n - t$ Byzantine faults satisfying validity, termination, forensic support]
Setup. Replicas in P are honest while replicas in Q and R are Byzantine. Replicas in P start with input v_1. Replicas in Q and R have access to both inputs v_1 and v_2. Q behaves as if it starts with input v_2 whereas R will use both inputs v_1 and v_2. Replicas in Q and R behave with P exactly like in World 1. In particular, replicas in Q do not send any message to any replica in P. Replicas in R perform a split-brain attack where one brain interacts with P as if the input is v_1 and it is not receiving any message from Q. Also, separately, replicas in Q and the other brain of R start with input v_2 and communicate with each other exactly like in World 2. They ignore messages arriving from P.
Output. For replicas in P, this world is indistinguishable from that of World 1. Hence, they output v_1. Replicas in P and the first brain of R output transcript T_1 corresponding to the output. Replicas in Q and the other brain of R behave exactly like in World 2. Hence, they can output transcript T_2. Since the protocol provides $(n - t, n - f, d)$-forensic support for $d > n - 2t$, the transcript of messages should hold $d > n - 2t$ Byzantine replicas undeniably corrupt. Suppose the client can find the culpability of $> n - 2t$ replicas from $Q \cup R$, i.e., ≥ 1 replica from Q.

World 4. [$n - t$ Byzantine faults satisfying validity, termination, forensic support]

Setup. Replicas in Q are honest while replicas in P and R are Byzantine. Replicas in Q start with input v_2. Replicas in P and R have access to both inputs v_1 and v_2. P behaves as if it starts with input v_1 whereas replicas in R use both v_1 and v_2. Replicas in P and R behave with Q exactly like in World 2. In particular, replicas in P do not send any message to any replica in Q. Replicas in R perform a split-brain attack where one brain interacts with Q as if the input is v_2 and it is not receiving any message from P. Also, separately, replicas in P and the other brain of R start with input v_1 and communicate with each other exactly like in World 1. They ignore messages arriving from Q.

Output. For replicas in Q, this world is indistinguishable from that of World 2. Hence, they output v_2. Replicas in Q and the first brain of R output transcript T_2 corresponding to the output. Replicas in P and the other brain of R behave exactly like in World 1. Hence, they can output transcript T_1.

Observe that the transcript and outputs produced by replicas in P, Q, and R are exactly the same as in World 3. Hence, the client will hold $> n - 2t$ replicas from $Q \cup R$, i.e., ≥ 1 replica from Q as culpable. However, all replicas in Q are honest in this world. This is a contradiction. □

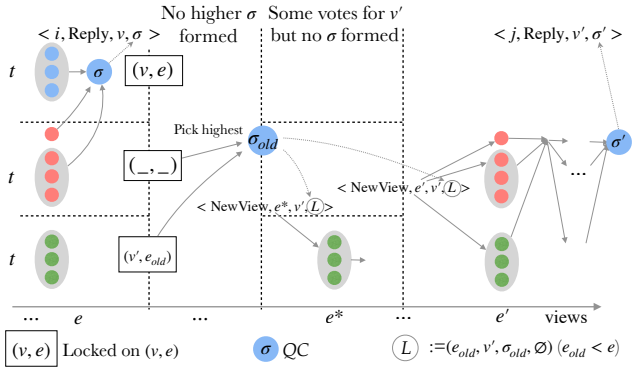

Figure 5: World 1 of Theorem 5.3. Replicas are represented as colored nodes. Replica partitions are P, $\{x\}$ (Byzantine), R (Byzantine), and Q from top to bottom.

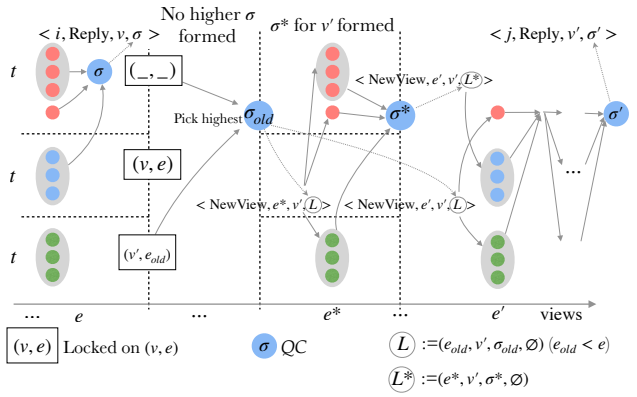

Figure 6: World 2 of Theorem 5.3. Replicas are represented as colored nodes. Replica partitions are P (Byzantine), $\{x\}$ (Byzantine), R, and Q from top to bottom.

C PROOF OF THEOREMS

C.1 Proof of Theorem 4.1

Proof. Suppose there are $f = 2t + 1$ Byzantine replicas, and let there be three replica partitions P, Q, R, $|P| = |Q| = t$, $|R| = t + 1$. To prove the result, suppose the protocol has forensic support for $d > 0$, we construct two worlds where a different set of replicas are Byzantine in each world.

World 1. Let R, Q be Byzantine replicas in this world. During the protocol, replicas in Q behave like honest parties. Suppose in view e, e' $(e < e')$, two honest replicas $p_1, p_2 \in P$ output two conflicting values v, v' after receiving two *commitQC*. The *commitQC* for v contains the Commit messages from P and R, the *commitQC* for v' contains the Commit messages from R and Q. All the other messages never reach P. During the forensic protocol, replicas in P send their transcripts to the client. Since the protocol has forensic support for $d > 0$, using these transcripts (two *commitQC*), the forensic protocol determines some subset of R are culpable (since Q behave like honest).

World 2. Let P, Q and some replica $r \in R$ are Byzantine replicas and replicas in r behave honestly. Again, in view e, e' $(e < e')$, two replicas $p_1, p_2 \in P$ output two conflicting values v, v' after receiving two *commitQC*. The *commitQC* for v contains the Commit messages from P and R, the *commitQC* for v' contains the Commit messages from R and Q. Replicas in R unlock themselves due to receiving a higher *prepareQC* formed in e^* $(e < e^* < e')$. During the forensic protocol, replicas in P send the same transcripts as of World 1 to the client (only two *commitQC*). Thus, the forensic protocol outputs some subset of R as culpable replicas. However, this is incorrect since replicas in R are honest (r is indistinguishable with replicas in $R/\{r\}$). This complete the proof. □

C.2 Proof of Theorem 5.3

Proof. Suppose the protocol provides forensic support to detect $d > 1$ Byzantine replicas with irrefutable proof and the proof can be constructed from the transcripts of all honest replicas. To prove this result, we construct two worlds where a different set of replicas are Byzantine in each world. We will fix the number of Byzantine replicas $f = t+1$, but the following argument works for any $f \geq t+1$.

Let there be four replica partitions $P, Q, R, \{x\}$. $|Q| = |P| = |R| = t$, and x is an individual replica. In both worlds, the conflicting outputs are presented in view e, e' $(e + 1 < e')$ respectively. Let $commitQC_1$ be on value v in view e, and signed by P, R, x. And let $commitQC_2$ (and a *precommitQC*) be on value v' in view e', and signed by Q, R, x. Suppose the leader of view e, e^*, e' $(e < e^* < e')$ is replica x.

World 1 is presented in Figure 5. Let R and x be Byzantine replicas in this world. In view e^*, the leader proposes value v' and Q sends Prepare on it, but a *prepareQC* is not formed. In view e', the Byzantine parties, together with Q, sign *prepareQC* on v'. e' is the first view where a *prepareQC* for v' is formed.

During the forensic protocol, all honest replicas in P and Q send their transcripts. Byzantine R and x do not provide any information. Since the protocol has forensic support, the forensic protocol can output $d > 1$ replicas in R and x as culprits.

World 2 is presented in Figure 6. Let P and x be Byzantine replicas in this world. Here, in view $e^* > e$, P and x, together with Q sign *prepareQC* on v'. In this world, e^* is the first view where a *prepareQC* for v' is formed. View $e' > e^*$ is similar to that of World 1 except that honest R receives a NewView message with $prepareQC^*$ (rather than $prepareQC_{old}$).

During the forensic protocol, Q sends their transcripts, which are identical to those in World 1. Byzantine P can provide the same transcripts as those in World 1. Observe that the transcripts from P and Q presented to the forensic protocol are identical to those in World 1. Thus, the forensic protocol can also outputs $d > 1$ replicas in R and x as culpable. In World 2, this is incorrect since replicas in R are honest.

Based on $2t$ transcripts, World 1 and World 2 are indistinguishable. To obtain an irrefutable proof of $d > 1$ culprits, the client needs to collect more than $2t$ transcripts, more than the number of honest parties available. This completes the proof. □

Remark. The above proof can be easily modified to work with parameters $d > 0$ when $m = t + 2$.

C.3 Proof of Theorem 6.1

Proof. We construct two worlds where a different set of replicas are Byzantine in each world. Let replicas be split into three partitions P, Q, and R, and $|P| = (n - 2\epsilon)/3, |Q| = |R| = (n + \epsilon)/3$ and $\epsilon > 0$ is a small constant. Denote the numbers of replicas from P, Q, R in a committee by p, q, r. Let κ denote the expected committee size; $t_H = 2\kappa/3$. With constant probability, we will have $p < \kappa/3$, $q > \kappa/3$ and $r > \kappa/3$ and $p + q < 2\kappa/3$ in steps 4 to 8.

World 1. Replicas in R are Byzantine in this world. We have $p+q < t_H$ and $q + r > t_H$. The Byzantine parties follow the protocol in *Graded Consensus*. Thus, all replicas in step 4 hold the same tuple of $b = 0$ and v ($v \neq v_\perp$). Then, the following steps are executed.

- Step 4 Honest committee members that belong to P and Q broadcast their votes on $(b = 0, v)$ whereas Byzantine committee members that belong to R send votes to replicas in P and not Q.
- Step 5 Replicas in P satisfy Ending Condition 0, and output $b = 0$ and the value v. Replicas in Q do not receive votes from committee members in R, so they update $b = 0$ and broadcast their votes on $(b = 0, v)$. Byzantine committee members that belong to R pretend not to receive votes from committee members in Q, and also update $b = 0$. And they send votes to replicas in P and not Q.
- Step 6 Replicas in Q update $b = 1$ since they receive $p+q < t_H$ votes. Replicas in R pretend not to receive votes from committee members in Q, and also update $b = 1$. Committee members in Q and R broadcast their votes.
- Steps 7-8 Committee members that belong to Q and R receive $q+r > t_H$ votes, so they update $b = 1$ and broadcast their votes.
- Step9 Replicas in Q and R satisfy Ending Condition 1, and output $b = 1$ and $v_\perp$, a disagreement with replicas in P.

During the forensic protocol, replicas in P send their transcripts and state that they have output $b = 0$. Q and R send their transcripts claiming in steps 4 and 5 they do not hear from the other partition, and they state that output $b = 1$.

If this protocol has any forensic support, then it should be able to detect some replica in R as Byzantine.

World 2. This world is identical to World 1 except (i) Replicas in Q are Byzantine and replicas in R are honest, and (ii) the Byzantine set Q behaves exactly like set R in World 1, i.e., replicas in Q do not send any votes to R in steps 4 and 5 and ignore their votes. During the forensic protocol, P send their transcripts and state that they have output $b = 0$. Q and R send their transcripts claiming in steps 4 and 5 they do not hear from the other partition, and they state that output $b = 1$.

From an external client's perspective, World 2 is indistinguishable from World 1. In World 2, the client should detect some replica in R as Byzantine as in World 1, but all replicas in R are honest. □

C.4 Proof of Theorem 7.1

Proof. Suppose two conflicting blocks b, b' are output in views e, e' respectively.

Case $e = e'$.
Culpability. The *commitQC* of b (the QC in $e + 3$) and *commitQC* of b' intersect in $t+1$ replicas. These $t+1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can get the proof based on the two blocks in $e+3$, so additional witnesses are not necessary in this case.

Case $e \neq e'$.
Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since b is output in view e, it must be the case that $2t + 1$ replicas are locked on (b, e) at the end of view e. Now consider the first view $e < e^* \leq e'$ in which a higher lock (b'', e^*) is formed where b'', b are not on the same chain (possibly b'' is on the chain of b'). Such a view must exist since b' is output in view $e' > e$ and a lock will be formed in at least view e'. For a lock to be formed, a higher *prepareQC* must be formed too.

Consider the first view $e < e^\# \leq e'$ in which a *prepareQC* in chain of b'' is formed. The leader in $e^\#$ broadcasts the block containing a *highQC* on (b'', e''). Since this is the first time a higher *prepareQC* is formed and there is no *prepareQC* for chain of b'' formed between view e and $e^\#$, we have $e'' \leq e$. The formation of the higher *prepareQC* indicates that $2t + 1$ replicas received the block extending b'' with *highQC* on (b'', e'') and consider it a valid proposal, i.e., the view number e'' is larger than their locks because the block is on another chain.

Recall that the output block b indicates $2t + 1$ replicas are locked on (b, e) at the end of view e. In this case, the $2t + 1$ votes in *prepareQC* in view $e^\#$ intersect with the $2t+1$ votes in *commitQC* in view e at $t+1$ Byzantine replicas. These replicas should be Byzantine because they were locked on the block b in view e and vote for a conflicting block in a higher view $e^\#$ whose *highQC* is from a view $e'' \leq e$. Thus, they have violated the voting rule.
Witnesses. Client can get the proof by storing a *prepareQC* formed in $e^\#$ between e and e' in a different chain from b. The *prepareQC*

is for the previous block in $e^{\#}$ whose *highQC* is formed in a view $e'' < e$. For the replicas who have access to the *prepareQC*, they must have access to all blocks in the same blockchain. Thus only one witness is needed ($k = 1$) to provide the *prepareQC* and its previous block containing the *highQC* on (b'', e''), the *prepareQC*, *highQC* and the first *commitQC* act as the irrefutable proof. □

D DESCRIPTION OF BFT PROTOCOLS

In the protocols in this paper, we assume that replicas and clients ignore messages with invalid signatures and messages containing external invalid values. When searching for an entity (e.g. lock or *prepareQC*) with the highest view, break ties by alphabetic order of the value. Notice that ties only occur when $f > t$ and Byzantine replicas deliberately construct conflicting quorum certificates in a view.

With $n = 3t + 1$, the descriptions of the PBFT protocol and HotStuff protocol are presented in Algorithm 5 and 6.

D.1 A Forensic Attack on HotStuff-view

Compared to HotStuff [32, Algorithm 2], Algorithm 6 highlights a slightly different voting rule in line 6. In addition to check whether $(LOCK.e < highQC.e) \vee (LOCK.v = v)$ holds as in HotStuff [32, Algorithm 2], when the value in NewView is the same as the value in lock, our voting rule requires $LOCK.e = highQC.e$.

We argue that the lack of this additional check on the view number will not affect the safety and liveness for HotStuff, but pose a threat for forensics. In the following, we exhibit a forensic attack on HotStuff-view protocol with the original voting rule.

- $e = i > 0$: An honest replica R receives a $\langle \text{Commit}, i, v, \sigma \rangle$ from the leader and updates its lock to be (i, v, σ). R sends $\langle \text{Commit}, i, v \rangle$ to leader, which is contained in a *commitQC* denoted as qc_1. v is output in this view.
- $e = i + 1$: R receives $\langle \text{Commit}, i + 1, v', \sigma' \rangle$ and updates its lock to be $(i + 1, v', \sigma')$.
- $e = i + 2$: A leader broadcasts $\langle \text{NewView}, i + 2, v', highQC \rangle$, where *highQC* is a QC from $i - 1$. Replica R receives the message and sends $\langle \text{Prepare}, i+2, v', i-1 \rangle$, because $LOCK.v = v'$ by checking the original voting rule. This message is contained into a *prepareQC* denoted as qc_2. Further, v' is output in this view.

In this execution, replica R follows the protocol, however, it will be mistakenly blamed by Algorithm 2 if the client receives the qc_1 for v and the qc_2 for v'. Since $qc_2.v \neq qc_1.v$ and $qc_2.e_{qc} = i - 1 \leq qc_1.e = i$ according to line 10.

While the actual *prepareQC* whose intersection with qc_1 should be blamed is generated in $e = i + 1$, it is possible that some honest replicas who have the same transcripts as R will be improperly held culpable in this case. By adding the condition to check $LOCK.e = highQC.e$, honest replicas will not vote for a NewView with stale *highQC*, which prevents them from the attack described above.

Algorithm 5 PBFT-PK protocol: replica's initial value v_i

$LOCK \leftarrow (0, v_\perp, \sigma_\perp)$ with selectors e, v, σ ▷ $0, v_\perp, \sigma_\perp$: default view, value, and signature
$e \leftarrow 1$
while true **do**
▷ **Pre-prepare and Prepare Phase**
  as a leader
    collect $\langle \text{ViewChange}, e-1, \cdot \rangle$ from $2t+1$ distinct replicas as status certificate M ▷ Assume special ViewChange messages from view 0
    $v \leftarrow$ the locked value with the highest view number in M
    if $v = v_\perp$ **then**
      $v \leftarrow v_i$
    broadcast $\langle \text{NewView}, e, v, M \rangle$
  as a replica
    wait for valid $\langle \text{NewView}, e, v, M \rangle$ from leader ▷ Use function Valid($\langle \text{NewView}, e, v, M \rangle$)
    send $\langle \text{Prepare}, e, v \rangle$ to leader
▷ **Commit Phase**
  as a leader
    collect $\langle \text{Prepare}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{Commit}, e, v, \sigma \rangle$
  as a replica
    wait for $\langle \text{Commit}, e, v, \sigma \rangle$ from leader ▷ *prepareQC*
    $LOCK \leftarrow (e, v, \sigma)$
    send $\langle \text{Commit}, e, v \rangle$ to leader
▷ **Reply Phase**
  as a leader
    collect $\langle \text{Commit}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{Reply}, e, v, \sigma \rangle$
  as a replica
    wait for $\langle \text{Reply}, e, v, \sigma \rangle$ from leader ▷ *commitQC*
    output v and send $\langle \text{Reply}, e, v, \sigma \rangle$ to the client
  call procedure ViewChange()
if a replica encounters timeout in any "wait for", call procedure ViewChange()

procedure ViewChange()
  broadcast $\langle \text{Blame}, e \rangle$
  collect $\langle \text{Blame}, e \rangle$ from $t+1$ distinct replicas, broadcast them
  quit this view
  send $\langle \text{ViewChange}, e, LOCK \rangle$ to the next leader
  enter the next view, $e \leftarrow e+1$
function Valid($\langle \text{NewView}, e, v, M \rangle$)
  $v^* \leftarrow$ the locked value with the highest view number in M
  if $(v^* = v \vee v^* = v_\perp) \wedge (M$ contains locks from $2t+1$ distinct replicas$)$ **then**
    return $true$
  else
    return $false$

Algorithm 6 General HotStuff protocol: replica's initial value v_i, protocol variant indicator $var \in$ {'HotStuff-view', 'HotStuff-hash', 'HotStuff-null'}

$prepareQC \leftarrow (0, v_\perp, \sigma_\perp, Info_\perp)$ with selectors $e, v, \sigma, Info$
$LOCK \leftarrow (0, v_\perp)$ with selectors e, v ▷ $0, v_\perp, \sigma_\perp, Info_\perp$: default view, value, signature, and info
$e \leftarrow 1$
while true **do**
▷ **Pre-prepare and Prepare Phase**
  as a leader
    collect $\langle \text{ViewChange}, e-1, \cdot \rangle$ from $2t+1$ distinct replicas as M ▷ Assume special ViewChange messages from view 0
    $highQC \leftarrow$ the highest QC in M
    $v \leftarrow highQC.v$
    if $v = v_\perp$ **then**
      $v \leftarrow v_i$
    broadcast $\langle \text{NewView}, e, v, highQC \rangle$
  as a replica
    wait for $\langle \text{NewView}, e, v, highQC \rangle$ from leader s.t. $highQC.v = v \vee highQC.v = v_\perp$ ▷ Validate v
    if $(LOCK.e < highQC.e) \vee (LOCK.v = v \wedge LOCK.e = highQC.e)$ **then** ▷ Voting rule, see Appendix D.1
      send $\langle \text{Prepare}, e, v, \text{Info}(var, highQC) \rangle$ to leader ▷ Use function $\text{Info}(var, highQC)$
▷ **Precommit Phase**
  as a leader
    collect $\langle \text{Prepare}, e, v, Info \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{Precommit}, e, v, \sigma, Info \rangle$
  as a replica
    wait for $\langle \text{Precommit}, e, v, \sigma, Info \rangle$ from leader ▷ $prepareQC$
    $prepareQC \leftarrow (e, v, \sigma, Info)$
    send $\langle \text{Precommit}, e, v \rangle$ to leader
▷ **Commit Phase**
  as a leader
    collect $\langle \text{Precommit}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{Commit}, e, v, \sigma \rangle$
  as a replica
    wait for $\langle \text{Commit}, e, v, \sigma \rangle$ from leader ▷ $precommitQC$
    $LOCK \leftarrow (e, v)$
    send $\langle \text{Commit}, e, v \rangle$ to leader
▷ **Reply Phase**
  as a leader
    collect $\langle \text{Commit}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{Reply}, e, v, \sigma \rangle$
  as a replica
    wait for $\langle \text{Reply}, e, v, \sigma \rangle$ from leader ▷ $commitQC$
    output v and send $\langle \text{Reply}, e, v, \sigma \rangle$ to the client
  call procedure ViewChange()
if a replica encounters timeout in any "wait for", call procedure ViewChange()

```
41: procedure ViewChange()
42:     broadcast ⟨Blame, e⟩
43:     collect ⟨Blame, e⟩ from t + 1 distinct replicas, broadcast them
44:     quit this view
45:     send ⟨ViewChange, e, prepareQC⟩ to the next leader
46:     enter the next view, e ← e + 1
47: function Info(var, highQC)                    ▷ var ∈ {'HotStuff-view', 'HotStuff-hash', 'HotStuff-null'}
48:     if var ='HotStuff-view' then
49:         return highQC.e
50:     if var ='HotStuff-hash' then
51:         return Hash(highQC)
52:     if var ='HotStuff-null' then
53:         return ∅
```