Optimistic Fast Confirmation While Tolerating Malicious Majority in Blockchains

Ruomu Hou
National University of Singapore
houruomu@comp.nus.edu.sg

Haifeng Yu
National University of Singapore
haifeng@comp.nus.edu.sg

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179323

***Abstract*—The robustness of a blockchain against the adversary is often characterized by the maximum fraction ($f_{\max}$) of *adversarial power* that it can tolerate. While most existing blockchains can only tolerate $f_{\max} < \frac{1}{2}$ or lower, there are some blockchain systems that are able to tolerate a malicious majority, namely $f_{\max} \geq \frac{1}{2}$. A key price paid by such blockchains, however, is their large *confirmation latency*. This work aims to significantly reduce the confirmation latency in such blockchains, under the common case where the actual fraction f of adversarial power is relatively small. To this end, we propose a novel blockchain called FLINT. FLINT tolerates $f_{\max} \geq \frac{1}{2}$ and can give optimistic execution (i.e., fast confirmation) whenever f is relatively small. Our experiments show that the fast confirmation in FLINT only takes a few *minutes*, as compared to several *hours* of confirmation latency in prior works.**

I. INTRODUCTION

Background. The robustness of a blockchain against the adversary is often characterized by the maximum fraction ($f_{\max}$) of *adversarial power* that it can tolerate. Here $f_{\max}$ corresponds to for example, the fraction of malicious nodes in a permissioned setting, or the fraction of adversarially-controlled stakes/computation power in a (hybrid/permissionless) Proof-of-Stake/Proof-of-Work setting.

Most blockchains in the literature (e.g., [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]) can only tolerate $f_{\max} < \frac{1}{2}$ or lower. Furthermore, due to the overhead of approaching the theoretical limits, there is usually a gap between the *theoretical* $f_{\max}$ and the *practical* $f_{\max}$ that an actual implementation intends to tolerate. The practical $f_{\max}$ considered in various blockchain implementations/experiments in the literature (e.g., [5, 8, 9, 10, 11]) typically ranges from $f_{\max} = 0.2$ to 0.25.

There are also some blockchains [12, 13, 14, 15] that can tolerate $f_{\max} \geq \frac{1}{2}$. These blockchains are more robust against, for example, large mining pools controlled by a few entities. More generally, they are more resilient against *correlated* byzantine failures, for example, when an attacker compromised a large fraction of nodes by exploiting a common vulnerability. The current state-of-the-art design for such blockchain systems is BCUBE [15]. BCUBE's design aims for $f_{\max} = 0.99$ in theory, while their actual implementation/experiments consider $f_{\max} = 0.7$.

Large confirmation latency. A key price paid by these blockchains such as BCUBE, however, is their large *confirmation latency*. Here the *confirmation latency* refers to the time needed for the blockchain to confirm the final position of a block in the distributed ledger, after the block is proposed. For example, in BCUBE's experiments [15] for $f_{\max} = 0.7$, it takes about 6 hours for a block to get confirmed. This is in sharp contrast with blockchains [2, 3, 4, 5, 6, 7, 8, 9, 10, 11] that only tolerates $f_{\max} < \frac{1}{2}$, whose confirmation latency is usually on the order of minutes or less.

Our goal. This work explores the possibility of significantly reducing such confirmation latency. Part of this hour-long confirmation latency, unfortunately, is somewhat fundamental: To tolerate $f_{\max} \geq \frac{1}{2}$, these blockchains [14, 15] need to rely on *byzantine broadcast* protocols [12, 13, 15, 16, 17, 18, 19, 20] that can tolerate $f_{\max} \geq \frac{1}{2}$, as their core. Existing lower bounds [13] show that, under certain conditions, the worst-case number of rounds needed by such a byzantine broadcast protocol is at least *linear*, with respect to either the number of nodes in the system or the number of committee members (if a committee is used).[1]

Let f, where $f \leq f_{\max}$, be the *actual* fraction of adversarial power at any given point of time. The above discussion suggests that reducing confirmation latency has some inherent obstacle when $f = f_{\max}$. But one would imagine that in common/typical cases, f is likely to be smaller than $f_{\max}$.

Thus our goal is to significantly reduce the confirmation latency, under common cases where f is relatively small (e.g., $f = 0.2$). When f is large (e.g., $f = 0.6$), our design should still remain secure. It is worth emphasizing that our notion of "small" is only in the relative sense. The reason is that our "small" f actually is similar to the maximum tolerance of many existing blockchain implementations [5, 8, 9, 10, 11], whose $f_{\max}$ is only about 0.2 to 0.25.

Our central contribution. The central contribution of this paper is the design, analysis, implementation, and evaluation of a novel blockchain protocol called FLINT. FLINT has an *optimistic track* and a *normal track*. FLINT always guarantees security, even if f is large (e.g., $f = 0.6$). But if f is small (e.g., 0.2), the optimistic track gives fast confirmation. When f is large, the normal track gives normal confirmation. Our experiments show that the fast confirmation in FLINT only takes a few *minutes*, as compared to several *hours* in

[1] While theoretically it is possible to overcome this linear bound in some other cases, existing theoretical approaches [20, 21] of doing so are not practically feasible in large-scale systems with thousands nodes: These approaches [20, 21] incur a large amount of communication — specifically, $\Omega(n^4)$ bits of communication, where n is the number of nodes in the system.

© 2023, Ruomu Hou. Under license to IEEE.

DOI 10.1109/SP46215.2023.00145
Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

BCUBE [15]. We believe that such a significant reduction, from hours to minutes, achieves an important step forward, in facilitating the broader application/adoption of blockchains that can tolerate a malicious majority.

A large body of prior works on blockchains, as well as byzantine fault-tolerant systems in general, have similarly explored various notions of *optimistic execution* or *optimistic track* [14, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38], with the goal of improving common-case performance. However, most of these designs fundamentally only work for $f_{\max} < \frac{1}{2}$. *To our knowledge,* FLINT *is the first blockchain system that tolerates* $f_{\max} \geq \frac{1}{2}$ *and can give optimistic execution (i.e., fast confirmation) whenever* f *is relatively small.*

Main problem. If we knew the value of f, and if we could choose which track to run based on that knowledge, then we could trivially use existing techniques to design the two tracks in FLINT. But f is actually *unknown*.

As a result, the optimistic track in FLINT will be invoked, regardless of whether f is actually small. We need to ensure that the optimistic track does not generate wrong outputs, even if f turns out to be large. Second, when f is large, some nodes may still generate (correct) fast confirmation from the fast track, while the remaining nodes will get normal confirmation from the normal track. We need to properly "couple" these two tracks to ensure agreement.

At a deeper level, achieving all these under $f_{\max} \geq \frac{1}{2}$ involves new technical challenges that did not exist when $f_{\max} < \frac{1}{2}$. Section II gives examples of such new challenges.

Our approach. Our design is structured, and is driven by a sequence of logical steps:

- *Ensure that the optimistic track does not generate wrong outputs, even if* f *turns out to be large.* We design a mechanism using `b_cert`, to ensure the consistency across all fast outputs.
- *Properly "couple" the optimistic track and normal track.* In the normal track, we use a standard byzantine broadcast protocol to disseminate the block from the block proposer. Concurrently, we use another *generalized* byzantine broadcast protocol to disseminate the fast outputs from the optimistic track. We finally let the return value (if any) from the generalized protocol take precedence.
- *Ensure that if the block proposer is honest, then its block (instead of an empty block) must be adopted.* To prevent the empty block from being (incorrectly) adopted as the decision, we design another certificate, `p_cert`, as a prerequisite for an empty block to be used as the decision.

Our results. Our formal and end-to-end security analysis shows that for any given $\frac{1}{2} \leq f_{\max} \leq 0.99$, we can configure FLINT such that (see Theorem 4):

- FLINT always guarantees security (i.e., safety and liveness) when $f \leq f_{\max}$.
- When $f \leq \min(1 - f_{\max}, \frac{1}{3}) - c$, FLINT generates fast confirmation for each block, with a small confirmation latency. Otherwise FLINT will have normal confirmation latency. Here the positive constant c can be made arbitrarily close to 0, by increasing the size of certain committees in FLINT.

We have also implemented a research prototype of FLINT. We run experiments with up to 10000 nodes, under similar and realistic settings as the state-of-the-art BCUBE protocol [15]. Our results show that FLINT drastically reduces confirmation latency, in common cases where f is relatively small. For example under $f_{\max} = 0.6$, FLINT confirms a block in around 3.8 minutes when $f \leq 0.2$, while BCUBE takes around 292 minutes [15] to confirm a block regardless of how small f is. This is a reduction of around 76 times in confirmation latency. For larger $f \in (0.2, 0.6]$, FLINT gives a confirmation latency that is rather close to BCUBE's.

Finally, FLINT does pay a small price in throughput when compared to BCUBE, due to the additional mechanism in FLINT. Our experiments show that relative to BCUBE, the throughput of FLINT is about 6% lower. We feel that this is a relatively small price to pay.

Roadmap. Section II gives some example technical challenges that we face in FLINT. Section III describes our system/attack model. Section IV through V presents the design of FLINT. Section VI provides our security analysis. Section VII presents implementation details and experimental results. Section VIII elaborates on the issue of external verifiability. Section IX discusses related works.

II. TECHNICAL CHALLENGES

Byzantine fault-tolerant systems typically build on a common set of top-level techniques, such as using various quorums and certificates. FLINT is no exception. The specific designs of these top-level techniques, however, greatly impact the final guarantees. Hence, different protocols are essentially about how to best design the various quorums/certificates, given the specific end goals. This section describes two example challenges faced by FLINT, given its specific goal.

Example challenge #1. A common design for the optimistic track is to let some *leader* node propose and then collect votes from all nodes. If sufficient votes are collected, then the protocol generates a fast confirmation. Otherwise the protocol falls back to the normal track. Here to get a fast confirmation, it is necessary for the leader to be honest.

But in the regime of $f_{\max} \geq \frac{1}{2}$, the normal track is rather slow (e.g., 6 hours [15]). This significantly amplifies the negative impact of a malicious leader in the above design. For example, if $f = 0.2$ and if we choose leaders randomly for each block, then roughly for every 5 blocks, we will need to pay the large overhead (e.g., 6 hours) of the normal track. Hence, we avoid such a design. Instead, we aim to generate fast confirmation as long as f is relatively small, without needing any specific node to be honest.

A natural idea for achieving our goal would be to replace the simple voting (coordinated by the leader), with a more general byzantine agreement protocol (such as HotStuff [32]) that can succeed as long as $f < \frac{1}{3}$. A deeper look, however,

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

reveals a problem with **validity**. **Validity** means that if the block proposer is honest, then the blockchain should adopt the block proposed by the block proposer, instead of adopting for example, an empty block. Now imagine that an honest block proposer proposes a block b. When $f > \frac{1}{2}$, HotStuff will not offer any guarantees, and may return an empty block as the decision value. This empty block will then become the fast output, which violates **validity**.

Overcoming this issue takes several more steps in our design. First, we require some additional certificate (i.e., `p_cert`), before the empty block is allowed to be used as the fast output. We ensure that this certificate can never be generated when the block proposer is honest. This in turn prevents the above problem with **validity**. This first step, unfortunately, has an undesirable side effect: It may prevent the proper generation of fast output when the block proposer is malicious. This brings us back to original problem of requiring an honest leader.

To overcome this, we observe that for the problem to occur while using `p_cert`, i) the value of f needs to be small (since otherwise we do not expect fast output anyway), and ii) the malicious proposer needs to send its block to a small fraction of honest nodes. This means that some honest node already sees some block from the malicious proposer. Based on this, we invoke HotStuff in a way, such that a block is always more favored than an empty block, as the decision value, when some nodes feed a block into HotStuff while others feed an empty block. (Our actually design is more complex, to account for all the corner cases.) Directly doing this, however, could result in poor performance. Hence as the last step, we design additional optimizations to ensure that our approach will be efficient.

Example challenge #2. A common design of the normal track is for each node to feed the various certificates, which it has collected (if any) from the optimistic track, into the normal track. This enables the normal track to properly follow the decision made (if any) in the optimistic track. In the regime of $f_{\text{max}} < \frac{1}{2}$, this can be easily done by feeding all such certificates into a byzantine agreement protocol.

With $f_{\text{max}} \geq \frac{1}{2}$, however, we will need to use byzantine broadcast (as in BCUBE [15]), instead of byzantine agreement. A byzantine broadcast protocol only allows broadcast from a single node. Hence, a naive design would require up to n parallel invocations of byzantine broadcast, resulting in prohibitively overhead. To overcome this, in FLINT, we design a generalized byzantine broadcast protocol that simultaneously allows many broadcasters, while without blowing up the overhead.

III. System Model and Attack Model

Since this work improves upon BCUBE [15], to facilitate a direct comparison, we inherit the system/attack model from [15]. Specifically, we consider a setting without PKI, except for potentially using an initial PKI to create the genesis block that specifies the initial stake distribution at bootstrapping time. Each node has a locally generated public/private key pair, where the public key also serves as the *id* of the node. Each node is either *honest* or *malicious*. A malicious node is fully byzantine, and may deviate from the protocol arbitrarily. We assume that there is an *adversary* controlling all the malicious nodes, and the malicious nodes can all collude in arbitrary ways. As in BCUBE and a number of other blockchains [2, 3, 4], we allow the adversary to be *mildly-adaptive* — namely, it takes some time (e.g., many epochs) for the adversary to adaptively corrupt nodes.

We use *Proof-of-Stake* in our design: There are some stakes in the system, and a stake can be transferred from node to node, or dynamically generated/destroyed if needed, via transactions in the underlying blockchain. As in typical Proof-of-Stake blockchains [3, 4, 5], we assume that at any given point of time, the fraction of stakes that are owned by malicious nodes is at most f_{max}. For the design of FLINT, we allow f_{max} up to 0.99. At any given point of time, we use f to denote the actual fraction of stakes that are owned by malicious nodes. The value f_{max} should be provided to FLINT as a parameter, while f is unknown to FLINT. We will adopt the beacon generation mechanism in BCUBE [15], and hence need to also inherit the *weak Proof-of-Work* assumption [15] needed by that mechanism: We assume that the aggregate computational power of the malicious nodes is no more than 100 times of that of the honest nodes. This assumption is separate from, and in addition to, the assumption on f_{max}.

As in most large-scale blockchain systems (e.g., [5, 7, 39, 40]) including BCUBE, the nodes in FLINT form an overlay network. For convenience, we view each (undirected) edge in this overlay as two directed edges in opposite directions. We say that a direct edge from A_1 to A_2 is *good*, if i) both A_1 and A_2 are honest nodes, and ii) a small-sized message (e.g., 10KB) sent by A_1 will be received by A_2 within δ_1 time. Under relatively large δ_1 value (e.g., $\delta_1 = 10$ seconds), we expect most edges among honest nodes to be good. Same as BCUBE [15], we assume the subgraph containing all the honest nodes and all the good edges to be strongly connected, and we let d be any value that is no smaller than the diameter of this subgraph. Note that eclipse/partitioning attacks [41, 42] on the overlay network may cause this subgraph to be disconnected — dealing with such attacks is an active research topic [42, 43, 44] and is beyond our scope. We assume that the physical clock readings on two honest nodes differ by at most δ_2 (e.g., $\delta_2 = 2$ seconds). We assume that CPU processing time is negligible. Parts of our protocol will use the notion of *rounds*, where each round lasts $\delta = \delta_1 + \delta_2$ time. (Our implementation uses $\delta = 12$ seconds.) Each node uses its local physical clock to determine the beginning/end of each round. One can easily verify that with our definition of δ and despite clock error, a small-sized message sent along a good edge at the beginning of a round will be received by the receiver, by the beginning of the next round. We define $\Delta = d \cdot \delta$.

As in most blockchains, we assume that a public *genesis block* provides some trusted unbiased randomness, as well as an initial stake distribution among the nodes, for bootstrapping. We assume that standard crypto operations such as digital signatures cannot be broken, and we view hash functions as

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

random oracles.

Transaction and blocks. We assume that nodes in the blockchain generate transactions. These transactions are then disseminated to all nodes via a background gossiping process on the overlay network. A node proposing a new block simply chooses some random transactions that it has seen but have not yet been included in any confirmed blocks. Whenever a node A_1 intends to send a transaction (potentially as part of a block) to another node A_2, A_1 will check whether A_2 has already seen that transaction, to avoid redundant sending of transactions.

External/public verifiability. Typical blockchains provide a key *external/public verifiability* property, in order to support *light-clients* [45]. Here a light-client does not fully/continuously participate in the blockchain protocol. Instead, it uses the blockchain only by interacting with one or more *full-nodes* on demand. (In our description of the FLINT protocol, a node, regardless of whether it is in various committees, is always a full-node.) Despite this, a light-client can still securely extract/verify the current blockchain state, by cross-checking against multiple full-nodes [46, 47].

When $f \geq \frac{1}{2}$, unfortunately, such external/public verifiability is no longer possible to achieve [48]. Hence FLINT does *not* provide external/public verifiability when $f \geq \frac{1}{2}$. This is similar to prior designs [14, 15] for tolerating a malicious majority. Section VIII provides more discussions on this.

IV. Overview of Flint

FLINT builds and improves upon BCUBE. The following reviews BCUBE first, and then describes FLINT.

Review of BCUBE **[15].** In BCUBE, conceptually, each node has an infinite list of *slots*, where all slots are initially unfilled. Let time t_0 be the time that the blockchain starts running, and let x be the inter-block time (e.g., 98 seconds) in BCUBE. For each slot i ($i = 1, 2, \ldots$), all nodes in BCUBE will invoke a byzantine broadcast protocol called OVERLAYBB [15] at time $t_0 + i \cdot x$. That invocation will eventually output a block on each node. A node then places that block into its i-th slot, and the block becomes the i-th block in the blockchain on that node. Note that the OVERLAYBB invocation for slot i may start, before the OVERLAYBB invocation for slot $i-1$ returns. Hence there will be multiple concurrently-active invocations.

To invoke OVERLAYBB, the system needs to elect a random committee (denoted as `committeeN`). To do so, BCUBE has a *beacon generation mechanism*, which generates periodic beacons. Each beacon is a random value that is agreed upon by all honest nodes. Roughly speaking, there will be a *beacon* s_i for each slot i. The system can then select a random set of stakes, by using $\texttt{hash}(i|s_i)$ as the randomness to index into the stake distribution. The owners of the selected stakes will be the `committeeN` for slot i. BCUBE further uses the first member in `committeeN` as the block *proposer* for slot i. After the beacon is released, all nodes in the system will know the public keys of the proposer and all the members in `committeeN`.

The proposer will construct a block and feed that block into its invocation of OVERLAYBB, as the *proposed block*. Hou et al. [15] have proved:

Fact 1 (Guarantee of OVERLAYBB)**.** *If* `committeeN` *has at least one honest member, then:*

- OVERLAYBB *must eventually return on all honest nodes.*
- OVERLAYBB *must return the same block on all honest nodes.*
- *If the block proposer is honest, then* OVERLAYBB *must return the block proposed by the proposer.*

These properties lead to the **Safety** and **Liveness** of the BCUBE blockchain itself: First, for any given slot, since OVERLAYBB must return the same block, all honest nodes must have the same block in that slot, which gives **Safety**. In particular, there will not be any forks. Second, the sequence of blocks on each node must grow at a steady rate since OVERLAYBB is periodically invoked and completed, which gives **Liveness**.

Finally, the block confirmation latency in BCUBE essentially equals the time needed for OVERLAYBB to return. Since it takes several hours for OVERLAYBB to return, BCUBE's confirmation latency is also several hours [15].

Overview of FLINT**.** FLINT is the same as BCUBE, except that FLINT replaces OVERLAYBB in BCUBE with another protocol called FLINTBB that we have designed. FLINT keeps all other parts in BCUBE unchanged. Since FLINTBB is the main novelty in FLINT, the remainder of the paper will elaborate on FLINTBB. For clarity, we will imagine that each node holds exactly one stake/coin. Generalizing to the case where each node may hold multiple coins is trivial, as in BCUBE [15]. As an optimization, FLINTBB uses aggregate signatures to reduce signature size, in the same way as BCUBE [15] — see Appendix III for details.

V. Design of FlintBB

A. Overview of FlintBB

Top-level design of FLINTBB**.** The following gives our FLINTBB protocol, at the highest level:

1) The block *proposer*, which may be either honest or malicious, constructs a block.
2) All nodes spend some fixed amount of time executing the `OptimisticTrack`() subroutine in FLINTBB. Here the amount of time (e.g., 10 minutes) is chosen such that when f is small, fast outputs can usually be generated within so much time. If `OptimisticTrack`() returns within that time frame on a node A, then node A outputs that return value as the *fast output*.
3) All nodes stop executing `OptimisticTrack`().
4) Finally, all nodes invoke the `NormalTrack`() subroutine in FLINTBB. `NormalTrack`() always eventually returns. A node outputs the return value of `NormalTrack`() as the *normal output*, if the node previously did not generate a fast output. Otherwise the node discards the return value of `NormalTrack`().

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

`OptimisticTrack()` enables a node to generate a fast output when f is small. When f is large, `OptimisticTrack()` may fail to return on some nodes. In such a case, a (slower) normal output will be generated from `NormalTrack()`. A node that has already generated a fast output will still participate in `NormalTrack()`, to help other nodes (if any) to generate the normal output. Such extra work will not delay the creation of the next block, since as explained in Section IV and same as in BCUBE [15], for each slot, FLINT invokes FLINTBB at a pre-determined time. Note that all blocks eventually will still be totally-ordered.

We will aim for a modular design, while exploiting existing building blocks whenever possible. Our `OptimisticTrack()` and `NormalTrack()` subroutines will use HotStuff [32] and OVERLAYBB [15], respectively, as building blocks and in a black-box fashion. HotStuff [32] is an existing BFT agreement protocol, while OVERLAYBB [15] is an existing byzantine broadcast protocol. Roughly speaking, HotStuff achieves agreement faster but can only tolerate $f < \frac{1}{3}$, while OVERLAYBB achieves agreement slower but can tolerate all $f < 1$.

Example scenario to drive our discussion. Consider a system with many nodes, and a large f value such as $f = 0.6$. Even under large f, some honest nodes might still generate fast outputs. For example, this can happen if the malicious nodes behave maliciously in some steps, but honestly in others. Let A_1 through A_4 be any four honest nodes in this system. Imagine that A_1 and A_2 have generated fast outputs b_1 and b_2, respectively. On A_3 and A_4, `OptimisticTrack()` fails to return. Hence A_3 and A_4 generate normal outputs b_3 and b_4, respectively. Let b be the block constructed by the block proposer in FLINTBB.

Our discussions in this section will focus on intuitions, and do not constitute a security analysis. (Section VI later will give a complete formal analysis.) Intuitively, to guarantee correctness, we need to ensure that $b_1 = b_2 = b_3 = b_4$. When the block proposer is honest, we further need $b_1 = b_2 = b_3 = b_4 = b$. We hence drive the discussions in the remainder of this section, by explaining step-by-step:

- How to ensure $b_1 = b_2$: See Section V-B.
- How to ensure $b_3 = b_4$: See Section V-C.
- How to ensure $b_1 = b_3$, namely, how to "couple" the optimistic track and the normal track: See Section V-C and V-D.
- How to ensure $b_1 = b$ and $b_3 = b$, when the proposer is honest (i.e., how to ensure **validity**): See Section V-E.

While not explicitly mentioned, in all the above steps, we also need to ensure the generation of fast outputs when f is small.

B. Basic Design of `OptimisticTrack()`

To use FLINTBB, the system needs to elect a committee of nodes. We use `committeeH` to denote this committee. This committee is chosen randomly in the same way as `committeeN` in BCUBE [15], as reviewed in Section IV. Consider any given slot in FLINT. Note that *all* nodes, regardless of whether they are in `committeeH`, will execute the FLINTBB protocol for that slot. In particular, non-committee-members will also invoke FLINTBB, except that the protocol will ask them to take fewer actions (see Algorithm 2 later for those actions). The very first node in `committeeH` will be the block *proposer* for that slot.

f_{max}	max fraction of malicious stakes
f	actual fraction of malicious stakes at specific time
f'	fraction of malicious stakes in `committeeH`
α, β	internal parameters in FLINTBB
`committeeH`	committee for running HotStuff
`committeeN`	committee needed by OVERLAYBB and CERTBB

TABLE I: Key notations in FLINTBB.

Some technical notations. Recall that our end goal is to:

- When $f \leq f_{\text{max}}$, guarantee security.
- When $f \leq \min(1 - f_{\text{max}}, \frac{1}{3}) - c$, generate fast output (in addition to guaranteeing security).

We use f' to denote the fraction of malicious stakes (and nodes, when each node has 1 stake) in `committeeH`. Due to randomness, f' might not exactly equal f. For convenience, we translate the conditions on f, to conditions on f'. Specifically, given f_{max}, FLINTBB will compute two internal parameters α and β (explained later), where $0 \leq \alpha < \min(1 - \beta, \frac{1}{3}) \leq 0.5 \leq \beta < 1$. For example, for $f_{\text{max}} = 0.6$, our later experiments use $\alpha = 0.279$ and $\beta = 0.720$. We then aim to:

- When $f' \leq \beta$, guarantee security.
- When $f' \leq \alpha$, generate fast output (in addition to guaranteeing security).

We will later show that the conditions on f will lead to the corresponding conditions on f', with overwhelming probability. Table I summarizes our key notations.

Starting point. Our basic design of `OptimisticTrack()` is illustrated in Figure 1. Here the proposer first *propagates*[2] the block that it has constructed, with a signature, to all nodes in the overlay network. Note that the proposer may be honest or malicious. Next, all nodes in `committeeH` invokes HotStuff [32], while feeding the block that it has received, as its input into HotStuff. If the node has received none, it feeds the special (empty) block Ψ into HotStuff. This special block Ψ is publicly-known. Our invocation of HotStuff will then give a return value on each node in `committeeH`.

The invocation of HotStuff in FLINTBB will stop, either when `OptimisticTrack()` generates a fast output or when `OptimisticTrack()` times out, whichever is earlier. Note that the HotStuff invocation is tied to the current invocation of FLINTBB. For different invocations of FLINTBB, their internal invocations of HotStuff are unrelated.

Problem so far. Consider any two honest nodes, A_1 and A_2, in `committeeH`. If $f' \leq \alpha < \frac{1}{3}$, then all guarantees of HotStuff

[2]In various places in FLINTBB, a node may *propagate* a certain message x on the overlay network, by sending x to its neighbors and by asking its neighbors to recursively relay/forward this message to their neighbors. Care should be taken to prevent the adversary from overwhelming the relaying capacity of an honest node. Our technical report [1] explains how to achieve this.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

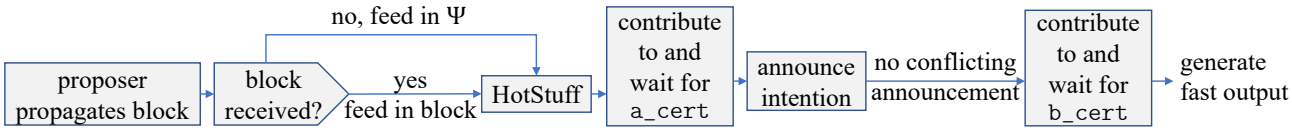


Fig. 1: Basic design of the optimistic track, as in Section V-B, for nodes in committeeH. (The part about p_cert in Section V-E is not included in this basic design yet.)

type	purpose	possible range for threshold
b_cert	cert for a block (either normal block or Ψ)	$\beta < t_b \leq 1-\alpha$
a_cert	cert needed to announce intention of contributing to b_cert	$\alpha < t_a \leq 1-\alpha$
p_cert	additional cert for special block Ψ	$\beta < t_p \leq 1-\alpha$

TABLE II: Various certificates and their respective thresholds.

must hold [32]. This means that HotStuff must return the same value on A_1 and A_2. Nodes A_1 and A_2 could then safely use this return value as their respective fast outputs b_1 and b_2.

But if f' is large, then HotStuff will not offer any guarantees. In such a case, if A_1 and A_2 naively adopt the return values from HotStuff as fast outputs, then b_1 and b_2 may differ.

How to ensure $b_1 = b_2$. In FLINTBB, a node does not directly use the return value from HotStuff as the fast output. Instead, each node A_1 in committeeH will sign the return value x that A_1 gets from HotStuff, and propagates that signature. When A_1 does this, we also say that A_1 *contributes* to the b_cert (i.e., *block certificate*) of x. A valid b_cert for x comprises at least $\lceil t_b \cdot |\texttt{committeeH}| \rceil$ such signatures, where t_b is the *threshold* for b_cert. A node will use x as the fast output, if and only if it sees a valid b_cert for x. Table II summarizes the various certificates in FLINTBB.

Recall that we want security when $f' \leq \beta$, and we further want fast output when $f' \leq \alpha$. If we choose t_b such that $t_b > \beta$, then for all $f' \leq \beta$, a valid b_cert must contain the contribution from at least one honest node. We will later ensure that different honest nodes never contribute to the b_cert of two different blocks — namely, there will be a unique block, such that each honest node either contribute to the b_cert of that block, or does not contribute to any b_cert at all. Combined with the earlier reasoning, we have the following key property:

> *If $t_b > \beta$ and $f' \leq \beta$, then there can be at most one block for which there is a valid* b_cert.

This means that in our earlier example, despite that A_1 and A_2 get different return values from HotStuff, if they both generate fast outputs, then their fast outputs must be the same since they both need to see a valid b_cert.

How to ensure fast output. We also need to ensure that the requirement of b_cert does not hinder the desired generation of fast output, when f' is small. Specifically, when $f' \leq \alpha$, HotStuff must return the same block x on all honest members in committeeH. If we choose t_b such that $t_b \leq 1-\alpha$, then the contribution from all the honest nodes will already be sufficient to form a valid b_cert for x, enabling fast outputs. Note that here, running HotStuff ensures that all honest members in committeeH contribute to the b_cert of x.

Choosing the threshold t_b. The two requirements on t_b so far, together, imply that t_b can be any value in $(\beta, 1-\alpha]$, as summarized in Table II.

Mechanism related to a_cert. We next explain how to ensure that different honest nodes never contribute to the b_cert of two different blocks. Before contributing to the b_cert of a block x_1, a node A_1 announces its *intention* to do so, by propagating an announcement in the network. If within Δ time, A_1 does not see any conflicting announcements (i.e., for another block x_2 from another node A_2), then A_1 will proceed and contribute to the b_cert of x_1. This effectively prevents two honest nodes from contributing to the b_cert of different blocks, since at least one of them must see the conflicting announcements from the other.

Naively doing this, however, would introduce a vulnerability: Malicious nodes can make announcements freely, causing honest nodes to always refrain from contributing to b_cert. Note that we only care about this problem when $f' \leq \alpha$, since when $f' > \alpha$, we do not hope for fast output anyway.

Now to deal with this problem when $f' \leq \alpha$, in FLINTBB, a node A_1 first contributes to the a_cert for the block x_1 that A_1 intends to later make an announcement for. As usual, such contribution is done by A_1 generating a signature on the certificate type and the block x_1, and then propagating that signature. Next, to announce its intention to contribute to the b_cert of x_1, node A_1 is required to include in its announcement a valid a_cert for x_1. One can hence view a_cert as an *announcement certificate*. We set the threshold t_a for a_cert such that $\alpha < t_a \leq 1-\alpha$. Namely, $\lceil t_a \cdot |\texttt{committeeH}| \rceil$ signatures are needed for a valid a_cert. When $f' \leq \alpha$, since HotStuff returns the same value x_1 on all honest nodes, the adversary cannot obtain a valid a_cert for any other value x_2, while the honest nodes can always get a valid a_cert for x_1. This prevents the malicious nodes from making announcements other than for x_1.

Comment. In some sense, the above mechanisms can be viewed as a generalization of the mechanism in Abraham et al. [35]. Abraham et al.'s protocol is a byzantine agreement protocol for tolerating up to $\frac{1}{2}$ fraction of malicious nodes. Their protocol can generate a fast output under *good-case*, where all honest nodes have the same input value. Part of their design also uses mechanisms related to our use of b_cert and a_cert. However, our normal case is different from theirs: In

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

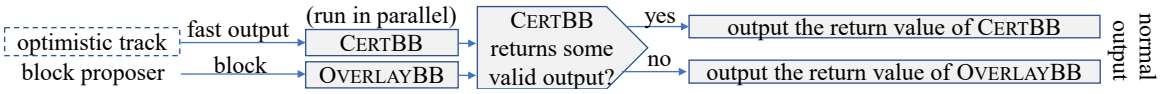


Fig. 2: Design of the normal track, as in Section V-C.

their normal case, the fraction of malicious nodes is still below $\frac{1}{2}$. Hence their protocol only needs to deal with the (single) tolerance level of $\frac{1}{2}$. In our normal case, f' can be above $\frac{1}{2}$, and we have two separate fault-tolerance levels, α and β. We need to show (as we did earlier) that the possible ranges of t_a and t_b are still non-empty under our setting.

Nodes not in `committeeH`. Our discussion so far focused on A_1 and A_2, which are in `committeeH`. To help nodes not in `committeeH` to also generate fast outputs, after A_1 gets a fast output, A_1 will propagate its fast output b_1 and the corresponding `b_cert`. When a node not in `committeeH` sees this, it immediately adopts b_1 as its own fast output. By our earlier reasoning, here b_1 must be unique.

C. Design of `NormalTrack()`

Recall the driving scenario in Section V-A, where A_1 and A_2 respectively generate fast outputs b_1 and b_2, while A_3 and A_4 respectively generate normal outputs b_3 and b_4. Our discussion so far has explained how we ensure $b_1 = b_2$. Now we explain how our design of `NormalTrack()` ensures $b_3 = b_4$ and $b_1 = b_3$. Figure 2 illustrates our design.

How to ensure $b_3 = b_4$. `NormalTrack()` uses OVERLAYBB [15] as a building block. See Section IV for a review of the functionality of OVERLAYBB. Recall that in `OptimisticTrack()`, there is a proposer node that constructs the block for the current slot. This proposer will also act as the block proposer in OVERLAYBB in `NormalTrack()`. The OVERLAYBB protocol further needs a committee (denoted as `committeeN`), and we select `committeeN` in the same way as BCUBE [15], as reviewed in Section IV. Here `committeeN` is different from the `committeeH` that we used in `OptimisticTrack()`, since they may have different sizes.

By Fact 1, OVERLAYBB must return the same block on all honest nodes. This is true even if f is large, or more precisely, as long as `committeeN` has at least one honest member. Hence if A_3 and A_4 were to directly use their respective return values from OVERLAYBB as their normal outputs, we would already have $b_3 = b_4$ as desired.

How to ensure $b_1 = b_3$. We also need to ensure that A_3's normal output b_3 agrees with A_1's fast output b_1. Recall that when A_1 generated b_1, A_1 must have had a valid `b_cert` for b_1. To ensure $b_3 = b_1$, every node that has generated a fast output will disseminate its fast output, together with the corresponding `b_cert`, to all nodes.

It is possible that no honest nodes have fast output, while some malicious node has it. The malicious node may then try to disseminate its fast output only to some honest nodes but not to others, causing inconsistency. To prevent such an attack, we need to use another *generalized* byzantine broadcast protocol called CERTBB, which we have designed and will discuss later, to disseminate the fast outputs (if any). In `NormalTrack()`, the invocation of CERTBB is done in parallel with the invocation of OVERLAYBB (see Figure 2). If node A_1 previously generated a fast output b_1, then A_1 will be a *certificate-broadcaster* in CERTBB, and will feed into CERTBB the hash of b_1 and the `b_cert` for b_1. CERTBB will then disseminate these to all nodes.

After CERTBB and OVERLAYBB both return, if node A_3 gets from CERTBB a return value containing a valid `b_cert`, then A_3 will directly adopt that return value as its normal output. Otherwise A_3 uses the return value from OVERLAYBB as its normal output. Namely, CERTBB overrides OVERLAYBB, as illustrated in Figure 2.

Summary on why $b_1 = b_3 = b_4$. Given our design so far, we give some quick intuitions on why $b_1 = b_3 = b_4$. We provide formal proofs later.

First, if some honest node A_1 generates a fast output b_1, then A_1 will be a certificate-broadcaster in CERTBB. By the guarantees of CERTBB (explained later), all honest nodes must get the hash of b_1 as the return value from CERTBB, which ensures all normal outputs (such as b_3 and b_4) must equal b_1.

Second, if no honest node has generated a fast output (which means there is no b_1), then it is still possible that some malicious nodes may act as certificate-broadcasters. We consider two cases, depending on the return value of CERTBB on honest nodes (this return value must be identical across all honest nodes by the guarantees of CERTBB):

- CERTBB does not return any valid value. Then all the honest nodes must use the return value from OVERLAYBB as their normal outputs. By the properties of OVERLAYBB in Fact 1, all such normal outputs (such as b_3 and b_4) must be identical.
- CERTBB returns the hash of some block x, together with a valid `b_cert`. All honest nodes will then take x as their normal outputs, and thus $b_3 = b_4 = x$.

D. Design of CERTBB

Goal of CERTBB. We now explain the design of CERTBB. CERTBB needs the system to provide a committee of nodes. For simplicity, CERTBB just re-uses `committeeN` as its committee. As long as `committeeN` has at least one honest member, we need CERTBB to guarantee:

- The return value from CERTBB must be the same on all honest nodes. This return value is allowed to be `null`.
- If some honest certificate-broadcaster feeds into CERTBB the hash of a fast output b_1, together with a corresponding `b_cert`, then CERTBB must return the hash of b_1 on every honest node.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

	want fast output?	want to adopt the block proposed by proposer?
Scenario 1: proposer is honest and $0 \leq f' \leq \alpha$	yes	yes
Scenario 2: proposer is malicious and $0 \leq f' \leq \alpha$	yes	do not care
Scenario 3: proposer is honest and $\alpha < f' \leq \beta$	do not care	yes
Scenario 4: proposer is malicious and $\alpha < f' \leq \beta$	do not care	do not care

TABLE III: Four scenarios and the desired behavior of FLINTBB. FLINTBB does not know which scenario it is in.

Central technical issue. These properties are closed related to the properties of OVERLAYBB in Fact 1. The key difference is that in CERTBB, there can be many certificate-broadcasters, while in OVERLAYBB, there is a single broadcaster which is the block proposer. Hence we also call CERTBB as a *generalized* byzantine broadcast protocol. Since all nodes in the system can be certificate-broadcasters, simply using one OVERLAYBB instance for each certificate-broadcaster will incur prohibitive overheads. Avoiding such overheads is hence the main technical issue.

Our idea. A central insight here is that every (honest and malicious) certificate-broadcaster must feed the hash of *the same block* b_1 into CERTBB, if it does feed something into CERTBB. The reason is that it is impossible for two different blocks b_1 and b_2 to both have valid `b_cert`. Different certificate-broadcasters may still feed different `b_cert`'s into CERTBB, since there can be multiple different `b_cert`'s for b_1. But all these `b_cert`'s are "equivalent", given that they are all for b_1.

This insight enables us to draw the following connection to OVERLAYBB, which assumes a single broadcaster. Imagine that in OVERLAYBB, there is a single virtual broadcaster X, who is malicious. The only value that X is allowed to broadcast is b_1, but X may choose to send b_1 to only a subset of the nodes. We do not view the `b_cert`'s as part of the object to be broadcast. Instead, we view a `b_cert` for b_1 as a "signature" on b_1 by X. Having different `b_cert`'s for b_1 simply means that X can generate different "signatures" for b_1, all of which are valid. Now imagine that X generates different "signatures" for b_1, and send those to a subset of the nodes. Those nodes will then correspond to the certificate-broadcasters in CERTBB.

The above conceptual connection shows that if we view the hash of b_1 as the object to be broadcast, and view the `b_cert`'s as "signatures" from the X, then the problem that CERTBB needs to solve is similar to OVERLAYBB. Hence we are able to obtain CERTBB, by adapting OVERLAYBB. For space constraints, we defer the complete pseudo-code and formal analysis on CERTBB to Appendix II.

E. Adopt Block Proposed by Honest Proposer

How to sure $b_1 = b$ and $b_3 = b$ (i.e., validity). Our design of `OptimisticTrack`() and `NormalTrack`() so far already ensures:

- When $f' \leq \beta$, all outputs of FLINTBB are identical. Namely, $b_1 = b_2 = b_3 = b_4$ in our running example.
- When $f' \leq \alpha$, FLINTBB generates fast output.

But the puzzle still misses its final piece: Let b be the block proposed by the block proposer. When the block proposer is honest, we need to further ensure that $b_1 = b_2 = b_3 = b_4 = b$. Namely, we want the block proposed by the honest proposer to be adopted.

To approach this issue in a systematic way, Table III exhaustively enumerates all possible scenarios. Our design so far achieves the desired behavior in all scenarios, except Scenario 3. In Scenario 3 where f' is large, the adversary may cause HotStuff to return the special (empty) block Ψ on all nodes, and to ignore the block b that the honest proposer has proposed. (Here Ψ is the only possibility other than b, since the adversary cannot fake the signature of the honest proposer.) With our current design, the honest nodes will then output Ψ instead of b, as fast outputs. We would then have $b_1 = b_2 = \Psi \neq b$.

Preventing Ψ from getting a `b_cert` in Scenario 3. To resolve this problem, we introduce a new kind of certificate called `p_cert`. A valid `p_cert` is needed, before a node contributes to the `b_cert` for the special block Ψ. In Scenario 3, the requirement of `p_cert` will eventually prevent Ψ from getting a valid `b_cert`. In turn, Ψ will not be used as a fast output, and all honest nodes will generate normal outputs, by adopting the return values of OVERLAYBB in the normal track. The guarantees of OVERLAYBB in Fact 1 then ensure that b is adopted. This means that all the outputs will equal b, as desired.

We now explain the specifics of `p_cert`. Recall that at the beginning of FLINTBB, the block proposer propagates its block to all nodes. If a member of `committeeH` does not receive this block, it will contribute to `p_cert`. Such contribution is done by the node generating a signature on the certificate type (i.e., `p_cert`), and then propagating that signature. A valid `p_cert` requires at least $\lceil t_p \cdot |\texttt{committeeH}| \rceil$ signatures from different members in `committeeH`, where t_p is the threshold.

As long as we set $t_p > \beta$, this requirement of `p_cert` will prevent Ψ from getting a valid `b_cert` in Scenario 3: In that scenario, the proposer is honest. Thus all honest members in `committeeH` must receive the block from the proposer, and will not contribute to `p_cert`. Since $f' \leq \beta < t_p$, the malicious members in `committeeH` will not be sufficient to form a valid `p_cert`. In turn, honest nodes will never contribute to the `b_cert` of Ψ. Since a valid `b_cert` must contain contribution from at least one honest node, Ψ can never get a valid `b_cert`.

Still ensuring fast output in Scenario 2. We want to ensure that requiring `p_cert` does not cause new problems in other scenarios in Table III. It turns out that Scenario 2 may be affected. In Scenario 2, we do not care whether the block proposed by the (malicious) proposer is adopted. But we do

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

want a fast output, and the requirement of p_cert may hinder such fast output. We next explain how to avoid this problem, by setting $t_p \leq 1-\alpha$. This ultimately means that t_p should be chosen such that $\beta < t_p \leq 1-\alpha$, as summarized in Table II.

In Scenario 2, the malicious block proposer may choose not to send its block to honest nodes. We consider two cases. First, if no honest member in committeeH receives any block from the proposer, then there will be at least $1-f' \geq 1-\alpha \geq t_p$ fraction of the members in committeeH contributing to p_cert, resulting in a valid p_cert. Then in this case, the requirement of p_cert will not hinder anything.

Second, if some honest member A_1 in committeeH receives a block from the malicious proposer, then the remaining honest members may or may not be sufficient to form a p_cert, which would be a problem. But intuitively, the block received by A_1 is already a potential candidate for fast output, even if Ψ cannot be used due to the lack of p_cert. Based on this intuition, we now let nodes in committeeH invoke HotStuff as follows. When invoking HotStuff, a node feeds into HotStuff exactly one of the following, in decreasing order of precedence:

1) Ψ together with a valid p_cert, if the node has collected a valid p_cert.
2) A block signed by the proposer, if the node has received such a block.
3) Some arbitrary value such as 0.

We call the first two kinds of values in the above list as *usable values* — namely, these are eligible values for the contribution of b_cert. We further invoke HotStuff (see Appendix I) in a way such that: If at least one honest node feeds a usable value into HotStuff, then HotStuff must return some usable value. With such a design, given that A_1 feeds a usable value into HotStuff and since f' is small in Scenario 2, HotStuff must return a usable value on all honest nodes. All honest nodes will contribute to the b_cert of this usable value, resulting in a valid b_cert. This then leads to fast output. We have further designed some additional optimizations to ensure that our above approach will be efficient – see Appendix I.

Taking all 4 scenarios into account. We have discussed Scenario 2 and 3 in Table III so far. Our design works, without the need of further adjustment, for Scenario 1 and 4 in Table III. Our security analysis later will cover all 4 scenarios.

VI. Security Analysis

To make our security analysis rigorous, we provide the complete pseudo-code for FLINTBB in Algorithm 1 through 3. The pseudo-code is based on the design in the previous section. Let κ be a security parameter. We define *negligible probability* to be any probability that is $exp(-\Omega(\kappa))$. As in typical security analysis, we consider executions of polynomial length with respect to κ. FLINT uses two committees for each slot, committeeH and committeeN. We set their sizes to be no smaller than κ. Following our model as inherited from BCUBE [15] (see Section III), our analysis assumes that the mildly-adaptive adversary does not have sufficient time to adaptively compromise the committee members, after the corresponding beacon (for choosing those members) was revealed and before the committee finishes doing its work.

We first reason about FLINTBB. The next theorem shows that when $f' \leq \beta$, FLINTBB guarantees security, in the form

Algorithm 1 FLINTBB (proposer, committeeH, committeeN).

1: proposer constructs a block;
2: fast_output.block $\leftarrow$ null; fast_output.b_cert $\leftarrow$ null;
3: let timeout be a value such that timeout $> 6\Delta + y$, where y is the time needed for HotStuff to return at Line 22 when f' is below $\frac{1}{3}$; // A fast output may potentially be generated earlier than timeout.
4: spend timeout time in executing the next two lines:
5: fast_output$\leftarrow$OptimisticTrack(proposer, committeeH);
6: output fast_output.block; // this is fast output
7: // If the above execution finishes early, wait until time out. If it does not finish in time, simply stop and proceed to the next line.
8: NormalTrack(proposer, committeeN, fast_output);

Algorithm 2 OptimisticTrack(proposer, committeeH).

9: **/* Nodes in committeeH execute Line 10–38. */**
10: proposer propagates its block (with its signature);
11: wait for Δ time;
12: **if** I have not received any block signed by the proposer **then** contribute to p_cert;
13: wait for Δ time;
14: **if** I get a valid p_cert **then**
15: input $\leftarrow \Psi$|p_cert;
16: **else if** I previously received a block at Line 12 **then**
17: input $\leftarrow$ block received at Line 12; // Tie-breaking can be done arbitrarily, if multiple blocks were received.
18: **else**
19: input $\leftarrow$ 0; // Here input can be an arbitrary value.
20: **end if**
21: run the following two lines in parallel:
22: result $\leftarrow$ HotStuff(input); // By discussions in Appendix I, here HotStuff must return either a block with proposer's signature, or Ψ|p_cert, or Ψ|c_cert.
23: propagate input if input is usable value, and adopt received usable value as input, by the design in Appendix I;
24: **if** result is some block **then**
25: $b \leftarrow$ result;
26: **else if** result is Ψ|p_cert or Ψ|c_cert **then**
27: $b \leftarrow \Psi$;
28: **end if**
29: contribute to the a_cert of b;
30: wait until I get a valid a_cert of b.
31: // Propagate a_cert to declare intention to contribute to b_cert.
32: propagate the a_cert of b;
33: wait for Δ time;
34: **if** I do not see any a_cert corresponding to any value different from b **then** contribute to the b_cert of b;
35: wait until I see a valid b_cert of b;
36: output.block $\leftarrow b$; output.b_cert $\leftarrow$ the valid b_cert;
37: propagate output;
38: **return** output;
39:
40: **/* Nodes not in committeeH execute Line 41–42. */**
41: wait until I receive a valid fast output, namely, a block (potentially Ψ) together with a corresponding b_cert;
42: **return** that fast output;

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3 NormalTrack(proposer, committeeN, fast_output).

43: run the following two lines in parallel and wait until both return:
44: $b \leftarrow$ OVERLAYBB (proposer, committeeN);
45: $h \leftarrow$ CERTBB (committeeN, fast_output);
46: // see Appendix II for the pseudo-code of CERTBB
47: **if** I previously generated a fast output at Line 6 **then** already_output $\leftarrow$ true **else** already_output $\leftarrow$ false;
48: // We will prove that honest nodes must have the same h here.
49: **if** $h =$ null **then**
50: output the block b, if not already_output; // normal output
51: **else if** $h = \text{hash}(\Psi)$ **then**
52: output Ψ, if not already_output; // normal output
53: **else**
54: propagate the block corresponding to the blockhash h, if I have that block;
55: wait until the block with blockhash h is received;
56: output that block, if not already_output; // normal output
57: **endif**

of **Termination**, **Agreement**, and **Validity**:

Theorem 2. *Consider any execution of* FLINTBB *(i.e., Algorithm 1), with α and β such that $0 \leq \alpha < \min(1-\beta, \frac{1}{3}) \leq 0.5 \leq \beta < 1$, and where all honest nodes invoke* FLINTBB *with the same parameters (i.e.,* proposer*,* committeeH*, and* committeeN*). Let f' be the fraction of malicious members in* committeeH*. If $f' \leq \beta$ and if* committeeN *contains at least one honest member, then:*

- *(***Termination***) Each honest node must eventually generate either a fast output or a normal output (but not both).*
- *(***Agreement***) The output, regardless of fast or normal, must be the same on all honest nodes.*
- *(***Validity***) If* proposer *is honest, and if its block is received by all honest nodes at Line 12 of Algorithm 2, then all honest nodes must output the block proposed by* proposer*.*

Proof. Deferred to our technical report [1]. □

The next theorem shows that when $f' \leq \alpha$, FLINTBB must generate a fast output:

Theorem 3. *Under the same conditions as in Theorem 2 and if we further have $f' \leq \alpha$, then* FLINTBB *must generate a (fast) output on all honest nodes at Line 6 in Algorithm 1.*

Proof. Deferred to our technical report [1]. □

With the above properties of FLINTBB, we can now reason about the FLINT blockchain itself. The following theorem shows that i) FLINT always guarantees security (i.e., **Safety** and **Liveness**) when $f \leq f_{\max}$, and ii) FLINT generates fast confirmation for each block when $f \leq \min(1 - f_{\max}, \frac{1}{3}) - c$.

Theorem 4 (Final Guarantees of FLINT)**.** *Consider any given constant $c > 0$ and $f_{\max} \in [\frac{1}{2}, 0.99]$. We can always find a configuration (i.e., values for α and β) of* FLINT *such that except with negligible probability:*

- *If $f \in [0, f_{\max}]$, then* FLINT *guarantees for every slot:*
  - **Safety***: For any two honest nodes A_1 and A_2, if A_1 and A_2 both have a confirmed block for that slot, then their confirmed blocks must be the same.*
  - **Liveness***: For any honest node A_1, after A_1 invokes* FLINTBB *for that slot, A_1 must eventually have a confirmed block in that slot. If the block proposer in that slot is honest, and if the proposer's block is received by all honest nodes by Line 12 of Algorithm 2, then the confirmed block for that slot must be the proposer's block.*
- *If f further satisfies $f \in [0, \min(1 - f_{\max}, \frac{1}{3}) - c]$, then* FLINT *further guarantees for every slot:*
  - **Fast-confirmation***: An honest node must be able to obtain (fast) confirmation for a block in that slot, namely, a (fast) output at Line 6 in Algorithm 1.*

Proof. See Appendix IV. □

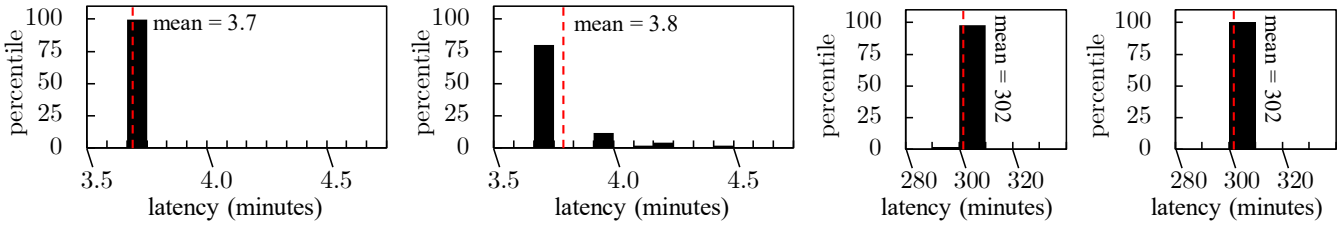
Remark. It is interesting to note that the security of FLINT, namely **Safety** and **Liveness**, only relies on the timely delivery of messages sent at Line 32, Line 44, and Line 45 in our pseudo-code. This is because, as one can trivially but tediously verify, the proofs for these two properties do not rely on other messages. The timely delivery of those other messages is only for **Fast-confirmation**.

VII. IMPLEMENTATION AND EXPERIMENTAL RESULTS

A. Experimental Methodology

The goal of our experiments is to do a direct comparison with BCUBE [15], in terms of block confirmation latency and system throughput. To this end, we have implemented a FLINT prototype in Go, using TCP for communication. As part of FLINT, we have also implemented HotStuff [32], OVERLAYBB [15], and CERTBB. We keep our experimental settings the same as BCUBE's, whenever possible. Specifically, since beacon generation takes at least a day, our experiments directly inject random beacons, and we did not implement the beacon generation mechanism. BCUBE's experiments [15] also directly inject random beacons. We run our experiment on 25 high-end PCs in a local-area network. To run as many as FLINT nodes as possible, each PC runs 400 FLINT nodes as Go routines. We have ensured that these Go routines do not interact with each other via the shared memory space. We have implemented a throttling mechanism at the application layer, to ensure that each FLINT node consumes no more than 20Mbps bandwidth. This is the same as the 20Mbps available bandwidth in [15]. Due to CPU constraints and same as [15], we replace all cryptographic functions with dummy functions of the same output size. We use block size of 2MB, which is the same block size as in BCUBE. We use dummy data for the transactions. Typical Bitcoin transactions are of 0.5KB size — but we use a larger transaction size of about about 2.5KB, to reduce CPU overhead and to allow 400 FLINT nodes on each PC. Transaction size actually does not affect our final results. We let each FLINT node always hold 1 stake/coin.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.



(a) $f = 0.02$ (b) $f = 0.2$ (c) $f = 0.3$ (d) $f = 0.6$

Fig. 3: Histogram of block confirmation latency values in FLINT, under $f_{\max} = 0.6$ and different f values.

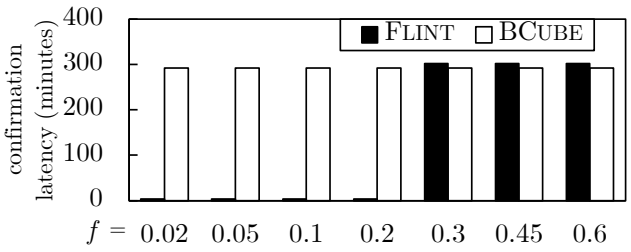


Fig. 4: Block confirmation latency under $f_{\max} = 0.6$.

The overlay network is built in the same way as in BCUBE's experiments [15]. Specifically, each node A_1 picks 20 random nodes, that are not on the same physical machine as A_1, to establish edges to. Each edge, once established, will be undirected. Each node will accept up to 22 such edge-establishment requests, after which it will decline. Hence each node has at least 20 edges and at most 42 edges. Same as BCUBE's experiments, we use $d = 6$ in our experiments. We set `timeout` in Algorithm 1 to be 10 minutes. For the propagation of low priority messages, instead of sending the message immediately to up to 42 neighbors at once, a node sends the message to 4 neighbors at a time. This helps to reduce duplicates.

Following [15], our experiments focus on $f_{\max} = 0.5$, 0.6, and 0.7. In all our experiments, the malicious nodes drop all messages. All results for FLINT in the next are from our experiments using our prototype. Results for BCUBE are directly quoted from [15] — we did not re-run BCUBE experiments ourselves. Nevertheless, since the experimental settings for FLINT and BCUBE are almost exactly identical, these results are directly comparable.

FLINT has a number of internal parameters, such as the values of α and β. We choose all these internal parameters in a systematic way, rather than as "magic numbers". For space constraints, we defer the details of the parameter choosing to Appendix V.

B. Results on Block Confirmation Latency

We obtain the confirmation latency of FLINT in our experiments, by measuring the average block confirmation latency of the first 50 blocks in the blockchain on all honest nodes. Figure 4 compares such confirmation latency against that of BCUBE, under $f_{\max} = 0.6$. Note that BCUBE's confirmation latency, by design, does not depend on f. One can see that for all $f \leq 0.2$, FLINT achieves a confirmation latency drastically smaller than BCUBE. For example, for $f = 0.2$, FLINT gives a confirmation latency of about 3.8 minutes, which is about 76 times smaller than the confirmation latency of about 292 minutes in BCUBE. For other f values below 0.2, FLINT reduces confirmation latency by a similar factor.

For larger f between 0.3 and 0.6, FLINT no longer provides fast confirmation. In fact, because of the extra optimistic track in FLINT, its confirmation latency is slightly higher than BCUBE's. But the difference is rather small — only about 3%.

To provide deeper insights, Figure 3 further plots the histogram of the confirmation latency values. Namely, for each honest node, there are 50 confirmation latency values, one for each of the 50 blocks. We collect all these values from all honest nodes, and Figure 3 plots the histogram for all such values altogether.

Figure 3(a) and Figure 3(b) correspond to fast confirmation under $f = 0.02$ and $f = 0.2$, respectively. In both figures, most confirmation latency values concentrate around 3.7 minutes. Such small variance is obviously desirable. In Figure 3(b), there are small groups of confirmation latency values around 3.9, 4.2, and 4.4 minutes. These are due to view changes in HotStuff. Specifically, the first group of values result from having one view change in HotStuff, due to the first leader in HotStuff being malicious. Each view change, including the time out, takes about 15 seconds. The second group is due to the first two leaders in HotStuff being malicious, and so on. The probability of the first x leaders in HotStuff being malicious drops exponentially with x. Hence the group size quickly decreases.

Similarly, Figure 3(c) and Figure 3(d) plot the histograms for $f = 0.3$ and $f = 0.6$. These are cases where FLINT provides normal confirmation. The confirmation latency here is largely determined by how long the normal track takes to complete, which is independent of f. Hence these two figures are similar.

Finally, the above results are all for $f_{\max} = 0.6$. We have also obtained similar results under $f_{\max} = 0.7$ and $f_{\max} = 0.5$, which are not included in the above figures, due to space constraints. For example for $f_{\max} = 0.7$, FLINT provides fast confirmation in about 3.7 minutes for all $f \leq 0.1$. This is about $352/3.7 \approx 95$ times smaller, compared to BCUBE's confirmation latency of about 352 minutes under $f_{\max} = 0.7$.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

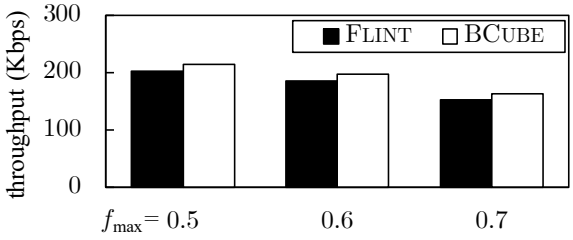


Fig. 5: Throughput under different $f_{\max}$.

C. Results on Throughput

FLINT does not aim to improve the throughput of BCUBE. In fact, because of the extra optimistic track in FLINT, FLINT will achieve slightly lower throughput than BCUBE. Our experiments here aim to quantify this. To do so, we first wait until FLINT generates the first block, then measure the time needed to generate the next 50 blocks, and finally obtain the throughput by dividing the total size of these 50 blocks by the time duration. Following common practices (e.g., [5, 7, 8, 11]) and also BCUBE's experiments [15], all nodes behave honestly in these experiments.

Figure 5 compares the throughput of FLINT and that of BCUBE. As expected, FLINT offers lower throughput than BCUBE. But the difference is relatively small, around 6% in all three cases. We feel that this is a small price to pay, for the up to around 95 times reduction in confirmation latency that FLINT offers in the common case when f is relatively small.

VIII. Discussions on External Verifiability

Section III explained that when $f \geq \frac{1}{2}$, FLINT does *not* provide external/public verifiability. This section explains how to use FLINT in two application scenarios, despite that it does not offer external/public verifiability. We further discuss how a more nuanced version of external/public verifiability could potentially be offered by FLINT.

Application: Hybrid/consortium blockchain for SCN. Supply chain management is concerned with the flow of materials/products/services among various businesses (firms). The number of firms in a global *supply chain network* (SCN) can easily reach thousands [49, 50]. Blockchains offer promising opportunities to reduce the cost, improve the coordination, and enhance the traceability in supply chains [51, 52, 53]. To exploit such opportunities, the firms in the SCN can form a *hybrid/consortium blockchain*, where each firm runs one or several full-nodes in the blockchain.[3]

In terms of security requirements, such a blockchain has some unique characteristics:

- The firms in SCN often do not trust each other [54]. For example, they may be competitors of each other, or they may be in different jurisdictions. In such a context, having a large $f_{\max}$ as in FLINT is important. A large $f_{\max}$ could mean, for example, even if firms in all other jurisdictions behave maliciously, all the firms in the local jurisdiction will still have a *consistent* view on the blockchain state as long as they are all honest.
- A firm in SCN usually trusts itself (i.e., its own nodes). Furthermore, each firm is a business and can easily afford to maintain a full-node. In such a context, if a firm has some employees as light-clients, such light-clients should all interact with FLINT via the firm's own full-node, which is trusted by the firm itself.

Note that the trust model here is rather different, from the trust model in a typical application scenario of state machine replication (SMR) [48]. In SMR, a single firm maintains multiple servers to provide a replicated service to clients. There, the main goal is to guard against the potential byzantine failure of *each* of the servers, and the firm does not trust any one server more than the others.

Application: Cryptocurrency system. External verifiability is usually needed in cryptocurrency systems to support light-clients. To use FLINT in cryptocurrency systems, we will need a light-client to obtain the blockchain state from a full-node that the light-client trusts. Note that different light-clients can trust different full-nodes, and hence no trusted central authority is needed. For example, a light-client may choose to trust the full-node maintained by some local financial firm licensed in the client's local jurisdiction. Security is guaranteed for that client, as long as the full-node that it trusts is not malicious.

A more nuanced version of external verifiability. Section III explained that when $f \geq \frac{1}{2}$, it is impossible to provide external verifiability. While we do aim to tolerate a malicious majority, one would imagine that in practice, it is unlikely for f to be always above $\frac{1}{2}$. For example, f may be 0.6 due to a large-scale attack, and then reverts back to 0.1 after the attack is discovered and measures are taken. Our security analysis have already shown that FLINT will guarantee **Safety** and **Liveness** throughout such a scenario.

Interestingly, there is also possibility for FLINT to potentially provide a more nuanced version of external verifiability, in the above scenario. Specifically, FLINT could potentially offer external verifiability, if f is currently 0.1 (and despite that previously f was 0.6). Roughly speaking, the intuition is that when $f = 0.1$, the optimistic track will succeed, and each confirmed block will have an accompanying `b_cert`. Recall that a valid `b_cert` must contain the contribution from at least one honest node A, assuming that the beacon and the stake distribution have limited bias. By the **Safety** property of FLINT, the blockchain state on this honest node A must be correct. We can further extend the `b_cert` to include a hash of previously confirmed blocks. In this way, a valid `b_cert` could potentially prove to a light-client that the blockchain state corresponding to this `b_cert` is correct. We leave the detailed design of such a mechanism to future work.

IX. Related Works

Blockchains can be viewed as a kind of BFT (byzantine fault-tolerant) systems. For BFT systems, there have been

[3] At blockchain bootstrapping time, stakes can be allocated to the various firms (or a smaller set of founding members), which correspond to their respective weightage. Later on, a stake can be split or be transferred from firm to firm, as needed and via transactions on the blockchain, to allow fine-grained and dynamic adjustment of weightage.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

many interesting works on designing *optimistic tracks* for the common cases, where certain *optimistic conditions* are satisfied.

Blockchain/BFT systems for $f_{\max} < 0.5$. There is a large body of works [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] on designing BFT systems for $f_{\max} < 0.5$ and that come with an optimistic track. Obviously, none of these can be directly used to achieve our goal, since we aim for $f_{\max} \geq 0.5$. One may wonder whether it is possible to adapt these designs to tolerate $f_{\max} \geq 0.5$, for example, by replacing their normal track with OVERLAYBB [15]. It turns out that in most of these designs [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34], the optimistic track and the normal track are tightly-coupled, making replacement infeasible. A few other designs [35, 36, 37] have somewhat weaker coupling. But the optimistic tracks in these designs all require the leader to be honest [35, 37] (or even all nodes to be honest [36]). This turns out to be a significant drawback, under our setting of $f_{\max} \geq 0.5$, as explained in Section II.

There has also been a separate line of research on *early-stopping* byzantine agreement/broadcast protocols [55, 56, 57, 58, 59, 60, 61, 62], which can run faster if f is smaller than $f_{\max}$. With the exceptions [61, 62], these protocols fundamentally do not aim to deal with $f_{\max} \geq 0.5$. The protocols in [61, 62] do consider $f_{\max} \geq 0.5$. However, the optimistic track in [61] is only constant number of rounds faster than the normal track, and hence has limited practical relevance. The optimistic path in [62] requires the leader to be honest — again, Section II explained why this is a significant drawback.

Blockchain/BFT systems for $f_{\max} \geq 0.5$. There are a few prior blockchain/BFT designs [14, 38] that can tolerate $f_{\max} \geq 0.5$ and that come with an optimistic track.

Thunderella [14] mentions a theoretical construction of such a blockchain, without implementation/experiments. In their construction, there is a special *leader* node. For their optimistic track to succeed, the leader needs to be honest.[4] A malicious leader can force Thunderella to resort to the normal track. Section II already explained why requiring the leader to be honest is a significant drawback. One way to avoid this problem is to stick to a good leader, until it becomes bad. But doing so directly contradicts with the key *decentralization* benefit of using blockchains. In particular, the leader does have some privileges. For example, the leader may favor certain transactions when proposing new blocks, in a stealthy way to avoid being classified as "bad". In comparison, our optimistic track in FLINT succeeds *as long as f is relatively small* (namely $f \leq \min(1 - f_{\max}, \frac{1}{3}) - c$), and *we do not need any specific node to be honest*. Finally, we note that Thunderella's optimistic track, if it does succeed, incurs fewer number of rounds than the optimistic track in FLINT. Hence there is potential to further combine the instantaneous confirmation in Thunderella with our FLINT design. Doing so would add an extra "super-optimistic track" into FLINT, beyond the current optimistic/normal track. We leave the details to future work.

Liu-Zhang et al. [38] have explored designing optimistic track in secure multi-party computation (MPC). While they do consider $f_{\max} \geq 0.5$, their protocol is for a rather different context. Their design involves running one byzantine agreement instance for each party in MPC. If their design were mapped to our blockchain context, then confirming a block would involve running one byzantine agreement instance for each member of the committee, which would be prohibitively expensive. Their design further uses strong crypto primitives such as fully homomorphic encryption. These primitives typically require trusted setup, and it is unclear how to do so in our context.

[4] In addition, their optimistic track further requires f to be no larger than $(1 - f_{\max})/2$. Roughly speaking, we only need f to be smaller than $1 - f_{\max}$ and we do not need any leader to be honest.

X. CONCLUSIONS

This work has presented FLINT, a novel blockchain that can tolerate $f_{\max} \geq \frac{1}{2}$ and can give optimistic execution (i.e., fast confirmation) whenever f is relatively small. We have proved such properties via a formal security analysis, and also demonstrated these properties experimentally via an implementation. We believe that FLINT achieves an important step forward, in facilitating the broader application/adoption of blockchains that can tolerate a malicious majority.

ACKNOWLEDGMENT

We thank the anonymous IEEE Symposium on Security and Privacy reviewers for their detailed and helpful comments on this paper.

DISCLOSURE BY AUTHORS

Ruomu Hou is currently a Research Assistant and also a Ph.D. student in School of Computing, National University of Singapore. Haifeng Yu is currently an Associate Professor in School of Computing, National University of Singapore.

REFERENCES

[1] R. Hou and H. Yu, "Optimistic fast confirmation while tolerating malicious majority in blockchains," School of Computing, National University of Singapore, Tech. Rep., 2023, also available at https://dl.comp.nus.edu.sg/handle/1900.100/12.

[2] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling Blockchain via Full Sharding," in *CCS*, 2018.

[3] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *International Conference on Financial Cryptography and Data Security*, 2019.

[4] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO*, 2017.

[5] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *SOSP*, 2017.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *EuroSys*, 2018.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

[7] H. Yu, I. Nikolic, R. Hou, and P. Saxena, "OHIE: Blockchain Scaling Made Simple," in *IEEE Symposium on Security and Privacy*, 2020.

[8] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *CCS*, 2016.

[9] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," in *IEEE Symposium on Security and Privacy*, 2018.

[10] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the Blockchain to Approach Physical Limits," in *CCS*, 2019.

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *CCS*, 2016.

[12] T.-H. H. Chan, R. Pass, and E. Shi, "Sublinear-round byzantine agreement under corrupt majority," in *IACR International Conference on Public-Key Cryptography*, 2020.

[13] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[14] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *EUROCRYPT*, 2018.

[15] R. Hou, H. Yu, and P. Saxena, "Using throughput-centric byzantine broadcast to tolerate malicious majority in blockchains," in *IEEE Symposium on Security and Privacy*, 2022.

[16] C. Ganesh and A. Patra, "Broadcast extensions with optimal communication and round complexity," in *PODC*, 2016.

[17] M. Hirt and P. Raykov, "Multi-valued byzantine broadcast: The $t < n$ case," in *ASIACRYPT*, 2014.

[18] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang, "Improved extension protocols for byzantine broadcast and agreement," *arXiv preprint arXiv:2002.11321*, 2020.

[19] G. Tsimos, J. Loss, and C. Papamanthou, "Nearly quadratic broadcast without trusted setup under dishonest majority," *IACR Cryptology ePrint Archive*, 2020.

[20] J. Wan, H. Xiao, E. Shi, and S. Devadas, "Expected constant round byzantine broadcast under dishonest majority," in *Theory of Cryptography Conference*, 2020.

[21] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Good-case latency of byzantine broadcast: a complete categorization," in *PODC*, 2021.

[22] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Symposium on Operating Systems Design and Implementation*, 1999.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *SOSP*, 2007.

[24] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[25] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: a scalable and decentralized trust infrastructure," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019.

[26] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: Redundant byzantine fault tolerance," in *ICDCS*, 2013.

[27] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *EuroSys*, 2010.

[28] C. Stathakopoulou, T. David, and M. Vukolić, "Mir-BFT: High-throughput BFT for blockchains," *arXiv preprint arXiv:1906.05552*, 2019.

[29] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: Resilient concurrent consensus for high-throughput secure transaction processing," in *IEEE International Conference on Data Engineering*, 2021.

[30] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *IEEE Symposium on Security and Privacy*, 2020.

[31] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *CCS*, 2019.

[32] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *PODC*, 2019.

[33] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, 2016.

[34] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[35] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Brief announcement: Byzantine agreement, broadcast and state machine replication with optimal good-case latency," in *DISC*, 2020.

[36] K. Kursawe, "Optimistic byzantine agreement," in *IEEE Symposium on Reliable Distributed Systems*, 2002.

[37] A. Momose, J. P. Cruz, and Y. Kaji, "Hybrid-BFT: Optimistically responsive synchronous consensus with optimal latency or resilience." *IACR Cryptol. ePrint Arch. Report 2020/406*, 2020.

[38] C.-D. Liu-Zhang, J. Loss, U. Maurer, T. Moran, and D. Tschudi, "MPC with synchronous security and asynchronous responsiveness," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2020.

[39] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, https://bitcoin.org/bitcoin.pdf.

[40] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014, https://ethereum.github.io/yellowpaper/paper.pdf.

[41] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *IEEE Symposium on Security and Privacy*, 2017.

[42] M. Tran, I. Choi, G. J. Moon, V.-A. Vu, and M. S. Kang., "A stealthier partitioning attack against bitcoin peer-to-peer network," in *IEEE Symposium on Security and Privacy*, 2020.

[43] S. Bojja Venkatakrishnan, G. Fanti, and P. Viswanath, "Dandelion: Redesigning the bitcoin network for anonymity," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2017.

[44] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *USENIX Security Symposium*, 2015.

[45] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "Sok: Blockchain light clients," in *International Conference on Financial Cryptography and Data Security*, 2022.

[46] A. Kiayias, N. Lamprou, and A.-P. Stouka, "Proofs of proofs of work with sublinear complexity," in *International Conference on Financial Cryptography and Data Security*, 2016.

[47] S. Agrawal, J. Neu, E. N. Tas, and D. Zindros, "Proofs of Proof-of-Stake with Sublinear Complexity," 2022, arxiv preprint, arXiv:2209.08673.

[48] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[49] K. Lavassani, Z. Boyd, B. Movahedi, and J. Vasquez, "Tentier and multi-scale supplychain network analysis of medical equipment: Random failure and intelligent attack analysis," 2021, arxiv preprint, arXiv:2104.14046.

[50] J. Wu and J. Birge, "Supply chain network structure and firm returns," *SSRN Electronic Journal*, Jan. 2014.

[51] P. Dutta, T.-M. Choi, S. Somani, and R. Butala, "Blockchain technology in supply chain operations: Applications, challenges and research opportunities," *Transportation Research Part E: Logistics and Transportation Review*, vol. 142, 2020.

[52] V. Gaur and A. Gaiha, "Building a transparent supply chain," *Harvard Business Review*, 2020.

[53] R. Cole, M. Stevenson, and J. Aitken, "Blockchain technology:

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

implications for operations and supply chain management," *Supply Chain Management*, Jun. 2019.
[54] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "Permissioned blockchains: Properties, techniques and applications," in *SIGMOD*, 2021.
[55] D. Dolev, R. Reischuk, and H. R. Strong, "Early stopping in byzantine agreement," *JACM*, vol. 37, no. 4, pp. 720–741, 1990.
[56] P. Berman, J. A. Garay, and K. J. Perry, "Optimal early stopping in distributed consensus," in *International Workshop on Distributed Algorithms*, 1992.
[57] I. Abraham and D. Dolev, "Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity," in *STOC*, 2015.
[58] G. Goren and Y. Moses, "Byzantine consensus in the common case," *arXiv preprint arXiv:1905.06087*, 2019.
[59] A. W. Krings and T. Feyer, "The byzantine agreement problem: optimal early stopping," in *Hawaii International Conference on Systems Sciences*, 1999.
[60] J. A. Garay and Y. Moses, "Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 247–290, 1998.
[61] M. Fitzi and J. Nielsen, "On the number of synchronous rounds sufficient for authenticated byzantine agreement," in *DISC*, 2009.
[62] T. Albouy, D. Frey, M. Raynal, and F. Taiani, "Good-case early-stopping latency of synchronous byzantine reliable broadcast: The deterministic case," in *DISC*, 2022.
[63] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT*, 2018.

APPENDIX I: HOW FLINTBB INVOKES HOTSTUFF

In `OptimisticTrack()`, all nodes in `committeeH` invoke HotStuff [32] to get a return value.[5] We call this return value as the *one-shot decision* from HotStuff. Strictly speaking, the HotStuff protocol [32] actually generates a *sequence of decisions*. We next explain how to get the one-shot decision from HotStuff.

Getting one-shot decision. For HotStuff to generate the sequence of decisions, each potential decision is coordinated by a *leader*, which means there is a sequence of leaders. If the leader is honest, HotStuff is guaranteed to adopt that leader's input as the decision. If the leader is malicious, then HotStuff may either get a decision, or may fail to get any decision, in which case HotStuff does a view change and proceeds to the next leader.

In our context, we invoke HotStuff on `committeeH`, which is chosen randomly by the beacon. The committee members already have a (random) ordering among themselves, when chosen. We simply use the ordered members in `committeeH` as the sequence of leaders needed by HotStuff.

To get the one-shot decision, each node simply runs HotStuff to first get the sequence of decisions. Recall from Section V-E the definition of usable values. The very first *usable value* in that sequence of decisions will be the one-shot decision that we need. This one-shot decision is guaranteed to be generated (if it has not yet been), when an honest node A_1 becomes the leader in HotStuff and if A_1 feeds a usable value as its input into HotStuff. Hence, if at least one honest node feeds a usable value as its input into HotStuff, then HotStuff must return some one-shot decision, which is a usable value.

On a given node, after HotStuff generates the one-shot decision, HotStuff will continue to run in the background to help other nodes to generate their one-shot decisions. HotStuff will be stopped when a fast output is generated, or when the optimistic track times out, whichever is earlier.

A simple optimization. We further use a simple optimization to get the one-shot decision faster, by letting nodes without usable values adopt usable values as their inputs. Specifically, let v_1 and v_2 be the originally-intended input values to the invocations of HotStuff by nodes A_1 and A_2, respectively. HotStuff asks for the input value from a node, only when that node becomes the leader in HotStuff. Now imagine that v_1 is a usable value, while v_2 is not. In parallel with the execution of HotStuff, A_1 will propagate v_1 on the overlay network. If A_2 receives this v_1 before A_2 becomes the leader in HotStuff, then A_2 will later feed v_1 instead of v_2 into HotStuff when A_2 becomes the leader. Note that this whole mechanism does not involve any changes to the internals of HotStuff.

There are two kinds of usable values so far: i) Ψ together with a valid `p_cert`, and ii) a block signed by the proposer. During the propagation of the usable values, when a node sees two different blocks both signed by the proposer, the node immediately forms a `c_cert` from those two (conflicting) signatures, and will now use $\Psi|$`c_cert` as its input to HotStuff. This `c_cert` is a *conflict certificate*, which can convince other nodes that the proposer is malicious. When the proposer is malicious, we can always safely use Ψ as the fast output. Hence the special block Ψ together with a valid `c_cert` now becomes the third kind of usable value. The node creating the `c_cert` will also propagate $\Psi|$`c_cert` to all other nodes. Finally, `c_cert` has a smaller size than `p_cert`, while `p_cert` has a smaller size than a block. Hence if a node receives multiple usable values (i.e., $\Psi|$`c_cert`, $\Psi|$`p_cert`, and blocks signed by proposer) during the propagation of usable values, for better performance, it simply adopts the smallest-sized usable value among those.

The above simple optimization has been fully taken into account, in our pseudo-code and our formal security analysis in Section VI.

APPENDIX II: PSEUDO-CODE AND FORMAL GUARANTEES OF CERTBB

Recall from Section V-D that we obtain CERTBB, by adapting OVERLAYBB. The adaptation will be relative simple, if properly driven by the key conceptual connection in Section V-D. Algorithm 4 provides the complete pseudo-code of CERTBB. For convenience in the pseudo-code, for any value h, we view `null` as a valid aggregate signature on h with zero signer. Most of the steps in Algorithm 4 are

[5]When sending messages in HotStuff, nodes in `committeeH` communicate directly, rather than via the overlay network. To do so, the nodes in `committeeH` propagate their IP addresses first and then run HotStuff.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

Algorithm 4 CERTBB (committeeN, fast_output). // This algorithm requires a precondition: Throughout its execution, there is at most one value for which any honest node sees a valid b_cert for that value. We will prove that when we invoke this algorithm, this precondition always holds.

accepted $\leftarrow$ null; $m \leftarrow |$committeeN$|$;
if fast_output.block $\neq$ null **then**
  $h \leftarrow$ hash(fast_output.block);
  send $\langle h,$ fast_output.b_cert, null$\rangle$ to myself;
end if
// The following loop consists of round 1 through $2d(m+1)$.
for $t \leftarrow 1$; $t \leq 2d(m+1)$; t++ **do**
  receive messages in the form of $\langle$hash, cert, sig$\rangle$ from all my neighbors and myself;
  discard a message if cert is not valid b_cert for hash, or if sig is not valid signature on hash; // null is viewed as a valid signature with 0 signer
  among all remaining messages, let $\langle \text{hash}_0, \text{cert}_0, \text{sig}_0 \rangle$ be the message where sig_0 has the largest number of signers (with arbitrary tie-breaking);
  if (I am in committeeN) and $(2d(|\text{sig}_0|+1) \geq t)$ **then**
    accepted $\leftarrow \text{hash}_0$;
    $\text{sig}_1 \leftarrow$ combine sig_0 and my own signature on hash_0;
  end if
  if (I am not in committeeN) and $(2d(|\text{sig}_0|+1) \geq t+d)$ **then**
    accepted $\leftarrow \text{hash}_0$;
    $\text{sig}_1 \leftarrow \text{sig}_0$;
  end if
  send $\langle \text{hash}_0, \text{cert}_0, \text{sig}_1 \rangle$ to all my neighbors;
  wait for δ time;
end for
return accepted;

directly from OVERLAYBB [15].[6] Instead of reasoning about these individual steps, we directly give the final guarantees of CERTBB in the following lemmas, whose proofs are deferred to our technical report [1]:

Lemma 5. *Consider any execution of* CERTBB *where all honest nodes invoke* CERTBB *with the same* committeeN*. If some honest committee member A in* committeeN *accepts h in some round t_A where $1 \leq t_A \leq 2d(m+1)$, then for every honest node B, there exists some t_B such that $1 \leq t_B \leq 2d(m+1)$ and B accepts h in round t_B.*

Lemma 6. *Consider any execution of* CERTBB *where all honest nodes invoke* CERTBB *with the same* committeeN*. If some honest node A that is not in* committeeN *accepts h in some round t_A where $1 \leq t_A \leq 2d(m+1)$, then for every honest committee member B in* committeeN*, there exists some t_B such that $1 \leq t_B \leq 2d(m+1)$ and B accepts h in round t_B.*

Lemma 7. [Final guarantees of CERTBB**]** *If all honest nodes invoke* CERTBB *with the same parameter* committeeN*, with* committeeN *containing at least one honest node, then:*

- *The return value from* CERTBB *must be the same on all honest nodes.*
- *If some honest node X invoked* CERTBB *(*committeeN*,* x_output*) with* x_output.block $\neq$ null *and with* x_output.b_cert *being a valid* b_cert *for* x_output.block*, then* hash(x_output.block) *must be the return value from* CERTBB *on all honest nodes.*
- *If the return value from* CERTBB *on an honest node is not* null*, then that honest node must have seen a valid* b_cert *for that return value.*

APPENDIX III: USING AGGREGATE SIGNATURES

As an optimization, FLINTBB uses aggregate signatures to reduce signature size. Aggregate signatures are used for p_cert, a_cert, and b_cert. In addition, OVERLAYBB [15] internally uses aggregate signatures. CERTBB, which is adapted from OVERLAYBB, similarly uses aggregate signatures. HotStuff [32] internally uses threshold signatures. For simplicity, FLINTBB replaces those with aggregate signatures as well. For example, one possible aggregate signature scheme would be the *MSP-pop* scheme [63] with BLS381. This aggregate signature scheme does not need trusted setup, and it allows each node to independently generate its public keys.[7] Each node can also add its own signature to an existing aggregate signature, without interacting with other nodes. Each aggregate signature in this scheme has 96 bytes. FLINTBB uses a bit vector to indicate the signers of an aggregate signature. For aggregate signatures in a_cert, b_cert, p_cert, and HotStuff, we need $|$committeeH$|$ bits in the vector. For aggregate signatures in OVERLAYBB and CERTBB, we need $|$committeeN$|$ bits.

APPENDIX IV: PROOF FOR THEOREM 4

Proof. We only prove the theorem for $c \leq \min(1-f_{\max}, \frac{1}{3})$, which trivially generalizes to larger c. With the given c and $f_{\max}$, we set $\beta = f_{\max} + 0.95c$ and $\alpha = \min(1-\beta, \frac{1}{3}) - c'$, with c' being any constant in $(0, 0.02c]$. Note that we must have $0 < \alpha < \min(1-\beta, \frac{1}{3}) \leq 0.5 \leq \beta < 1$.

We first claim that, except with negligible probability:

- If $f \in [0, f_{\max}]$, then every committeeN used in the (polynomial length) execution has some honest member.
- If $f \in [0, f_{\max}]$, then every committeeH used in the (polynomial length) execution has no more than β fraction of malicious members.
- If $f \in [0, \min(1-f_{\max}, \frac{1}{3}) - c]$, then every committeeH used in the (polynomial length) execution has no more than α fraction of malicious members.

We prove these three properties one by one:

- First, we adopt the simple approach from Snow White [3] to capture the effects of beacon bias, if any. Recall that the committee members for slot i are chosen by using

[6]Note that CERTBB only needs to disseminate the hash of the fast outputs from the certificate-broadcasters, which are of small size. OVERLAYBB instead disseminates the entire block from the block proposer. Because of this additional difference, part of the design in OVERLAYBB for dealing with (large) blocks is no longer needed in CERTBB.

[7]*MSP-pop* requires each node to have a proof-of-possession of the private key. Following [15], we require such a proof to be included as a transaction in the blockchain before the corresponding public key can be used.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

$\texttt{hash}(i|s_i)$ as the randomness. Here s_i is the beacon used for that slot, and the hash function is viewed as a random oracle. Since the length of the execution is $\text{poly}(\kappa)$, there can be at most $\text{poly}(\kappa)$ different beacon values that the adversary can try. We will be pessimistic, and will let the adversary freely choose which one of those $\text{poly}(\kappa)$ beacon values should be used for slot i. Doing so then fully captures the effect of possible beacon bias.[8]

Consider a given slot and a given beacon value among the $\text{poly}(\kappa)$ possible beacon values. If $f \leq f_{\max}$, then the probability that committeeN has no honest member is at most $(f_{\max})^{|\texttt{committeeN}|} \leq (f_{\max})^{\kappa} \leq 0.99^{\kappa} = exp(-\Omega(\kappa))$. There can be at most $\text{poly}(\kappa)$ possible beacon values and at most $\text{poly}(\kappa)$ slots. Since $\text{poly}(\kappa) \cdot \text{poly}(\kappa) \cdot exp(-\Omega(\kappa)) = exp(-\Omega(\kappa))$, a union bound immediately shows that except with negligible probability, regardless of which beacon is used to choose committeeN in each slot, committeeN always has at least one honest member.

- This property can be proved in a similar (but easier) way as the previous property. We omit the details for brevity.
- Consider a given slot and a given beacon value. If $f \in [0, \min(1 - f_{\max}, \frac{1}{3}) - c]$, then $E[f'] = f \leq \min(1 - f_{\max}, \frac{1}{3}) - c$. In turn, $\Pr[f' > \alpha] = \Pr[f' > \min(1 - \beta, \frac{1}{3}) - c'] \leq \Pr[f' > \min(1 - f_{\max} - 0.95c, \frac{1}{3}) - 0.02c] \leq \Pr[f' > (\min(1 - f_{\max}, \frac{1}{3}) - c) + 0.03c] =$ (by Chernoff bound) $exp(-\Omega(|\texttt{committeeH}|)) = exp(-\Omega(\kappa))$. Same as earlier, a union bound again shows that except with negligible probability, f' is always no larger than α.

Our remaining proof will condition upon the above three properties. We use a simple induction on the slot number to prove **Safety**, **Liveness**, and **Fast-confirmation**. Assume the theorem holds up to slot i. For slot $i+1$, all the conditions in Theorem 2 are satisfied. By Theorem 2's **Agreement** property, A_1 and A_2 must generate the same output from their respective invocations of FLINTBB for slot $i+1$. Hence **Safety** follows. By **Termination** in Theorem 2, after A_1 invokes FLINTBB for slot $i+1$, FLINTBB must eventually output. This output will then be the confirmed block in slot $i+1$. Furthermore, by **Validity** in Theorem 2, if the proposer is honest and if its block is received by every honest node by Line 12, then the confirmed block must be the proposer's block. Hence **Liveness** follows. Finally, if $f \in [0, \min(1 - f_{\max}, \frac{1}{3}) - c]$, then since committeeH has no more that α fraction of malicious members, all conditions in Theorem 3 are satisfied. By Theorem 3, A_1's invocation of FLINTBB must generate an output within the timeout value in Algorithm 1. Hence **Fast-confirmation** follows. □

[8]In practice, the beacon generation mechanism that we adopt from BCUBE [15] actually bounds possible beacon bias. Hence our analysis here is rather pessimistic.

APPENDIX V: CHOOSING PARAMETERS IN FLINT

FLINT has a number of internal parameters. This section explains how these parameters can be chosen in a systematic way (instead of as "magic numbers"). Our discussion uses $f_{\max} = 0.5$, 0.6, and 0.7 as examples.

Size of committeeN. FLINT uses two committees, committeeN and committeeH, for each slot. Here committeeN needs to have at least one honest member even when $f = f_{\max}$. FLINT selects the members of committeeN in the same way as BCUBE [15] (see Section IV). BCUBE also has the same requirement on committeeN, namely, containing at least one honest member. Hence we directly adopt the committee size from BCUBE [15]. Specifically, we use $|\texttt{committeeN}| = 45$, 55, and 80, respectively, for $f_{\max} = 0.5$, 0.6, and 0.7.

We refer the reader to [15] on the details of how BCUBE obtains these committee sizes, but the following provides some intuitions. If the committee members were all chosen uniformly randomly, then with the above committee sizes, the probability of the committee not having any honest member would be at most about 2^{-40}. But the committee members are actually chosen using beacons with potential bias. BCUBE has provided a detailed analysis showing that, with its beacon generation mechanism, the bias in the beacon roughly amplifies the error probability by at most about 2^{10}, which gives BCUBE a final error probability of about 2^{-30}. The exactly same analysis applies to FLINT.

Size of committeeH. Let f' be the fraction of malicious members in committeeH. We want f' to be no larger than β, even when $f = f_{\max}$. Here β is also a tunable parameter in FLINT. In practice, we want β to be as small as possible, so that α can be larger.

Hence we choose the size of committeeH and the value of β together. Specifically, we need to calculate the smallest β such that $\Pr[f' > \beta]$, which will be the error probability, is small. We aim for a final error probability of 2^{-40} here, so that this error probability is dominated by the earlier 2^{-30} error probability from committeeN. As explained earlier, the bias in the committee selection process may amplify the error probability by a factor of about 2^{10}, as compared to the ideal case where the committee members were chosen uniformly randomly. Hence we calculate the smallest β such that $\Pr[f' > \beta] < 2^{-50}$, assuming that the committee members are chosen uniformly randomly. Compensating for the bias will then give the desired final error probability of $2^{-50} \cdot 2^{10} = 2^{-40}$.

Figure 6 plots such β values, as a function of committee size and $f_{\max}$. There is a clear trade-off between β and committee size. To strike a balance, we simply choose 1000 as the committee size for committeeH, which leads to $\beta = 0.810$, 0.720, 0.626 respectively for $f_{\max} = 0.7$, 0.6, 0.5.

Thresholds for various certificates. We set the parameter α in FLINT to be $\alpha = \min(1-\beta, \frac{1}{3}) - 0.001$, so that α is smaller than $\min(1-\beta, \frac{1}{3})$. We set the thresholds for the various certs as functions of α and β, by taking the middle point in the

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.

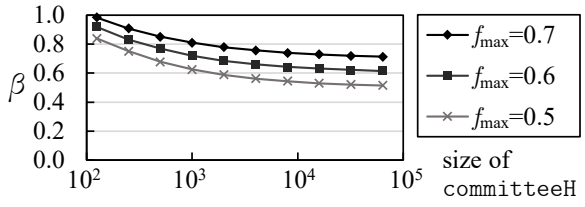


Fig. 6: The smallest β value such that $\Pr[f' > \beta] < 2^{-50}$.

respective possible ranges in Table II. For example, we set $t_b = \frac{\beta+(1-\alpha)}{2}$.

Invocation frequency of FLINTBB: Overview. Similar to BCUBE [15] and as explained in Section IV, FLINT invokes FLINTBB periodically, where each invocation does not necessarily wait for previous invocations to complete. A higher invocation frequency gives smaller inter-block time, but also requires more bandwidth.

To determine how frequently to invoke FLINTBB, we reserve sufficient bandwidth for each FLINTBB invocation, to ensure timely delivery of messages sent at Line 32, 44 and 45 in Algorithm 1 through 3. We make sure that such reservation will be sufficient even under $f = f_{\max}$. (Our reservation also fully takes into account the bandwidth needed for relaying those messages in the overlay, where applicable.) We focus on these *high-priority* messages, since Section VI has shown that for the security of FLINT to hold, we only need the timely delivery of these messages. The remaining *low-priority* messages in FLINT (e.g., messages sent by HotStuff) only serve to enable optimistic execution when f is small, and do not affect security. Those messages can be delivered with best-effort. When f is small, the high-priority messages will end up using only a small fraction of the bandwidth reserved for them. Such reserved, but unused, bandwidth can then be utilized by those low-priority messages, which in turn enables fast output. Such generation of fast output in FLINT will be demonstrated in our experiments later.

Invocation frequency of FLINTBB: Detailed calculation. We now provide the detailed calculation on how frequently FLINT should invoke FLINTBB. Recall that we want to reserve sufficient bandwidth for each FLINTBB invocation, to ensure timely delivery of messages sent at Line 32, 44 and 45 in Algorithm 1 through 3. Let x be the time between two successive invocations of FLINTBB. The following uses $f_{\max} = 0.7$ as an example to explain how we determine the bandwidth needs, which eventually leads to $x = 105$ seconds under our setting of 20Mbps available bandwidth on each node. (In comparison, BCUBE [15] uses $x = 98$ seconds under such a setting.)

We first consider the bandwidth needed at Line 45. Line 45 invokes CERTBB (i.e., Algorithm 4). In Algorithm 4, the only line that consumes bandwidth is Line 89. The message sent at Line 89 contains a hash (20 bytes), a `b_cert` (241 bytes), and an aggregate signature (106 bytes). The last part is 106 bytes, since the signature itself is 96 bytes, and we need another 80 bits to indicate which members in `committeeN` are signers. Each `b_cert` has a size of at most 241 bytes, which consists of a 20-byte hash, a 96-byte aggregate signature, and a 1000-bit array to indicate which members in `committeeH` are signers. Hence the message has total $20 + 241 + 106 = 367$ bytes. A node needs to send this messages to all its neighbors, and in our later experiments, each node has at most 42 neighbors. Finally, Algorithm 4 has a total $2d(m+1)$ rounds, where each round lasts δ time. Hence at any given point of time, there will be at most $\lceil 2d(m+1)\delta/x \rceil$ active invocations of Algorithm 4. Thus the total bandwidth needed for the message sent at Line 89 (and hence at Line 45) is at most $w_1 = \frac{367\times 42}{\delta} \cdot \lceil \frac{2d(m+1)\delta}{x} \rceil$.

We move on to Line 32. At Line 32, by our discussion in our technical report [1] on how nodes relay messages in the overlay network, each node at most sends/relays two `a_cert`'s. Each `a_cert` has the same size as a `b_cert`, which is 241 bytes. Let z be the `timeout` value in Algorithm 1. Line 32 will only be invoked between time $t_0 + 2\Delta$ and time $t_0 + z$, where t_0 is the time when Algorithm 2 starts. Hence at any given point of time, there are at most $\lceil (z - 2\Delta)/x \rceil$ active invocations of Algorithm 2 that may execute Line 32. By a similar reasoning as for Line 45, the total bandwidth needed for Line 32 is at most $w_2 = \frac{2\times 241\times 42}{\delta} \cdot \lceil \frac{z-2\Delta}{x} \rceil$.

Next for Line 44, we directly quote the results from [15]. Theorem 4 in [15] gives the total execution time x_1 of OVERLAYBB, which is at most 21120 seconds under our later experimental setting. Appendix II in [15] provides a formula for calculating the maximum number x_2 of bytes that a node need to send in a round, which is at most 124572 bytes under our later experimental setting. Hence the bandwidth needed at Line 44 is at most $w_3 = \frac{x_2}{\delta} \cdot \lceil \frac{x_1}{x} \rceil$.

Finally, we ensure that the sum $w_1+w_2+w_3$ does not exceed 90% of the available bandwidth on each node, to allow some leeway. Hence given available bandwidth $\mathbb{B}$ on each node, we find the smallest x such that $w_1 + w_2 + w_3 \leq 0.9\mathbb{B}$, and this will be the x value we use in FLINT. Plugging in $\mathbb{B} = 20$Mbps eventually gives $x = 105$ seconds.

Our experiments also consider $f_{\max} = 0.6$ and $f_{\max} = 0.5$. Following the above process, we use $x = 86$ and 79 seconds for those two cases, respectively. (In comparison, BCUBE [15] uses $x = 81$ and 74 seconds for $f_{\max} = 0.6$ and 0.5.)

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on September 01,2023 at 02:03:43 UTC from IEEE Xplore. Restrictions apply.