The Oasis Blockchain Platform

Oasis Protocol Project

June 23, 2020

1 Introduction

The range of applications where blockchain technology can be applied has been widening. Where blockchain v1.0 designs (e.g., Bitcoin [35]) provided consensus for serializing token transfers, blockchain v2.0 designs (e.g., Ethereum [46]) added support for generic computation via smart contracts. For next generation blockchains, one of the main goals is to enable *private* computation.

The Oasis Blockchain Platform is a Layer 1 Proof of Stake (PoS) [30] smart contract platform that provides scalability, extensibility, and privacy. The main features of the platform enable efficient *verifiable* and *confidential* smart contract execution. The modular design of the platform allows the consensus layer to be easily changed to use a completely different consensus mechanism to benefit from the latest progress in the space; furthermore, the consensus layer accommodates multiple smart contract runtimes simultaneously. These independent parallel runtimes, or "ParaTimes", can use different verifiable computing and confidential computing techniques with little or no changes to the interfaces. Section 3 defines these components more precisely.

Before the consensus layer accepts an update submitted by a ParaTime, it first verifies that the update value, which represents a state change for the source ParaTime, is correct. This is done using a form of *verifiable computation*; currently we use replicated computation in the ParaTime for verifying smart contract transaction execution results.

Our verifiable computing implementation is called "discrepancy detection" [48], incorporating an efficient fast-path optimization that allows us to use smaller ParaTime execution committees of size $F + 1$ most of the time, where F is the number of faulty or Byzantine nodes. This compares well with traditional Byzantine Fault Tolerance (BFT) based replicated computation techniques that require committees of size $3F + 1$, or Ethereum 1.0 which uses a $2F + 1$ honest-majority design. This greatly improves scalability for smart contracts that run on the Oasis Blockchain Platform, since the computing cost for a given desired level of security is reduced. Our modular layered design makes this possible because it separates smart contract execution from consensus, and this separation allows the Oasis Blockchain Platform to reduce the replication factor for smart contract execution without sacrificing security, separate from the validator replication in the consensus committee. Note that our architecture does not require the use of discrepancy detection and ParaTime implementers can choose to use other verifiable computing techniques for smart contract execution. ParaTimes can be implemented, registered, and operated by anyone, and they can commit arbitrary values, e.g., something as simple as document hashes for timestamping [26], rather than ParaTime state updates.

ParaTimes may implement other confidential computing techniques such as multiparty computation (MPC) [13, 7], fully homomorphic encryption (FHE) [22], zero-knowledge proofs (ZKP) [6, 27], etc. Our goal is to enable a range of privacy enhancing technologies so that different ParaTimes can choose different technologies and make different trade-offs.

The Oasis Blockchain Platform was designed and implemented in conjunction with a reference ParaTime implementation by Oasis Labs. Having a working ParaTime allowed us to exercise the system interfaces, so that the module boundaries were not designed *in vacuo*. This ParaTime provides private computation using Trusted Execution Environments (TEEs) [32] for efficient, cost-effective confidential smart contract execution; it also provides verifiable computing via discrepancy detection. Like for verifiability, our architecture does not require ParaTimes to provide confidentiality, nor does it require (or pre-

The source for this paper is maintained at `git@github.com:oasislabs/paper-mainnet-light-overleaf.git` and this version was built from `NOT CHECKED-IN`.

clude) the use of any particular confidential computing technique. Indeed, a ParaTime that does not use any confidential computing techniques can easily coexist with a TEE-based ParaTime and provide only a guarantee of verifiable execution.

The modular nature of the architecture also allows the consensus layer implementation to be updated independently of the ParaTimes and vice versa. New ParaTimes can be easily incorporated without any impact on existing ParaTimes. For example, a private closed smart contract system for use within an enterprise environment can be realized as an independent ParaTime that operates in parallel with a public, open smart contract ParaTime.

This paper describes our architecture, the consensus layer design, and describes how a reference implementation of a confidential ParaTime fits into the overall architecture. It describes the interfaces as currently designed/implemented—they could change as the system further evolves. This paper does not describe any particular blockchain network built using the Oasis Blockchain Platform. In particular, while this paper suggests some potential areas in which the platform might be enhanced or extended in the future, the community governance of such blockchain networks and how future engineering changes will be considered are beyond of scope of this paper.

Next, in Section 2, we discuss the principles used in our system design. Section 3 describes our system architecture in more detail. Following this, Section 4 gives a high-level overview on how our discrepancy detection technique works and discusses the key ideas behind the approach. Section 5 gives details about our ParaTime design and in particular discusses the reference TEE-based private computation runtime that has been built. Next, Section 6 discusses how key managers fit in with confidential ParaTimes, and presents some potential future directions for more featureful key managers. Finally, Section 7 gives some concluding remarks.

We include appendices that provide more detailed analysis, which might be useful for readers interested in the security arguments used in the design. Appendix A explains discrepancy detection technique in more detail, compares its probabilistic security model with that of standard BFT committee selection techniques, and relates its security to resources that must be spent by the adversary. Appendix B discusses the security of discrepancy detection under the stronger assumption that the adversary is non-adaptive, so must risk losing some stake as penalty for having the compute nodes that they control return a detected discrepant answer. Following this, Appendix C discusses the need to check that security checking operations actually took place.

2 Design Goals and Principles

2.1 Goals

We want to make the Oasis Blockchain Platform flexible, extensible, scalable, secure, and better fault-isolated. Here is what we mean:

- flexible. It should be easy to modify system parameters to accommodate different deployment scenarios (e.g., constraints on ParaTime nodes).
- extensible. The system should be reasonably easy to extend by adding new components to implement different functionality/semantics (e.g., different ParaTimes or different confidential computing techniques).
- scalable. The total transaction throughput of the system should increase with the total number of nodes, ideally in a linear fashion, even for complex transactions. Scalability is a form of efficiency at scale, since it allows the system to grow without incurring huge expenses and driving up the per-transaction costs.
- secure. The system should be able to enforce security policies that are impossible for earlier blockchain systems, allowing a greater breadth of applications. In particular, in addition to the supporting verifiable computations, the system also supports the ability to perform those computations in *private*, using confidential state / inputs. Furthermore, our system architecture and implementation should make design choices in favor of *simplicity* when feasible; this makes careful design reviews and implementation audits more practical and effective.
- fault-isolated. Fault isolation is a basic component of fault tolerance. We want our system to be fault tolerant both in terms of *security fault isolation*, so that a security vulnerability in one component should not compromise the security of another component, and in terms of *performance fault isolation*, so that performance problems in one component does not unduly affect the performance in the rest of the system.

The combination of flexibility and extensibility allows the development of additional ParaTimes with different security, performance, or usability trade-offs. In particular, the support for confidential smart contracts and improved isolation properties provides both developers and users more choices in the types of applications that can be built using the Oasis Blockchain Platform.

2.2 Principles

We wanted to follow the principle of deriving the simplest and most flexible design possible, while still meeting our goals. The functional decomposition should expose simple, orthogonal inter-component interfaces.

Expanding these ideas in our system design resulted in the following:

- Modular Architecture: We have designed and implemented a clear separation between the consensus layer and an independent ParaTime smart contract execution layer.
- Simple Consensus Layer: The consensus layer is kept as simple as possible. The consensus component only handles validator committee management operations such as token balance transfer, staking, delegation, debonding, etc; ParaTime committee scheduling; and commits of values from ParaTimes. This helps increase security and efficiency, since (complex) smart contract executions are isolated from consensus operations, comparing well with Layer 2 solutions.
- Independent ParaTimes: Independent ParaTimes offer flexibility, since it is easy to incorporate different runtimes for different needs, e.g.:
  - Security requirements. Different ParaTimes can be optimized for confidential smart contracts versus integrity-only smart contracts.
  - Performance requirements. The reference ParaTime implementation offers no explicit concurrency for smart contract execution. However, multiple instances can be run simultaneously, achieving simple concurrency via explicit separate-world parallelism. Other ParaTimes developed later by the community can offer greater parallelism using more advanced techniques.
  - Non-technical / external requirements. For example, enterprises may require that their private ParaTimes execute only on a specific designated set of server nodes, so that they run on-premise or within particular legal jurisdictions.

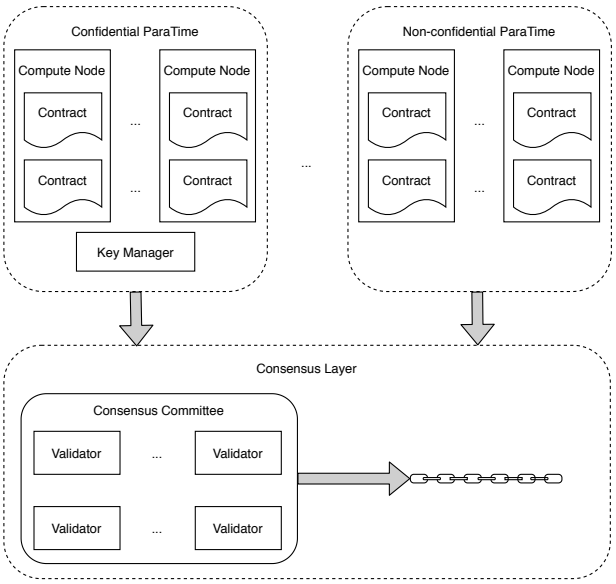

Figure 1: Oasis Blockchain Platform Architecture

If a ParaTime is unable to handle its transaction load or fails for whatever reason, the impact on the rest of system is simply that no state updates will be submitted from that ParaTime to be committed to the consensus layer. Other ParaTimes are not affected. A ParaTime cannot submit too many values or otherwise spam the consensus layer, since each ParaTime must pay consensus layer transaction fees. This provides us with encapsulation and fault isolation: ParaTimes are distinct entities and cannot affect nor interfere with each other.

The explicated partitioning between components provides a clear verifiable computing interface. This consensus-ParaTime interface is extensible, though obviously adding interfaces (especially if multiple extant interfaces can interact/interfere with each other) can increase complexity and auditing effort. The reference ParaTime implementation uses discrepancy detection for verifiable computing (see Section 4), with the detection occurring at the consensus node's ParaTime interface. The use of discrepancy detection enables a lower replication factor in common fast-path execution, making the verifiable computing implementation much more efficient and cost effective. Any Para-

Time developer can easily extend this implementation and add other verifiable computing mechanisms such as Arbitrum [28] or ZKP-based transaction verification [27].

3 Architecture

We describe and give more details on the major components in this section. The Oasis Blockchain Platform contains these major components that interact at well-defined interfaces.

3.1 Architecture Overview

The main components are in Figure 1:

- Consensus layer. The consensus layer accepts values from its clients (the ParaTimes) and writes these values into the next block in our blockchain. The consensus layer also includes the necessary machinery for validator and ParaTime committee selection, and for handling the consensus-layer or "native" token operations, which are needed for its own management. The meaning of the values generated by a ParaTime is entirely defined by that ParaTime, so a ParaTime could be built that only supports Bitcoin-like transactions or even document timestamping [26]. Note that here we are mainly concerned with ParaTimes that support smart contract execution and will not discuss these other uses below.
- ParaTimes. ParaTimes are where smart contract execution occurs. Each ParaTime defines its own smart contract execution environment and the semantics for the interfaces it exposes, and chooses the mechanism and associated parameters (e.g., the degree of server replication) used to verify results of contract execution. All ParaTimes will support some notion of event sequencing—typically transactions—since ParaTimes commit values to the consensus layer. While they are possible, we do not consider non-transactional, non-verifiable computing ParaTime designs below. A ParaTime may include:
  - Key managers. Key managers are only needed for TEE-based confidential ParaTimes, since the TEE-based ParaTime compute nodes store encrypted contract state; key managers are not needed for FHE-based confidential ParaTimes or non-confidential ParaTimes.
  - Committee selection pool criteria, e.g., when the ParaTime uses a rotating set of compute nodes for replicated computation (see Section 4), what requirements are there for nodes to qualify beyond simple stake thresholds?

The consensus layer provides only a specific set of consensus-backed services and no direct support for confidential computing. These services are (1) native token operations such as transfer, staking, delegation, etc; (2) validator committee selection; (3) ParaTime committee selection; (4) ParaTime results verification; and (5) commitment of verified ParaTime results to a blockchain. In particular, this means that native token account balances and transfers are not confidential. The native token is used for staking and commitment processing fees.

Verifiable computing is implemented at the consensus node's ParaTime interface. The only verification mechanism that is currently implemented is discrepancy detection. It is generic in that multiple ParaTimes can use it, with a separate instance of the mechanism for each ParaTime, with ParaTime-specific interface endpoints. The replicated ParaTime nodes are where smart contract execution occurs; the value delivered to the consensus-ParaTime interface is the Merkle hash of the new ParaTime state that resulted from contract execution. The discrepancy detector can either accept a result to pass onto the blockchain code for consensus processing, or reject results as unverified, signaling the ParaTime layer to perform exception, "slow-path" processing. See Section 4 for details.

Confidentiality for smart contract execution is implemented solely within ParaTimes. For ParaTimes that use techniques such as FHE, an observer of the contract state cannot determine what it is doing unless the observer has the cryptographic keys that encrypted the input values for the FHE contract. For ParaTimes that use TEE-based computation, contract execution occurs within the TEE environment, and contract state is encrypted before being stored. Because of this, both the ParaTime compute nodes and the key manager are jointly responsible for contract state confidentiality, and mutual authentication via TEE-based remote attestation is critical. Regardless of the confidentiality technique used, the value passed to the consensus layer is a Merkle hash of the encrypted ParaTime state, and thus leaks no information as long as the state encryption leaks no information.

The reference confidential ParaTime implementation uses the TEE-based approach to achieve reasonable performance. Smart contract developers can choose among the available ParaTimes based on which of them satisfies their application's security needs, and if none suffices, they can propose their own ParaTime as a ParaTime developer.

Note that ERC-20 [12] style smart contracts running within ParaTimes can implement their own tokens independent of the token used at the consensus layer. Furthermore, such a smart contract could be developed to run on a confidential ParaTime; this would result in a token with confidential token account balances, transfers, etc.

3.2 Consensus Layer

The consensus layer is a simple BFT / Proof of Stake blockchain responsible for consensus. It is based on the Tendermint [10] BFT consensus protocol.

In addition to blockchain management transactions, each block in the blockchain contains a limited number of ParaTime value update entries. The contents of these entries are the focus of the consensus layer: the clients of the consensus layer—the ParaTimes—submit contents for these entries, and the production of blocks extending the blockchain determines the global serialized execution order for the operations represented by these values.

Note that the main abstract service provided is simply a consensus mechanism for the values being committed. It is possible to replace it with another consensus mechanism, e.g., Libra HotStuff [5], instead of Tendermint, with very little impact on the rest of the system.

The consensus layer handles not only commitment of ParaTime values and consensus committee management, but also provides services for ParaTimes:

- native token. The native token is used for staking both consensus and ParaTime committees, and for payment for consensus operations.

- committee selection and scheduling. The consensus layer keeps track of virtual time epochs and handles the committee selection process. .

- verification. The consensus-ParaTime interface is where the ParaTime-specific verifiable computing protocol terminates.

3.3 Consensus–ParaTime Interface

Abstractly, the consensus-ParaTime interface is simple: the replicated consensus nodes each accepts a submitted value from the ParaTime layer that has undergone verifiable computing checks, and the consensus algorithm decides which of the submitted value(s) to commit to the blockchain.

Of course, if there are N validator nodes and the ParaTime is also replicated, say using a compute committee of C nodes, this could mean that we would send a total $N \cdot C$ messages, each containing a signed value, delivered via point-to-point communications; or we could use a gossip network [43] to pass C signed messages to the N validators. Alternatively, we could require the C ParaTime nodes gather signatures until, say, $\frac{2}{3}$ of the ParaTime nodes have signed a single value, and then to forward that value to all the N validator nodes. The details of how the verification mechanism decides on which value to use at each of the N validators is a detail that belongs within the verification abstraction.

One way to think of this is to include a verification mechanism defined process that is co-located with each validator process representing the ParaTime layer as the consensus client. The verification-defined process handles verification-specific communications and submits a single value to the validator process with which it is co-located; thereafter, it is the job of the validator process to reach consensus with its peers in the validator committee about which value to accept onto the blockchain.

Rather than using a separate process, this is implemented as a code module used by validators that receives messages from senders within the ParaTime—thus, at the network-communication-level, a verification-defined messaging interface—and the consensus layer exposes a stable code interface that is used with this library. From an architectural viewpoint, combining this into a single process enables better control over communications and scheduling. From a deployment viewpoint, adding new ParaTimes will involve consensus-layer managed configuration changes for their registration, and—especially if a new verification mechanism is to be used—minor code changes that the network operators must accept.

We envision that if/when a new ParaTime needs to use an alternative verifiable computing scheme, additional libraries (external interfaces) will be added here.

3.4 ParaTime Layer

The ParaTimes are where smart contract execution occur. The decoupling of execution from consensus allows each component to be optimized for their respective tasks. In particular, when a ParaTime uses TEEs for confidentiality, the TEEs also provide protection for the computation's integrity which can translate to requiring a lower replication factor for smart contract execution than compared with non-TEE ParaTimes.

It is likely that we will have at least one key manager instance per TEE type and/or per ParaTime. For ParaTimes that use TEE-based computation a key manager is needed, because contract execution occurs within the TEE environment, and contract state is encrypted before being stored. For ParaTimes based on homomorphic encryption (FHE) no key manager is needed, because such ParaTime nodes operate only on encrypted data, and only the entities that applied the homomorphic encryption to the inputs could decrypt the results from the ParaTime computation outputs. Similarly, an verification-only ParaTime would not need to have a key manager.

The reference ParaTime implementation handles *confidential* smart contracts, the details of which will be described in another paper [39]. The Oasis community can implement additional new ParaTimes in the future, and/or extend the reference ParaTime implementation to provide new features. In this ParaTime, we rely on the TEE security assumptions and thus include a key manager.

3.5 Key Manager

Key managers are particular to ParaTimes, though multiple ParaTimes can choose to use the same key manager code. The key manager is responsible for maintaining control over the cryptographic keys used to protect confidential contract state.

The properties that key managers should provide are:

- Confidentiality. The confidential smart contract state must remain confidential. Generally, this means that only authorized, attested ParaTime compute nodes can access the keys, and implies that proper cryptographic protection is used for communications between the key managers and the ParaTime compute nodes.
- Availability. The keys are needed to execute confidential contracts, and the key managers must provide enough availability (through replication to geographically separated nodes in different fault zones) for the ParaTime. This also implies the integrity of the key store as well as communication security, since destruction of the key store or getting the compute nodes to process contract invocations with a bad key would severely damage the contract state.

In Section 6 we will discuss the design choices made in the key manager for the confidential ParaTime reference implementation in more detail.

4 Discrepancy Detection

Discrepancy detection is the verifiable computing technique that the Oasis Blockchain Platform uses to verify ParaTime execution currently. It permits the use of smaller ParaTime committees than other schemes, making the system more efficient, especially when the application is compute heavy compared to consensus.

The key ideas in discrepancy detection are (1) random selection of compute nodes from a population to form a compute committee and (2) accepting the results only if *all* committee members agree. A separate protocol, which we call "discrepancy resolution" is used when a discrepancy is detected. One can think of discrepancy resolution as another security parameter to discrepancy detection, and that the resolution protocol is the expensive, "slow path" mechanism used to correct faults, and the detection protocol is the cheap "fast path" mechanism used to detect faults.

Here is what discrepancy detection does:

- Results of compute node execution is signed by the node and sent to the discrepancy detectors via a gossip network. The detectors are implemented within verification code colocated within the validator nodes.
- The verification code checks the results. Each compute node should have signed only one result—double signing results in slashing. Absent double signing, the results should be identical, i.e., discrepancy free.
  - If the results are discrepancy free, they are submitted to the validators for consensus protocol processing and block creation.
  - If there is a discrepancy, the results are marked as disputed and the resolution phase is started to use the slow-path protocol to determine the correct results to use. The

nodes with disputed results that differ are penalized an amount that pays for the slow-path re-execution costs.

An important observation is that except for unusual events like hardware failures, ParaTime errors that result in non-deterministic execution, or an non-adaptive adversary (see Appendix B), the resolution mechanism should never be invoked.

The analysis showing that the security properties is based on calculating the probability of selecting an all-Byzantine committee, so that discrepancy detection would fail to detect any errors. The discrepancy detection security parameters are chosen to drive this probability as low as needed in order to drive the adversary cost to a level that should be unacceptable to the attacker.

The key result of this analysis is that the committee size needed is significantly smaller than that needed—for the same level of security—as would be for a conventional BFT scheme. This means that by using discrepancy detection, the cost in system resources needed to deliver the same level of security is far smaller, and that our system will be able to scale better.

Those who are interested in the analysis of the security of discrepancy detection should refer to Appendix A.

5 Parallel Runtimes

The ParaTime interface to the consensus layer enables the Oasis Blockchain Platform to run different ParaTimes. Indeed, multiple ParaTimes can run simultaneously, subject to the availability of consensus layer resources for processing updates. Parallel execution via ParaTimes can be an effective way of scaling transaction throughput when no inter-ParaTime coordination is needed.

ParaTimes can commit every transaction to the consensus layer or only commit periodically values that represent several transactions. In the latter case, summarizing multiple transactions into a single hash value means that ParaTime transaction throughput can be decoupled from the block production at the consensus layer. When a ParaTime hash represents many transactions, those that occur within a ParaTime time-line may be assigned a sequential order by the ParaTime, i.e., based on the authenticated data structure used to derive the summary hash value. However, a transaction T_A that ran on ParaTime A and a transaction T_B that ran on ParaTime B do *not* necessarily have any ordering relationship with each other, e.g., if the hash summary from each of the ParaTimes for these two transactions were committed to the same consensus block. Both T_A and T_B occurred *before* the block was added to the blockchain, but we cannot make any inferences about their relative order.

When two or more ParaTimes share the consensus layer, they can use message passing techniques for contracts hosted on one ParaTime to communicate with the other, e.g., Inter-Blockchain Communications protocol (IBC) [18] could be applied. Note that this adds complexity, since the ACID transaction properties provided by smart contract execution are provided at the level of transactions, and message passing schemes require transaction commitment to send messages—communicating contracts that want transactional properties will have to implement their own commitment/rollback mechanisms. It also remains to be seen whether applications will find the inter-chain communication overhead acceptable.

5.1 Private Computation Runtime

The reference ParaTime uses TEEs to support efficient execution of confidential smart contracts. This section discusses at a high level some of the security issues which are solved by the architectural design, as well as some general issues that remain to be addressed. These are issues that must be addressed by all TEE-based confidential ParaTimes, so they should be of independent interest.

5.1.1 TEE Security Model

In the TEE model, code can be placed in a TEE-provided secure execution environment, an *enclave*, such that an adversary cannot tamper with its execution after initialization. Further, enclaves running on TEE-enabled hosts can prove their identities—their initial state—to external observers through *remote attestation*, which involves the use of an external service similar to how certification authorities are used in many public-key cryptosystems. In the case of the Intel SGX realization [34], this is the Intel Attestation Service (IAS) [4]. In the future, support for Data Center Attestation Primitives (DCAP) [40] could be added to support remote attestation without the use of Intel's IAS servers.

The threats that the abstract TEE model are robust against include basically everything outside the CPU chip packaging boundary. Adversaries are allowed

to have control of the operating system (and hypervisor, if used)—even the boot ROMs—and could read and write the contents of DRAM at will while the CPU is executing. The adversaries are not, however, allowed access to the CPU's internal state: register values, architectural or otherwise; branch prediction table contents; on-chip cache contents; etc.

5.1.2 TEE Vulnerabilities

The TEE abstraction and TEE implementations are not (yet) congruent. In particular, we recognize the need for extensive security reviews in realizations of the TEE abstraction: Intel's SGX [34] has been shown to be vulnerable to a variety of attacks [9, 44, 19, 42, 16] taking advantage of implementation flaws, and it may take a few more generations / revisions before the security community is willing to believe that all reasonable "low-hanging fruit" flaws have been found; AMD's SEV [29] architecture has not been subject to a comparable level of review, though its implementations too have had vulnerabilities [8]; and the Keystone [32] project's open-source hardware design has yet to result in any production chips usable for evaluation.

When a service is implemented on a TEE-enabled node, it must be internally split into an enclave-hosted component and a non-enclave, external component. This is because the enclave component has no persistent storage,[1] and I/O is, by necessity, through the external component. This is true for SEV as well as SGX, since I/O hardware cannot be implicitly trusted. TEEs provide per-enclave cryptographic keys that can be used to ensure confidentiality and authenticity of data persisted outside the enclave; this allows the enclave components to be restarted and continue from an earlier checkpoint. The lack of *trusted* persistence, however, implies that an enclave component is subject to a state rollback attack: the external environment can replay old checkpoints, possibly over and over again across multiple restarts, and the enclave component will be unable to detect this.

The TEE state rollback problem has various implications for the applications of TEEs for confidential computing, which we will discuss next.

[1] SGX monotonic counters rely on Management Engine (ME) flash. Security researchers have found ME vulnerabilities and advise that ME should be disabled where possible. Furthermore, the monotonic counter values are chip specific, and contract state should not be tied to a particular CPU that might fail.

5.1.3 State Rollback

While cryptographic protocols can ensure that the other end of a communication endpoint is live, this only increases the scope of the number of components that an adversary would have to rollback—if this number is small, then the adversary can take advantage of this problem.

The simplest way to avoid state rollback attacks is to be stateless.

In our reference ParaTime, compute nodes do not checkpoint state, and they use external untrusted storage servers for contract state. This ParaTime's smart contract model views contract calls as authorizing state changes from the current consensus contract state. Contract execution is always given not just the call parameters and entry point, but also includes the current consensus contract state via its Merkle tree hash; the output of a contract call also includes the resultant contract state hash.

While message replays could cause re-execution, we ensure that re-execution is either idempotent or has no useful side-effects:

1. Smart contract execution is deterministic and idempotent. Confidential contract state is garbage collected, so replay of contract execution should not result in a new state being created unless it had been created but has already been garbage collected. There is a potential for increased load at both storage servers as well as compute servers, which is a denial of service concern that can be addressed separately.

2. Smart contract call results and generated events should not be accessible unless the transaction commits: the values should be encrypted and subject to atomic delivery [17]. For each transaction, the compute nodes monitor the consensus layer post-execution to verify that the transaction is committed before revealing the corresponding decryption key to decrypt for its results and events. Since our Byzantine fault tolerance parameters include power loss and other non-Byzantine faults, we would be guaranteed that some honest compute node will be available to reveal the results key.

   Atomic delivery is currently not fully supported in the reference ParaTime. Supporting this fully will enable deployers of confidential computing ParaTimes to relax limits on where a confidential ParaTime compute node may run.

5.1.4 Side-Channel Attack Mitigation

There are two main kinds of side channels that can cause information leakage when a TEE-based ParaTime is used. The first is persistent smart contract storage access patterns and execution timing, and the second is microarchitectural side channels in TEE realizations.

For the former, we believe that confidential smart contracts which include the memory access side channel in their threat model should use ORAM-style techniques [41] to obscure their memory access patterns. Since ORAM techniques thus far are still quite expensive, the reference confidential ParaTime does not include built-in ORAM support. It should be relatively simple, however, for a developer to modify the reference ParaTime to use ORAM techniques or to implement their own ParaTime that might better integrate such techniques. In general, we recommend the contract code to implement application-level ORAM or other data-oblivious techniques directly, rather than relying on ParaTime-level support; in this case the reference ParaTime implementation would suffice and no additional support from a ParaTime would be needed.

Similar to memory access patterns, the time taken to execute a smart contract call can leak information about the confidential contract's state and the contract call parameters. We recommend that confidential smart contracts which include this side channel in their threat model to use constant time or side-channel bandwidth limiting algorithms.

For the latter, there is no purely architectural way to limit confidential ParaTimes to running on hosts that adversaries could not access to mount side-channel attacks. For now, deployers of the reference confidential ParaTime can restrict its execution to specific operators who are trusted not to exploit possible side channels due to concerns about SGX side-channel vulnerabilities.

In the future, the community may consider establishing auditing procedures by which a wider base of SGX-enabled hosts can be utilized, e.g., verifying that controls are in place so that the SGX-based ParaTimes will never be hosted in a shared environment where side-channel attacks might be mounted, or verifying that personnel security controls are in place so that those who are authorized to access SGX nodes are vetted and will be unlikely to mount side-channel attacks using their authorized accesses. Of course, when Intel eventually releases a version of SGX or another TEE instantiation is identified that has sufficient side-channel information leak mitigation built in, it may be possible to safely allow audit-free participation, either by opening up participation in deployments of the reference ParaTime, or by running new open and decentralized ParaTimes.

5.2 Non-confidential Computation Runtime

Non-confidential runtimes provide similar guarantees to those provided by Ethereum in that computation is verifiable, but there is no encryption of smart contract state.

A non-confidential ParaTime still enjoys state integrity due to the use of discrepancy detection and serialization to the consensus blockchain, and it can use the same consensus APIs as confidential ParaTimes.

5.3 Smart Contract APIs

ParaTimes provide their own interfaces to the smart contracts they host. Different ParaTimes can have their own semantics, but they will typically include some notion of transaction aborting or committing as well as some way to access longer-term storage (with transactional semantics).

The consensus layer is concerned only with consensus, which is used to order the transactions processed by ParaTimes. ParaTimes are free to provide bindings to smart contract APIs for arbitrary languages, though the reference ParaTime is initially focused on supporting smart contracts written in Rust and Solidity.

6 Key Managers

In the ParaTime model, each confidential ParaTime can use its own key manager. A key manager does not have to be transaction-aware—it is responsible only for ensuring that keys needed for a particular smart contract can be made available to the TEEs responsible for executing that smart contract. However, as we will see below, having the key manager be transaction-aware can yield some security advantages.

6.1 ParaTime Key Manager

The design of the reference ParaTime key manager favors simplicity, reliability, auditability, and ease of implementation over features, even those focused on security. Because key managers are matched with Para-

Times, a ParaTime developer can decide to use our reference implementation or to build their own key manager as needed.

The design is similar in some ways to Google's cloud key management system [24], though it takes advantage of TEE features that were unavailable at the time of that system's (initial) design. The fundamental idea is to use key splitting from a master key to derive additional keys. In the ParaTime key manager, these are per-contract state storage encryption keys.

6.1.1 Application Splitting

The reference ParaTime key manager is internally split into enclave and enclave-external components. The enclave component is trusted to do key management, but it is incapable of being a blockchain client since I/O is handled by the external component and there is no persistent TEE-side state (rollback protection). The external component is trusted not to mount side-channel attacks on the enclave component and not to perform rollback attacks on the enclave component. It is *not* trusted to do key management; all key material is persisted outside of the enclave component using the TEE-provided enclave encryption key. Since the TEE-provided enclave encryption key is only accessible from within the TEE, the external component does not have direct access to the key material.

While the split trust model for the key manager may seem complex, it is actually well justified:

- The external component is operated by a trusted entity, but we do not want to assume that no operation errors will occur. For example, to decommission a machine (such as after a hardware failure), any persistent storage that may have held unencrypted key material must be carefully destroyed. For magnetic media, this can mean overwriting the disk blocks enough times so that remnant magnetic domains [23] are undetectable [25], or very powerful whole-disk degaussers be used to ensure that all remnant magnetic fields are undetectable. For flash-based storage such as SSDs, this essentially means using built-in whole-drive erasure mechanisms, since wear-leveling algorithms implemented within SSDs make techniques such as overwriting previously written files fail terribly; unfortunately, the efficacy of built-in sanitize commands vary [45].

- The enclave component relies on the external component for all of its I/O. We *have* to trust the external component to persist key material wrapped by the TEE hardware-protected/-provided enclave key.[2] There is no way to ensure that key material is properly updated or erased, even via probabilistic mechanisms such as in Vanish [21], if we do not trust the TEE-external component to handle persistence of encrypted key material: instead of only distributing out key fragments to a global-scale Distributed Hash Table (DHT), the TEE-external component can also record the fragments, invalidating the assumption that stored data will be lost due to DHT node churn or cleansing of timed-out data. Being able to update key material in the key manager is a necessary requirement for implementing contract state re-encryption for backward secrecy (see Section 6.2.3 below).

6.1.2 Replication

For fault tolerance, ParaTime key managers can be replicated. All of the security critical parts of the key manager run in an attested SGX enclave. This TEE-hosted component relies on the TEE-external component for all communications, but using SGX remote attestation, it can authenticate the identity of key manager replicas with which it communicates, so key managers can selectively share the master key only with other instances of the same enclave. The key manager also implements access controls to only allow authorized key manager instances to become an active replica. This prevents adversaries from mounting side-channel attacks against the TEE-hosted component of the key manager on their own machines.

6.2 Potential Future Key Manager Features

We have considered additional features that could be incorporated into the current ParaTime key manager or in key managers for other ParaTimes. Some of these should be practical to implement—should the engineering complexity vs security trade-off be favorable—while others will require additional research. These are only initial ideas for the community to review, discuss, and decide on whether these or other features should be developed and adopted, based on data from the

[2] We could rely on being able to reach replicas when an instance restarts. However, the TEE must have access to some authentication secret for replica access control, which would in turn have to be persisted using the external component.

reference ParaTime deployment and feedback from users.

6.2.1 Access Controls and Rate Limiting

In addition to access controls that restrict where key manager replicas can run, further access control mechanisms could offer defense-in-depth. In particular, as we mentioned in subsubsection 5.1.2, the possibility of information leaks from nodes executing confidential ParaTimes is a concern.

To partially mitigate this risk, further access controls could be added to the ParaTime key manager, so that state encryption keys for a given confidential smart contract are released only to compute nodes scheduled to run that contract, and even then, rate limited to make restarting nodes to gain statistical information from side channels less useful. Note that this requires the key manager to be transaction-aware, since notions of access control / rate limiting are determined by information from the ParaTime's transaction scheduler.

6.2.2 Proactive Secret Sharing

In order to mitigate the effects of a potential key manager compromise, we could split secret keys using secret-sharing techniques [33] so that a compromise of a sub-threshold number of key managers will not compromise the keys.

The load on the key managers is primarily generated during smart contract call set up, since contract execution cannot proceed until state can be accessed. While the load on ParaTime confidential compute nodes will vary depending on how smart contract code is written, it is likely to be higher than that of the key managers. Especially if/when platform developers decide to put in the engineering effort to take advantage of available contract parallelism, we will need more ParaTime confidential compute nodes than key managers, and it seems likely that we will be able to decentralize the control of ParaTime confidential compute nodes before we can the key managers.

Since the number of key manager replicas needed should be smaller than the number of confidential ParaTime compute nodes, we believe the risk of a confidentiality compromise, whether through a side-channel attack or not, is likely to be higher via ParaTime confidential compute nodes than via key managers.

6.2.3 Smart Contract State Re-encryption

Additionally, we have considered periodic rekeying of confidential smart contract storage to provide *backward secrecy.*That is, a compromise of the contract state encryption key used at time t will compromise the contract's state at that time, but after a later rekeying operation, the adversary would not be able to determine the result of new contract executions.[3]

While it depends on the length of the rekeying period and policies on how much of a contract's confidential storage should be encrypted, it is likely to be expensive. If re-encryption is performed by a special service separate from the standard ParaTime, then it will be clear to network-based adversaries that the contract state did not change from before the re-encryption and after, so unless a contract call is later performed which modifies the value associated with the key, an adversary who has access to the old key (but not the new) and is able to do traffic analysis will continue to know the value stored at such locations. If re-encryption is done as part of normal contract execution, the amount of state to be re-encrypted (which can be a security parameter) would have an impact on overall throughput, since all storage write operations must complete before the transaction can commit.[4] It would be far simpler to require confidential contracts that are concerned about this to use ORAM [41] techniques.

7 Conclusion

We have presented an overview of the Oasis Blockchain Platform's modular architecture. The main design goals are flexibility, extensibility, scalability, security and privacy, and improved isolation: flexibility and extensibility for allowing incremental engineering investments and migration paths toward longer-term optimizations and additional scalability improvements; scalability to be more efficient / cost-effective and to be able to address higher workloads; security and privacy for verifiable and confidential computing; and improved isolation for better fault

[3] Backward secrecy is a concept more typically used in the context of encryption protocols, where a key compromise of one or both of the communicating parties in the present does not leak information about messages encrypted and exchanged in the future. This contrasts with protocols that provides *forward secrecy*, where a compromise in the present does not compromise the secrecy of messages sent in the past.

[4] While it may seem feasible to separate the re-encryptions that do not change the data value from actual writes, this again would allow traffic analysis to learn what values did not change.

tolerance. The simplicity and economy of design also aids in engineering feasibility, making in-depth design reviews and code audits more practical and contributing to the system's robustness and security.

In future papers [37] and blog posts [36], we will give more details about the design of our confidential ParaTime, data from the initial use of our system and the conclusions that we draw from that, and provide some starting points for community discussions on how the platform can evolve in the future.

References

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev. 39*, 5 (Oct. 2005), 59–74.

[2] Abraham, I., Chan, T. H., Dolev, D., Nayak, K., Pass, R., Ren, L., and Shi, E. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 317–326.

[3] Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.-P., and Porth, C. BAR fault tolerance for cooperative services. In *ACM SIGOPS operating systems review* (2005), vol. 39, ACM, pp. 45–58.

[4] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013), vol. 13, ACM New York, NY, USA, p. 7.

[5] Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., and Sonnino, A. State machine replication in the libra blockchain, 2019.

[6] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference* (2013), Springer, pp. 90–108.

[7] Benhamouda, F., Halevi, S., and Halevi, T. Supporting private data on hyperledger fabric with secure multiparty computation. *IBM Journal of Research and Development 63*, 2/3 (2019), 3–1.

[8] Bhuren, R., Werling, C., and Seifert, J.-P. Insecure until proven updated: Analyzing AMD SEV's remote attestation. *arXiv preprint arXiv:1908.11680v2* (2019).

[9] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)* (2017).

[10] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on BFT consensus. `https://github.com/tendermint/spec/releases/download/v0.6/paper.pdf`.

[11] Buterin, V. Slasher: A punitive proof-of-stake algorithm. `https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/`.

[12] Buterin, V. ERC-20 token standard. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`, Nov. 2015.

[13] Canetti, R., Feige, U., Goldreich, O., and Naor, M. Adaptively secure multi-party computation. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), pp. 639–648.

[14] Carter, J., and N. Wegman, M. Universal classes of hash functions. *Journal of Computer and System Sciences 18* (Apr. 1979), 143–154.

[15] Castro, M., and Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[16] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), IEEE, pp. 142–157.

[17] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., and Song, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy* (2019).

[18] Cosmos IBC Working Group. `https://github.com/cosmos/ics`, 2020.

[19] Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., and Yarom, Y. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 171–191.

[20] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[21] Geambasu, R., Kohno, T., Levy, A. A., and Levy, H. M. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium* (2009), vol. 316.

[22] Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing* (2009), pp. 169–178.

[23] Gomez, R. D., Adly, A. A., Mayergoyz, I., and Burke, E. R. Magnetic force scanning tunneling microscope imaging of overwritten data. *Magnetics, IEEE Transactions on 28* (10 1992), 3141 – 3143.

[24] Google. Encryption at rest in Google Cloud Platform, key management. `https://cloud.google.com/security/encryption-at-rest/default-encryption#key_management`, 2017.

[25] Gutmann, P. Secure deletion of data from magnetic and solid-state memory.

[26] Haber, S., and Stornetta, W. S. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography* (1990), Springer, pp. 437–455.

[27] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. Zcash protocol specification. *GitHub: San Francisco, CA, USA* (2016).

[28] Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S. M., and Felten, E. W. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 1353–1370.

[29] Kaplan, D., Powell, J., and Woller, T. Amd memory encryption. *White paper* (2016).

[30] King, S., and Nadal, S. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. `https://peercoin.net/whitepapers/peercoin-paper.pdf`.

[31] Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[32] Lee, D., Kohlbrenner, D., Shinde, S., Song, D., and Asanović, K. Keystone: A framework for architecting TEEs. *arXiv preprint arXiv:1907.10119* (2019).

[33] Maram, S. K. D., Zhang, F., Wang, L., Low, A., Zhang, Y., Juels, A., and Song, D. X. CHURP: Dynamic-committee proactive secret sharing. *IACR Cryptology ePrint Archive 2019* (2019), 17.

[34] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. *Hardware and Architectural Support for Security and Privacy 10*, 1 (2013).

[35] Nakamoto, S., and Bitcoin, A. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin.org/bitcoin. pdf* (2008).

[36] Oasis Foundation. Oasis foundation blog. `https://medium.com/oasis-protocol-project`.

[37] Oasis Foundation. Oasis foundation research papers repository. `https://oasisprotocol.org/researchpapers`.

[38] Oasis Team. The autonomous Byzantine soldiers fallacy. Oasis Team Blog (In preparation) `https://www.oasislabs.com/blog` Draft at `https://docs.google.com/document/d/14g1PmflraoMCqN8WsHxTaOo1lQg3Ng59xm3Ap8ggDec`, 2020.

[39] Oasis Team. Oasis mainnet phase 2: Confidential paratime (draft). In preparation, 2020.

[40] Scarlata, V., Johnson, S., Beaney, J., and Zmijewski, P. Supporting third party attestation for Intel® SGX with Intel® data center attestation primitives. *White Paper*.

[41] Stefanov, E., Shi, E., and Song, D. Towards practical oblivious ram. In *Proceedings of the NDSS Symposium 2012* (2012).

[42] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 991–1008.

[43] Vyzovitis, D., and Psaras, Y. Gossipsub: A secure pubsub protocol for unstructured, decentralised p2p overlays.

[44] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2421–2434.

[45] Wei, M. Y. C., Grupp, L. M., Spada, F. E., and Swanson, S. Reliably erasing data from flash-based solid state drives. In *FAST* (2011), vol. 11, pp. 8–8.

[46] Wood, G., et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper 151*, 2014 (2014), 1–32.

[47] Yee, B. Stake-weighted or flat voting power? In preparation, 2020.

[48] Yee, B., Kos, J., Cheng, R., Angel, Y., Berčič, J., Bukošek, A., Gilbert, P., He, W., Janež, T., Jekovec, M., Jung, E., Scott, W., Us, P., and Song, D. Nimble: A new approach to scalability. `https://www.oasisprotocol.org/researchpapers`, Oct. 2019.

[49] Zerodium. How to sell your 0day exploit to ZERODIUM. `https://www.zerodium.com/program`, Nov. 2019.

A Discrepancy Detection

A.1 Overview

Standard consensus protocols use N replicated compute nodes and converge to a consensus value if there are more than $\sim\frac{2N}{3}$ honest nodes. For verifiable computation, each compute node should have a single result; computation results are signed, and when double-signing (or "equivocation") is detected, it results in "slashing", i.e., a kind of bond forfeiture as punishment, similar to Ethereum's Slasher scheme [11]. In such schemes, if a compute node gives a "bad" answer that differs from the super-majority answer, it is essentially ignored. However, we can take advantage of the requirement that the computation is supposed to be identical (e.g., deterministic computation): we can treat any differences in output values as a sign that there was an error, *without* determining which of the two (or more) values are correct.

This is *discrepancy detection*. We detect but do not correct errors, using discrepancy detection. After a discrepancy is detected, we can use a number of methods to do conflict resolution to determine the correct result.

Discrepancy detection is an inexpensive, "fast path" optimization. When a discrepancy in the results from different compute nodes is detected, we sound an alarm and use a fallback method—a "slow path"

(which can be more expensive, such as high replication factor standard BFT-based computation)—to determine which of the results is correct and to move forward (a.k.a., conflict resolution). Those nodes that turned in incorrect results can be slashed or have their rewards withheld as a disincentive.

In the following, we will go over some background and argue for the security of the approach at conceptual level. Those who are familiar with security background may skip forward to subsection A.2.

A.1.1 Adversarial Work Factor

In security and cryptography, typically designs have security parameters that are "free parameters", the values of which can be arbitrarily set. These are called security parameters because their values determine how much resources an adversary have to expend[5] to achieve a certain probability of breaking the system. Being free parameters, they allow system implementers make worst-case estimates about how much resources an adversary might have available, and to choose large enough parameter values such that actually compromising the system would be infeasible.

The performance of security and cryptography schemes are evaluated based on the *gap* between how much additional work the defenders might do versus how much additional work the adversary must do in response to overcome the defense. A large gap makes security scaling effective: we can improve security by having the system (or users) do just a little more work in order to force the adversary to have to do a lot more work.

A.1.2 Security of Byzantine Fault Tolerant Protocols

In standard Byzantine fault tolerance (BFT) schemes [31, 1, 15], the security is "absolute" in the sense that it is logically impossible for consensus not to be reached as long as the number of Byzantine nodes does not exceed the respective thresholds and message forgery is impossible. In other words, the proofs rely only on logic and not on computational complexity assumptions. In practice, of course, protection against message forgery relies on cryptographic signatures, and thus computational complexity limits rears its ugly head. To properly ensure that security can be achieved, we need to ensure (1) that the key length chosen for the cryptographic signature scheme used is long enough and (2) that the committee size is large enough.

Let us unpack what this means.

Generally, the use of cryptography in distributed systems is straightforward, at least in principle; systems designers can readily incorporate recommendations from cryptographers. Commonly-used security parameters are chosen so that any data being protected are expected remain secure for many *decades* given projected advances in hardware cost/performance and in cryptanalytic techniques.

Committee size determination, however, is less straightforward. In order for a standard BFT protocol to succeed, we must pick a committee size N such that $N \geq 3F + 1$ where F is the maximum number of Byzantine nodes. How do we know what is a reasonable bound for F from which we would derive N? And thinking in terms of adversary work versus defender work, the incremental cost for the attacker in compromising each additional node (increasing F by 1) is just $O(1)$ (though this is an oversimplification [38, 47]); the communication complexity of the BFT protocols is $f(N) \in O(N^2)$ [2], so the incremental cost for the defender is $\frac{\partial f}{\partial F} \in O(F)$, This contrasts poorly with cryptographic approaches where there is an *exponential* relationship between defender and attacker costs, which means that we are strongly motivated to get as accurate an estimate for F as possible, because the defense overhead is so high. Of course, every BFT system has to do this, so this is nothing new.

The estimate for what is a realistic value for F is inherently imprecise and based on imperfect information. We can only make statistical, probabilistic inference about the existence (and impact) of as-yet-to-be discovered bugs in the system based on historical data on bugs and their severity, perhaps based on similar but better studied systems.

Estimating the cost to compromise F (vs $F + 1$ for the incremental cost) systems is complex. For example, an exploit based on a zero-day remote-execution vulnerability in core infrastructure components such as the Linux kernel would be a "common mode" failure in that an adversary who develops or purchases [49] it is likely to be able to compromise almost all computers used to operate consensus committee member nodes, irrespective on the entities operating them. The cost of developing or acquiring such a vulnerability and building tools to take advantage of it is expected to

[5] This is usually measured in the number of instructions or the number of logic gate operations. Such a value can later be translated into adversary cost measured in dollars based on the cost of computation.

be high (non-recurring engineering cost). The cost of applying the knowledge or running the tools to exploit systems, on a per target basis, is much lower in comparison.

In contrast, selecting an individual within an entity / organization operating consensus nodes who has authorized root access to nodes and setting up that target for blackmail (with some probability of success) might be less costly, but would result in compromise of only a single entity's nodes. This is similarly imprecise: until tested, we cannot know whether any particular individual might succumb to temptation that could lead to blackmail.

A.1.3 Probabilistic Committee Security

In the Oasis Blockchain Platform, we accept the possibility that a committee could cause system-wide failures *if the probability can be driven arbitrarily low* with comparatively little overhead. This is, in a sense, a departure from standard BFT schemes, but given that message forgery protection is probabilistic (e.g., the adversary can guess the cryptographic keys used), it is not a drastic one.

Instead of an absolute assumption that there are at most F Byzantine nodes in our consensus committee, we make a more probabilistic assumption, e.g., that each node has some probability f of being (or becoming) Byzantine, or treat F as a random variable. Given this probabilistic view, can we take advantage of the separation of computation from consensus? It turns out we can.

Often executing a smart contract involves relatively expensive computation compared with consensus: ideally, we want to avoid replicating that computation $N = 3F + 1$ times, even though we have to have N nodes for consensus. Our goal is to form a compute committee consisting of C nodes, with $C < N$, while ensuring that the cost to compromise the system would be unacceptably high for the adversary. The compute nodes will be drawn from a pool of eligible nodes. Note that validators and compute nodes are not necessarily drawn from the same pool, so the number of Byzantine nodes (or probability of being Byzantine) can differ.

Under these assumptions, we can form compute committees and calculate the probability that the resultant committee would fail when discrepancy detection is applied. By choosing a large enough committee size, we can drive the failure probability arbitrarily low. And as it turns out, for a network based on discrepancy detection to remain uncompromised for *centuries*, the committee size needed is smaller than the size required when using standard BFT techniques. This is the crux of the security efficiency argument.

Sampling Compute Committee Members We assume that the system can randomly pick C committee members from a pool in a way that the adversary cannot influence, and we further assume that the adversary is rational and will not attempt to subvert the system unless they can do so undetected, i.e., when the entire committee consists of Byzantine nodes.

This means that a detected discrepancy *should never happen*, at least for intentionally injected faults or attacks.

This does not mean discrepancy detection is useless. What it means is that we can compute the expected number of committee selections needed until an all-Byzantine committee is selected as a function of the probability f of an individual node being Byzantine, and the committee size C. Once we have this, we use worst case estimates for f and choose C so that the expected amount of time that the adversary will have to wait for a fully-Byzantine committee to be randomly chosen to exceed all reasonable levels of patience.

The probability that we have a bad committee—one for which all C members are Byzantine and thus discrepancy detection will fail—is straightforward when the pool of candidate compute nodes is large so that we can approximate it as sampling with replacement and can use the binomial distribution: f^C. If we want the single committee selection failure probability to be below some threshold P, we need only set $C > \log_f(P)$.

In subsection A.2, we also examine the case of smaller pools of candidate compute nodes, for which the hypergeometric distribution must be used.

How Unlikely is Unlikely Enough? What is an acceptable value for C and other security parameters? We need to relate C to a resource—time, money, etc—that the adversary must consume in attacking the system.

In our case, the committee selection rate and the failure probability together determines the expected time between committee selections where a bad—all Byzantine—committee is selected. That is, the probability determines the expected number of committee selections between failures, and combining that with the selection rate determines the expected *time* between failures.

The high-level security argument for discrepancy detection is this: we can choose the discrepancy detection security parameters so that the expected time between failures is no shorter than some minimum amount of time that we choose, even when using pessimistic estimates on other parameters. The next section goes into the details for how this is computed.

A.2 Security Analysis

In the reference ParaTime, we use a compute committee size of C, noting that even a single honest worker will create a discrepancy in the transaction result, and the verifying consensus node will trigger a "slow path" re-execution to reveal which execution committee members were corrupt.

We assume that an adversary can corrupt some fraction f of all available compute nodes. From this, we derive the probability P that a committee of size C will have all C members chosen from the f fraction. By controlling C, we can drive this probability to be as low as needed.

Re-execution uses an honest majority committee, where the committee size C_{hm} is chosen such that the probability P' that $\lceil \frac{C_{\text{hm}}}{2} \rceil$ or more might be chosen from the f fraction of all compute nodes is much lower than P above. The exact slow-path mechanism should be thought of as a parameter to discrepancy detection, and while honest majority is what we have implemented in the reference ParaTime, ParaTime developers can choose a different slow path verification scheme.

Next, we discuss the resource model and then derive the cost that an adversary has to incur to compromise discrepancy detection committees.

A.2.1 Adversary Costs

Like all security arguments, we show that it is infeasible for an attacker to succeed. The general approach is to provide estimates for how many resources an adversary *must* expend, relative to some security parameter, in order to successfully mount an attack; if the rate of growth of the resource function is high (e.g., exponential), then we should be able to choose security parameter values to make this infeasible.

For integrity protection, the security analysis is simpler and more straightforward. The Oasis Blockchain Platform uses Proof of Stake to prevent adversaries from overwhelming the system by directly paying to run and thus control too many validators. We use a variation of "slashing" similar to Ethereum's Slasher scheme [11] and employ stake as a performance bond against bad behavior such as double signing.

For the compute committee, in contrast, the exact details of whether (or how) to require a performance bond or to slash misbehaving compute nodes is up to the incentive design for the ParaTime. In the reference ParaTime, in order to register a compute node to participate in a compute committee, the operator must post a performance bond, and, if problems are discovered, the node operator loses their performance bond. These performance bonds offset the cost to the system to recover from the problem (slow-path processing cost) and serves as disincentive to cause denial of service or performance degradation problems. The amount of the performance bond in excess of the cost offset is a security parameter.

We noted above that a rational adversary whose goal is attack verifiable computing would not mount their attack unless they had full control of the compute committee. Because the Oasis Blockchain Platform controls how frequently committee formation occurs, the rate of choosing committee members and its size, C, are security parameters. The two determines the expected amount of time that the adversary must wait—assuming secure randomness—before a "favorable" committee composition will occur.

Smart contract authors can weigh the importance of integrity and the expected time to a bad committee in order to appropriately set minimum stake amounts and minimum committee sizes to make malicious behavior economically disadvantageous. Next, we examine how to model the effect of a ParaTime's committee size on the security of the contracts executing on that ParaTime.

A.2.2 Adaptive Adversary

The rational adversary behavior is to wait until they definitely control all members of the committee, or to ensure that the slow path verification step can be compromised. We assume that we have chosen the security parameters so that the expected cost of compromising the slow path consensus is higher than that for discrepancy detection, and will focus only on the cost of compromising the discrepancy detection committee.

A key observation is that if the adversary only acts after it has control of the committee, there is no punishment. Therefore, the risk of having their stake slashed is *not* a significant deterrent to the adversary.[6]

[6]The cost of the stake needed to operate nodes is not a deter-

Instead of analyzing how much expected stake loss is incurred by the adversary, we focus on another resource, the expected amount of *time* the adversary needs to wait before the compromise will succeed. Here, we assume that the rate of committee selection is not under the control of the adversary, so that we can use the expected number of committee selections until the adversary wins to determine the expected amount of time needed to win.

When the number of candidate compute nodes is very large, the estimates are easier. Given a committee size of C, the probability that a given committee is composed completely of compromised nodes is about f^C. The adversary has to wait for an expected $W = f^{-C}$ selections to be able to control a compromised committee, assuming that the committee selection process is random and fair, at which point the adversary has won. We want this to be (much) larger than the maximum number of committees used in the lifetime of the blockchain network, N, so we require C to satisfy:

$$C > -\log_f(N) \tag{1}$$

Plugging in $f = \frac{1}{2}$ as the fraction of compromised nodes[7] and using a (worst case) rate of 32,000 committee selections per hour and a network lifetime of 1,000 years into Eqn 1 to obtain $N \approx 2.8 \cdot 10^{11}$, we get $C > 38$. If we used a smaller value, e.g., $f = \frac{1}{3}$, then we need $C > 23.9$.

The computation for the expected number of committee selections before the adaptive adversary will win using the hypergeometric distribution is similar. Suppose we (conceptually) allowed the contract to run forever. Then the expected number of committee selections needed to get a compromised committee is $W = \frac{\binom{T}{C}}{\binom{B}{C}}$. We need to pick C, given T and what we believe to be a worst case value for B, so that the expected number of committee selections will be (much) larger than the desired maximum committee selection of N.

With $B = 500$ and $T = 1{,}000$, we need to have $C \geq 38$ to obtain

$$W \gtrapprox 5.7 \cdot 10^{11} > N$$

At $C = 38$, we will keep the adversary waiting for an expected additional 1,034 years after the network lifetime of 1,000 has expired.

Using a smaller number of Byzantine nodes, $B = 333$ out of $T = 1{,}000$, a choice of $C = 24$ results in a safety margin of more than 825 additional years after the contract expires.

Note that interestingly, for $B = 33$ and $T = 100$, a choice of $C \geq 20$ suffices. At $C = 20$, we get 2,333 additional years of safety margin. This intuition for why a smaller committee is needed than the $B = 333$, $T = 1{,}000$ case is as follows: if the committee starts filling up with Byzantine worker nodes, the remaining pool becomes depleted of Byzantine workers, so getting the last few open committee slots filled with Byzantine workers is harder/less likely.

A.2.3 Tolerating Slow Workers

As observed in MapReduce [20] and many other distributed systems, often one or a few stragglers are responsible for causing a computation to slow dramatically. With straightforward discrepancy detection, we must wait for all C committee members to finish before we can decide whether to proceed with a single consensus answer, or to fall back to the slow path computation and select a (larger) committee for re-execution.

While the performance bond is supposed to incentivize node operators to ensure that their workers are sufficiently well provisioned with hardware resources, including network bandwidth and denial-of-service protection, so that downed nodes or stragglers should be rare, such mechanisms do not guarantee that there will be no stragglers. We can modify the discrepancy detection scheme to tolerate a small number of downed nodes or stragglers (d) using the following idea: we declare consensus is reached when a timeout occurs and at least $C - d$ nodes have reported their results. Stragglers can be honest nodes that are delayed by external influence of Byzantine actors. This means that a Byzantine actor who controls $C - d$ nodes of a committee has won: the Byzantine actor will cause messages from the d nodes to be dropped or delayed, and observers will see that these are stragglers, and that the protocol was followed correctly in arriving at the result reported by the $C - d$ Byzantine nodes.

This changes the game parameters, but its essence remains the same: we choose C to be large enough so that the probability that $C - k$ nodes will be chosen from the B subset of the T nodes is sufficiently small.

rent. We assume that there is interest to be paid for borrowing the funds for stake, but this should be more than offset by earnings, since operating nodes should be a profitable endeavor.

[7] This fraction is too large for honest majority, let alone BFT schemes: using such a pool would often yield a bad committee; instead, we would have to use all nodes when $f = \frac{1}{2} - \epsilon$ instead of sampling.

The probability of winning a single committee selection is $s = \sum_{k=0}^{d} \binom{B}{C-k}\binom{T-B}{k}/\binom{T}{C}$. Suppose we used $N \approx 2.8 \cdot 10^{11}$, $B = 500$, $T = 1{,}000$ as before, and want to allow $d = 1$, which is more than 2.5% of the result for zero stragglers, then we need to pick $C \geq 43$ to obtain

$$W \gtrapprox 4.7 \cdot 10^{11} > N$$

At $C = 43$, we will keep the adversary waiting an additional 680 years.

Using a smaller number of Byzantine nodes, $B = 333$ out of $T = 1{,}000$, a choice of $C = 28$ results in a safety margin of more than 1,967 additional years.

A.2.4 Safety Margins

In addition to using the expected number of committee selections to help choose security parameters, it is useful to look at the variance, since a large variance would compel contract users to require a larger committee size to reduce the likelihood of compromise. The variance for winning a single committee is

$$\sigma_1^2 = C\left(\frac{B}{T}\right)\left(\frac{T-B}{T}\right)\left(\frac{T-C}{T-1}\right)$$

and since the committee selections are independent events, the variance for the N iteration selection game is just

$$\sigma_N^2 = NC\left(\frac{B}{T}\right)\left(\frac{T-B}{T}\right)\left(\frac{T-C}{T-1}\right)$$

For the $B = 500$, $T = 1{,}000$ case where we picked $C = 38$, the variance is $\sigma_N^2 \approx 2.5 \cdot 10^{12}$, so the standard deviation is $\sigma_N \approx 1.6 \cdot 10^6$. This is a little more than 2 days of committee selections, so *years* of extra margin of expected committee selections is plenty.

For the $B = 333$, $T = 1{,}000$ case where we picked $C = 24$, the standard deviation is $\sigma_N \approx 1.2 \cdot 10^6$, which translates to about 1.57 days of committee selections.

B Non-Adaptive Adversary

Here, we assume that the adversary is non-adaptive. That is, the number of committee members or their identities might be public, but the adversary does not (or cannot) take advantage of the information, i.e., in real-time determine if they control an entire committee. While this may not be true in general, this is an interesting design point and not revealing the committee size or the identities of committee members is something that we may explore further. There would be additional overhead associated with this: an overlay routing scheme would be needed to make identifying other committee members difficult.

Being non-adaptive, the adversary's behavior is to have their compromised compute nodes always return the answer that the adversary wants, rather than the correct answer. Once the adversary succeeds, however, the adversary de-registers all of its nodes and quits.

Let $0 \leq f < 1$ be the fraction controlled by the adversary. Given a committee size of C, the probability that a given committee is composed completely of compromised nodes—and thus the wrong answer would be undetected—is about $s = f^C$, assuming the total number of nodes to choose from is very large, or $s = \binom{B}{C}/\binom{T}{C}$ using the hypergeometric distribution, where B is the number of Byzantine compute nodes and T is the total number of nodes.

Let G be the gain that the adversary achieves by successfully corrupting the committee's contract execution, and let S represent that stake that is forfeited by the adversary when a discrepancy is detected. If the expected cost in forfeiture for a successful attack is greater than the expected gain, then it is economically infeasible for the adversary to mount attacks.

Contract authors can pick the least acceptable committee size C and the least stake to be forfeited S when discrepancies are detected based on their security requirements.[8] Suppose the total number of committees chosen to process a contract over its lifetime is N.

The probability of successfully attacking a committee at least once when at most N committees are chosen is given by $\Pr(G) = 1 - (1-s)^N$ since $(1-s)^N$ is the probability of being detected in all of the committees. Setting $a = (1-s)$, we write the expected gain as $E(G) = G(1-a^N)$.

The expected forfeiture (slashed) amount is given by

$$E(S) = \sum_{k=1}^{N} a^{k-1} \sum_{j=0}^{C-1} \frac{\binom{B}{j}\binom{T-B}{C-j}}{\binom{T}{C}} \cdot jS \qquad (2)$$

Let $S' = \sum_{j=0}^{C-1} \frac{\binom{B}{j}\binom{T-B}{C-j}}{\binom{T}{C}} \cdot j$. Since S' is independent

[8] To avoid scheduling complexity, the system only provides a few tiers of committee sizes and stake values.

of k, we can write:

$$E(S) = \sum_{k=1}^{N} a^{k-1} SS' \tag{3}$$

$$= SS' \frac{1-a^N}{1-a} \tag{4}$$

The contract author can ensure

$$E(S) > E(G)$$

$$SS' \frac{1-a^N}{1-a} > (1-a^N)G \tag{5}$$

$$S > \frac{1-a}{S'} G \tag{6}$$

Suppose $B = 500$ and $T = 1{,}000$. If the value to the attacker of a successful attack is $G = \$100{,}000{,}000$, a choice of $C = 19$ yields a forfeiture lower bound of $S = \$20.077$, a very modest value.

Note that Eqn 6 is independent of N. This makes sense, since if after one committee selection the adversary has not "won" yet, then the state of the game is the same as when we started: the expected gain and expected forfeiture should be the same as before.

C Verifying Verification

Discrepancy detection allows us to verify that ParaTime compute nodes performed the same required computation to obtain the output. An important consideration is that, under some security/fault tolerance models, a compute node's output should contain additional check values that allow discrepancy detection to verify that all the appropriate processing occurred in that compute node.

Such verifications are not necessary with TEE-based ParaTime nodes: the TEE model provides remote attestation of code identity, and this makes code modifications easily detected. These verifications are also not necessary under the standard Byzantine fault tolerance model where nodes are either Byzantine or honest, but *is* required under the Byzantine/Altruistic/Rational (BAR) model [3], where rational actors take actions based on their self interest, so will behave honestly or lie, depending on economic incentives.

Let us look at a concrete example.

While executing transactions, the compute committee nodes must read and write untrusted external transactional persistent storage. While mathematically we require the verification of read proofs and write proofs (e.g., checking peer hashes in a Merklized data structures) to know that the read/write operations were done correctly, a rational actor may replace the proof verification code with code that bypasses the proof verification in an effort to improve performance.

We would like to ensure that this undesirable—but rational—behavior is detected. Otherwise, we create a situation where we may have a committee of rational nodes, none of which would detect misbehaving storage nodes! To incentivize verification, we use the following ideas to leverage discrepancy detection:

- Proof verification of authenticated data structures requires computing cryptographic hashes, e.g., the hash of a subtree in an n-ary tree is the hash of the hash values of its children. These required hash computation is much more expensive than the *comparisons* needed to check whether the result is correct.
- We can ensure that the internal hash values are computed by including them and any other intermediate result—or a summary thereof—as part of the output of the computation as auxilliary data.
- Discrepancy detection compares the outputs of all replicas. Thus, if there is at least one honest node that will correctly compute this auxilliary output, then any node that did not will be detected.

Note that even if the storage queries were to obtain results from a cache, rather than from the storage servers, the cache should either verify the proof—and provide an interface to provide the intermediate values goes into the verification along with the answer—or make it the responsibility of the cache client to verify the proofs and generate those intermediate value itself. The cached data should enable the cache to generate the same proofs as would be generated by the storage servers.

How do we compute this auxilliary information that proves that most of the expensive verification steps were done? We require that the intermediate expensive hash outputs (hashes associated with internal nodes) computed during storage proof checking will themselves be used as the input to another hash function. This hash function does not have to be cryptographic in nature, but instead can be universal hashing [14], which we will denote $\mathtt{h}_r(\widehat{\mathsf{aux}})$: the compute nodes can be given the (random r) choice of a hash function which will be given all of the intermediate values as input. Because this is not a collision resistant cryptographic hash function, its computation is

much cheaper.[9] The result of this hash is, however, unpredictable without first finding its inputs.

A compute node could compute the auxilliary output without actually doing the full verification, i.e., it could omit the actual comparisons that check the hash values against what was expected, even though it computes $\mathtt{h}_r(\widehat{\mathtt{aux}})$. This, however, seems unlikely, since the comparisons are extremely cheap compared to the cost of the cryptographic hash value computation—the savings would be negligible.

To fully close the gap, we *could* require storage nodes, under the control of a secure, deterministic pseudorandom number generator, to issue bogus proofs with some (low) probability. This introduces a pipeline hazard/bubble, since the storage clients must (1) detect the bogus proof (and potentially earn a reward), and (2) perform another remote procedure call to send a signed reply saying the proof was either accepted or rejected, and, if rejected, get the correct proof. Fortunately, since the expected gain in performance for omitting the hash comparison(s) needed to check the storage proofs is extremely low and *not* detecting the bogus proof results in slashing, the frequency of bogus proof injection can be very low. We do not currently implement bogus proof injection.

[9] The cost of universal hashing is similar to that of polynomial reduction used to compute error correction codes (ECC).