Rethinking General-Purpose Decentralized Computing

Enis Ceyhun Alp*
enis.alp@epfl.ch
EPFL

Eleftherios Kokoris-Kogias*
eleftherios.kokoriskogias@epfl.ch
EPFL

Georgia Fragkouli
georgia.fragkouli@epfl.ch
EPFL

Bryan Ford
bryan.ford@epfl.ch
EPFL

Abstract

While showing great promise, smart contracts are difficult to program correctly, as they need a deep understanding of cryptography and distributed algorithms, and offer limited functionality, as they have to be deterministic and cannot operate on secret data. In this paper we present Protean, a general-purpose decentralized computing platform that addresses these limitations by moving from a monolithic execution model, where all participating nodes store all the state and execute every computation, to a modular execution-model. Protean employs secure specialized modules, called *functional units*, for building decentralized applications that are currently insecure or impossible to implement with smart contracts. Each functional unit is a distributed system that provides a special-purpose functionality by exposing atomic transactions to the smart-contract developer. Combining these transactions into arbitrarily-defined workflows, developers can build a larger class of decentralized applications, such as provably-secure and fair lotteries or e-voting.

ACM Reference Format:
Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, and Bryan Ford. 2019. Rethinking General-Purpose Decentralized Computing. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3317550.3321448

*Equal contribution

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321448

1 Introduction

Decentralized systems have been studied extensively over the past decades [14, 17, 40, 44, 50]. One of the most recent, and arguably the most popular, decentralized system is the blockchain and its most promising application is smart contracts. Smart contracts are user-defined programs that are executed by a network of nodes that reach consensus on program state. Ethereum, the world's second largest cryptocurrency, claims to be a general-purpose decentralized computer that can support execution of arbitrary code in smart contracts. However, Ethereum and its successors [1, 2] fall short of their claim due to several shortcomings. First, they do not support non-deterministic operations and cannot securely operate on private data. Second, executing every contract on every node limits the overall throughput of the system, hence degrading its performance. Finally, verifying the execution of a smart contract might incur a high computational cost and cause the *verifier's dilemma* [30], where nodes are incentivized to deviate from the protocol by skipping the verification process. This phenomenon can undermine the security of smart contracts.

Enabling expressive and secure decentralized computation has the potential of revolutionizing the way web applications are deployed. First, smart contracts that can hold private data could revolutionize data sharing and give rise to data markets controlled by users. Second, efficient smart contracts that can support thousands of transactions per second could provide a more secure and fault-tolerant alternative to the centralized cloud-computing infrastructure. Finally, smart contracts that provide unlinkability (*e.g.*, Neff shuffles [34], which is impossible to implement in Ethereum) could revolutionize governance and decision-making processes. The current landscape, however, is fragmented into million-dollar startups that try to deliver some of these promises by rejecting the existing smart-contract offerings and instead (re)inventing standalone components that might reintroduce centralization (*e.g.*, using trusted hardware to hold private data).

In this paper, we argue that current smart-contract systems fail to either enable or sufficiently support a wide range of applications due to their *monolithic architecture where consensus and code execution are tightly coupled.* In order to

overcome these limitations, we present Protean, a system for general-purpose decentralized computing that leverages modularity, a well-studied design principle in computer systems [19, 22, 24, 48].

Protean is an ecosystem of special-purpose modules called *functional units* that are formed by partitioning the nodes of the system into subsets. Each functional unit provides a unique specialized computation (*e.g.*, consensus [13, 27], randomness generation [45], private-data sharing with access control [9, 25], or verifiable anonymity [34]) that is collectively performed by its nodes, and that is cryptographically guaranteed to be valid. In this way, Protean replaces the monolithic execution model of previous systems, where every node performs the same computations, with a modular execution model, where different sets of nodes perform different computations in isolation from each other.

To summarize, thanks to its modular design, Protean achieves the following: First, compared to the current smart-contract systems, Protean enables users to build a larger class of decentralized applications by supporting a richer set of computations. Protean achieves this by delegating computations to the relevant functional units where they are performed only by the nodes of the unit, in isolation from the rest of the system. Second, Protean provides great flexibility to application developers as it can support multiple implementations of the same specialized computation with different security-performance trade-offs. Third, since functional units can perform different computations in parallel, Protean can achieve a better throughput than the smart-contract systems where execution of a computation is replicated across the network. Finally, Protean can mitigate verifier's dilemma by assigning nodes to computations they prefer, as opposed to executing the whole smart contract.

2 Motivating Example

To get a better sense of the shortcomings of the current smart-contract systems and how Protean addresses them, we take a closer look at a popular class of decentralized applications, namely decentralized lotteries [16, 32]. Consider a lottery smart contract that runs on Ethereum [47]. A lottery round begins with the lottery organizer generating lottery tickets and specifying a well-defined time period (measured in number of blocks) during which participants are allowed to purchase lottery tickets by depositing money to the lottery account. Once the ticket acquisition period ends, Ethereum miners, which are nodes responsible for reaching consensus on the execution of smart contracts, executes the smart contract that is provided by the lottery creator. This smart contract would typically parse some source of public randomness; use the randomness to select the lottery winner; and transfer money to winner's account.

In order to guarantee the fairness of the lottery, it is crucial to have a source of randomness that is unbiasable and unpredictable. However, generating randomness in Ethereum is a challenging task since it is a deterministic system that inherently lacks randomness. To work around this problem, smart-contract developers create their own pseudo-random number generator (PRNG) implementations. However, researchers have shown that this practice causes vulnerabilities [10, 39, 43] in smart contracts as malicious users can manipulate the source of randomness.

A typical approach to generating randomness in Ethereum is parsing the randomness from a future block (*e.g.*, its timestamp, nonce or blockhash). This approach is vulnerable to attacks where a malicious miner can manipulate the mining process to bias the values of the block variables; hence the randomness. Another method for generating randomness is using a *commit-then-reveal* protocol. In this approach, each user chooses a secret and shares their commitment to the secret with other users. Later, users reveal their secrets and a random value is calculated by combining the secrets. This approach is susceptible to attacks where a user can choose not to reveal their secret to bias the randomness generation to their advantage (bias-via-abort [45]). Finally, another approach is to use an external oracle [3] that sends random values to the smart contract. The drawback of this approach is that it requires trusting a single entity for the quality of randomness; hence creating a single point of trust.

Although there are secure multi-party computation (SMPC) protocols for generating randomness [12, 21, 45], it is not possible to run them on Ethereum's monolithic execution model as they involve operations that are non-deterministic and/or use private data. In the following sections, we first describe how Protean avoids the shortcomings of the existing systems with its modular design. Then, we come back to lotteries and demonstrate how Protean can support secure and fair decentralized lotteries.

3 System Overview

In this section, we give an overview of Protean's design.

3.1 Goals

Protean has the following primary goals:

- *General-purpose computation*: Users can implement and execute arbitrary decentralized applications (that current smart-contract systems cannot support).
- *High performance*: Nodes can provision their resources to specialize in supporting a specific functionality, thereby improving the efficiency and performance of the system.
- *Extensibility*: The set of provided functionalities can be extended without major changes to the system.
- *Decentralization*: No single point of failure or compromise.

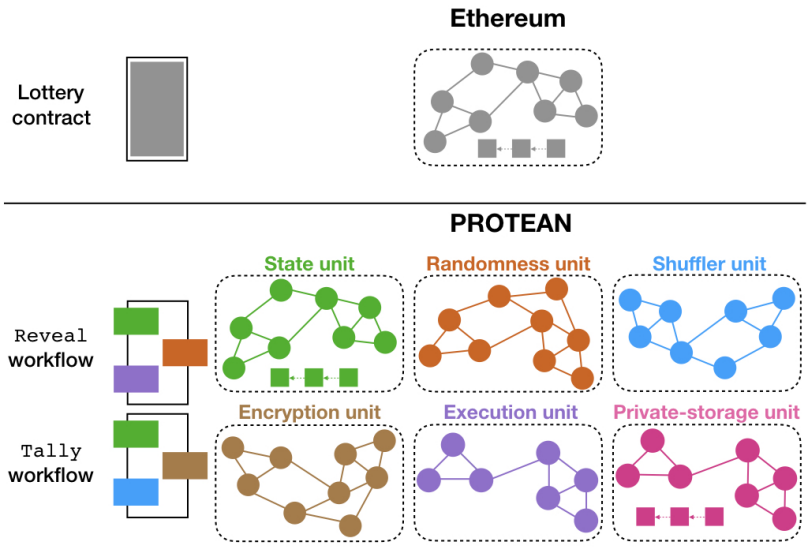

Figure 1. Ethereum vs PROTEAN: Ethereum's monolithic architecture requires every smart contract to execute on every node. PROTEAN's modular architecture delegates execution of a workflow to specialized functional units

3.2 System Design

PROTEAN consists of a collection of special-purpose modules called *functional units* (see Figure 1). Each functional unit is composed of a set of physical nodes that are available in PROTEAN; and they collectively perform a specialized computation. Associated with each specialized computation is a set of operations, namely *transactions*, that have well-defined semantics, are executed atomically by all the nodes in the unit, and trigger publicly-verifiable state transitions. Transactions serve as secure building-blocks that are used for building general-purpose decentralized applications. From the standpoint of an application developer, executing a transaction is analogous to making an API call to a software service: The service is treated like a black box and the complexity of its implementation is hidden from the users of the service.

Developers build a decentralized application by creating *workflows* and storing them in the publicly-visible system state. Each workflow contains a list of transactions and the execution dependencies between them, and performs a specific task of the application. Since workflows can contain transactions that are supported by different functional units, executing a workflow might require coordination between multiple units. Sharded blockchains, which are architecturally similar to PROTEAN, solve this problem by adopting a communication model that is either client-driven [28] or shard-driven [7]. PROTEAN uses the client-driven model, where users communicate with functional units to direct the execution of workflows, as it simplifies the design of functional units and is a better fit for the target applications (*e.g.*, lotteries, e-voting, auctions) that require clients to interact with the system. For applications that is not compatible with this design, we can use a *driver unit* [7] to direct the execution of workflows and run Byzantine-fault tolerant consensus.

Users join an application by executing one of its workflows. Upon retrieving the workflow from the system state, users contact a system functional unit, called *compiler unit*, that creates an *execution plan* for the workflow. An execution plan contains: (1) identities of the functional unit nodes (*i.e.*, network addresses and cryptographic identities) that are going to execute the transactions, (2) a transaction dependency-graph, and (3) additional parameters that are required for executing the transactions (*e.g.*, barrier point for joining the lottery). Additionally, compiler unit cryptographically signs the execution plan so that users can prove to the functional units that their request to execute a transaction is valid. Users follow the execution plan to submit transaction requests to functional units through the APIs that are specified and made publicly-available by the units.

3.3 Functional Units

Below, we list some of the functional units that can be used in PROTEAN. We note that this is not an exhaustive list as arbitrary functional units can be added to the system, as we discuss in Section 3.5.

- **Randomness unit**: This unit produces unbiasable, unpredictable and publicly-verifiable randomness [12, 21, 45].
- **State unit**: This unit exposes a key-value store that provides serializability, which is guaranteed by every blockchain system whether using Nakamoto [33],

PBFT [8, 27] or sharded consensus [28]. It verifies that the state updates are correct (*i.e.*, changes to a specific key are signed by the owner of the key) and consistent (*i.e.*, decides whether a state update commits or aborts based on the staleness of the state it changes [8, 28]).
- **Execution unit**: This unit is for executing arbitrary user-defined code and can be implemented by Ethereum Virtual Machine (EVM) [47] or WebAssembly (WASM) [4]. Code has to be deterministic for the nodes to reach consensus on its execution. Execution unit does not give any guarantees about the semantic correctness of the executed code.
- **Private-storage unit:** This unit enables private-data sharing with an auditable access-control mechanism. It guarantees the availability of data as described in Calypso [25].

The aforementioned functional units expose correct processes that are run by a distributed set of nodes. For now, we assume that nodes respect the adversarial model under which the units are proven to be secure. We discuss the dynamic evolution of functional unit memberships in Section 3.5.

3.4 Security Architecture

As we describe in Section 3.2, users orchestrate the execution of workflows by communicating with functional units to execute transactions. Due to our design decision, users can maliciously or inadvertently deviate from the correct execution of a workflow. More specifically, a user can misbehave by (1) making transaction requests to functional units that they are not supposed to, (2) not respecting the execution dependencies of transactions, and (3) submitting the same transaction request to a functional unit many times to perform a replay attack.

To address the first two cases, functional units run a scalable collective witnessing protocol, namely CoSi [46], to collectively sign (cosign) the output of transactions. When executing a workflow, users gather collective signatures from each functional unit, and send the set of signatures and the execution plan with every transaction request they make. When a functional unit receives a request, it verifies the collective signature on the execution plan, which is generated by the compiler unit, and checks that its identity is in the execution plan. Then, based on the dependency graph and the set of collective signatures, it verifies that the user has already executed all of the transactions that the current transaction depends on. If everything checks out, the functional unit safely proceeds with executing the transaction and returns a cosigned output to the user.

Although cosigning addresses the first two challenges, users can still use a valid execution plan to launch a replay attack on a functional unit by repeatedly submitting the same transaction request. This way, a malicious user can either carry out a denial-of-service attack, thereby making it hard for other users to run their transactions, or try to bias the output of a transaction (*e.g.*, randomness generation).

To mitigate replay attacks, functional units maintain *caches* that store the output of successfully executed transactions. Each entry in the cache is a key-value pair, where the key is a digest of the execution plan of the transaction and the value is the output of the transaction. Upon receiving a transaction execution request, a functional unit first computes the key and checks whether an entry for it already exists in its cache. If so, it returns the corresponding value without executing the transaction again. It is an open problem to devise a mechanism to keep the cache sizes under control without disrupting the execution of applications.

3.5 Management and Governance

So far, we have assumed that nodes have static unit-memberships and respect the threat model under which their functional unit is proven to be secure. However, this is an unrealistic assumption as having static unit-memberships means that Protean cannot protect the security of a functional unit if its nodes are compromised over time.

One potential solution to this problem is using OmniLedger's sharding mechanism [28] to securely assign nodes to functional units and periodically change their memberships to improve the long-term security of functional units. More specifically, Protean can operate in fixed time intervals, called *epochs*, during which the system configuration does not change. Nodes can join the system by creating Sybil-resistant identities and registering them. Then, at the beginning of each epoch, an administrative workflow (similar to Ethereum's validator management contract [6]) can run automatically to randomly assign the registered nodes to functional units. Protean can use a system functional unit called *directory unit* that uses skipchains [26, 35] to store the identities of the nodes (*i.e.*, network addresses and cryptographic identities) and their functional-unit assignments, thereby helping users with node and functional-unit discovery.

Although the aforementioned mechanism periodically assigns nodes to functional units, it does not consider the preferences of the nodes or whether the nodes have the necessary resources to efficiently perform the specialized function of their unit. For instance, if a lightweight mobile client is assigned to a functional unit that performs heavy cryptographic operations, it can degrade the performance of the functional unit, and in turn of the overall system. It is an open question to assign nodes to functional units that best fit their capabilities without sacrificing the security of functional units.

It is necessary for Protean to be able to update the set of functional units and transactions over time so that it can better serve the computational needs of users. To this end, Protean needs a consensus mechanism to decide whether a functional unit should be added to or removed from the

system. For instance, in order to introduce a new functional unit, parties that are part of the consensus group could vet the code and transactions of the functional unit. This decision-making process is a part of *blockchain governance*, which is an open problem with various proposed solutions [18]. One approach that can be used in Protean is the stake-based governance [11, 38] where legitimate stakeholders have voting power proportional to their stakes in the system.

4 Applications

In this section, we demonstrate how Protean can be used for building decentralized applications. To this end, we first present two different implementations of a decentralized lottery. Later, we describe an e-voting application to highlight the range of applications that Protean can support.

4.1 Decentralized Lotteries

As we mention in Section 2, smart-contract developers who choose to implement their own PRNG can face serious security and correctness problems in their applications. We present two different approaches for building secure decentralized lotteries that use different functional units to generate unbiasable and unpredictable randomness. The first approach uses the *randomness unit* that runs an SMPC-based protocol [45]. The second approach uses the *private-storage unit* [25] that generates randomness based on user inputs.

In the first approach, the smart-contract developer defines three workflows: *initialize*, *join*, and *reveal*. The lottery organizer runs *initialize* to setup the lottery by creating lottery tickets and defining deadlines, such as when to end the ticket purchase and finalize the lottery. Afterwards, anyone who wants to participate in the lottery runs *join* to purchase tickets by transferring the necessary funds to the lottery account. Once the ticket purchase is over, any participant can run *reveal* to parse randomness from the randomness unit and transfer all the data to the execution unit to decide the winner of the lottery. Since the randomness unit runs a secure protocol that is guaranteed to generate unbiasable and unpredictable randomness, our application avoids the security and correctness problems of the Ethereum-based lotteries that we highlight in Section 2.

In the second approach, we define the same workflows, however, with different transactions as we use a different set of functional units. More specifically, *initialize* works the same as before; but *join* is used by the participants to commit to their randomness. Each participant's randomness is then shared with the private-storage unit to guarantee that it will eventually be revealed. Once the deadline for participating in the lottery has passed, any authorized user can run *reveal* to have the private-storage unit reveal the random numbers that are submitted by the participants. The user sends these random numbers to the execution unit to compute a final random value using the input from the participants and pick the winner. This approach also guarantees the unbiasability and unpredictability of the generated randomness as it ensures: (1) all commitments are cryptographically hidden from the public until they can be revealed, and (2) all commitments are eventually atomically revealed after the deadline; hence avoiding a bias-via-abort [45].

4.2 E-Voting

A decentralized e-voting application can be created by defining three workflows: *setup*, *vote*, and *tally*. At *setup*, election authorities perform the necessary steps to initialize the election (*e.g.*, creating ballots). Afterwards, voters run *vote* to encrypt and cast their ballot. Once the voting period is over, any user can run *tally* to decrypt the ballots and verify their correctness, and compute the election results.

In order to implement a decentralized e-voting application, we need to tackle two challenges: decentralizing the process of tallying the votes and protecting the privacy of voters. To this end, besides the more generic functional units (*e.g.*, state unit, execution unit), we need two new functional units, namely *encryption unit* and *shuffler unit*. Encryption unit runs a distributed key generation (DKG) algorithm [21] to generate a collective private-public key pair such that the private key can only be reconstructed and used by a threshold of key shares, which are distributed to the nodes of the functional unit. Voters submit their ballots to the system after using a secure threshold-encryption protocol [42] to encrypt them with the collective public key. A ballot can be decrypted later only if a threshold of nodes combine their secret shares. Shuffler unit runs a verifiable-shuffling protocol [34] that permutes and re-encrypts the ballots to remove the link between the voter and their vote. It also generates zero-knowledge proofs to prove the correctness of the permutation. We note that the same e-voting application cannot be securely implemented in Ethereum as some of the operations use secret data (*e.g.*, private key) that cannot be put on the blockchain where it is publicly visible [41].

5 Related Work

Numerous systems have been proposed to address the privacy and performance limitations of the smart contracts. We discuss how these systems compare to Protean.

Hawk [29] supports privacy-preserving smart contracts by executing contracts off-chain and verifying the correctness of the execution on-chain via zero-knowledge proofs (ZKPs). Hawk guarantees privacy at different levels as it hides the details of monetary transactions from public and protects the private inputs of a contract both from public and other parties who participate in the contract. However, Hawk has several limitations: First, Hawk is limited in the range of applications it can support. Second, Hawk relies on a trusted manager to not reveal the private inputs, which creates a single point of compromise in the system. Finally, ZKPs have

high computational overhead that can degrade the overall performance of the system.

Chainspace [7] is a smart-contract platform that employs the well-studied sharding technique to achieve horizontal scalability. Additionally, Chainspace uses a technique that is similar to Hawk's to support privacy-preserving smart contracts. Users execute contracts on the client side and generate ZKPs to prove the correctness of the execution. Then, consensus nodes check the proofs to validate the execution without requiring access to data that is used during the execution. As Chainspace offloads smart-contract execution to the client side, it is users' responsibility to both correctly implement the algorithms that are required for their applications, which can be difficult and insecure, and generate the ZKPs, which can be infeasible to do on users' devices. In contrast, Protean adopts the opposite approach by providing the users with services that they can use to build and execute their applications within the system. Therefore, we argue that Protean can support a wider class of applications than Chainspace.

Arbitrum [23] argues that executing every contract on every node limits the scalability and privacy of smart contracts in Ethereum. Arbitrum addresses this shortcoming by limiting contract execution to a set of nodes (managers) that agree off-chain on the output of smart contract execution. If managers unanimously agree on the output of a computation, they send a digital signature to the verifier (*e.g.*, consensus protocol, smart contract) where the correctness of execution is validated by verifying the signature. In case the managers disagree, they participate in a refereed game, which is arbitrated by the verifier, to determine who is right. The drawback of Arbitrum is that it relies on financial incentives to achieve its goals of scalability and privacy. Managers that are not driven by financial incentives can continuously trigger the refereed game to degrade the performance of the system and/or reveal potentially sensitive data.

Ekiden [15] supports confidentiality-preserving smart contracts by performing smart-contract computations off-chain in a trusted execution environment (TEE). Executing smart contracts in a TEE enables Ekiden to safely compute on private data, prove the correctness of the execution, and support a wider range of smart contracts. However, TEE in Ekiden is a single point of compromise in terms of the integrity and privacy of the smart contract execution, which conflicts with the idea of decentralization.

Aspen [20] takes a different approach to sharding blokchains by introducing service-oriented sharding. Aspen builds on a multi-blockchain structure where each blockchain exposes a different service and stores only the transactions that belong to its service. In addition to high scalability, Aspen also achieves extensibility by enabling users to introduce new services without disrupting the operation of others. Aspen and Protean both move away from monolithic architectures and have overlapping goals in terms of performance and extensibility. However, Aspen does not support general-purpose decentralized applications.

6 Discussion

In the previous sections, we have described how Protean uses the existing decentralized protocols in functional units to enable a wide range of applications that current smart-contract systems cannot support. We now discuss the future directions in designing new functional units to further enrich the type of applications that Protean can support.

Accessing real-world data (*e.g.*, result of a sports match, stock prices) from an external source (*e.g.*, the Internet) is one of the biggest needs of smart contracts. Current solutions that provide data feed to smart contracts, however, do not satisfy our goal of full decentralization: Users have to trust either one of the few data feed platforms [3, 5] or a system that relies on trusted hardware to guarantee the authenticity of data [49]. A decentralized data-feed system that can provide provably-authentic data in a privacy-conscious manner (*i.e.*, without leaking information about data requests) can form an *oracle unit* in Protean to support new classes of applications, such as prediction market and insurance market.

Due to Ethereum's monolithic design, its execution environment, EVM, has several limitations. For instance, cryptographic operations on some widely-used elliptic curves are either not supported natively in EVM; or they are supported by precompiled contracts, but the financial cost of executing them is high [31, 37, 49]. We believe that new virtual machines with specialized instruction-sets to perform a dedicated computation (*e.g.*, verifying Ed25519 cryptographic signatures) can create a more efficient and expressive alternative to EVM. These virtual machines can be used in Protean to provide more functionalities with high efficiency.

Finally, an open question that is worth exploring is incentivizing functional units to operate under the threat model that they are proven to be secure (*e.g.*, Byzantine setting). Game-theoretic mechanisms [36] present a promising way to mathematically enforce a certain fraction of the nodes in a functional unit to act honestly. We envision an *arbitrator unit* in Protean that implements this logic and assigns payoffs to nodes. The challenge is finding payoffs that make sense in the Byzantine setting with open participation.

7 Conclusion

We conjecture that the limitations of smart contracts stem from their monolithic architecture where consensus and code execution are tightly coupled. To address these limitations, we have presented Protean, a general-purpose decentralized computing platform that leverages modularity. Thanks to its modular design, Protean can support a large class of decentralized applications that are currently insecure or impossible to implement with smart contracts.

Acknowledgments

We would like to thank Christian Mouchet, Cristina Basescu, David Froelicher, Gaylor Bosson, Kelong Cong, Kirill Nikitin, Ludovic Barman and Philipp Jovanovic for their comments and feedback. This project was supported in part by the grant #2017 − 201 of the Strategic Focal Area "Personalized Health and Related Technologies (PHRT)" of the ETH Domain and by grants from the AXA Research Fund, Byzgen, DFINITY, and the Swiss Data Science Center (SDSC). Eleftherios Kokoris-Kogias is supported in part by the IBM PhD Fellowship.

References

[1] eosio | Blockchain software architecture.

[2] POA Network.

[3] Provable - Oraclize 2.0 - blockchain oracle service, enabling data-rich smart contracts.

[4] WebAssembly.

[5] Augur: A Decentralized Oracle & Prediction Market Protocol.

[6] Ethereum's Sharding Specification. 2018.

[7] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A Sharded Smart Contracts Platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[9] E. Androulaki, C. Cachin, A. De Caro, and E. Kokoris-Kogias. Channels: Horizontal Scaling and Confidentiality on Permissioned Blockchains. In *European Symposium on Research in Computer Security*, pages 111–131. Springer, 2018.

[10] N. Atzei, M. Bartoletti, and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186. Springer-Verlag New York, Inc., 2017.

[11] M. Borge, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, and B. Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *1st IEEE Security and Privacy on the Blockchain*, Apr. 2017.

[12] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In *19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[13] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[14] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[15] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141*, 2018.

[16] CuriosMind. World's Hottest Decentralized Lottery Powered by Blockchain, Feb. 2018.

[17] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *12th USENIX Security Symposium*, Aug. 2004.

[18] F. Ehrsam. Blockchain Governance: Programming Our Future . 2017.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[20] A. E. Gencer, R. van Renesse, and E. G. Sirer. Short Paper: Service-Oriented Sharding for Blockchains. *Financial Cryptography and Data Security 2017*, 2017.

[21] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt*, volume 99, pages 295–310. Springer, 1999.

[22] M. Jung, P. Dalbhanjan, P. Chapman, and C. Kassen. Microservices on AWS, 2017.

[23] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1353–1370. USENIX Association, 2018.

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[25] E. Kokoris-Kogias, E. C. Alp, S. D. Siby, N. Gailly, L. Gasser, P. Jovanovic, E. Syta, and B. Ford. CALYPSO: Auditable Sharing of Private Data over Blockchains. Cryptology ePrint Archive, Report 2018/209, 2018.

[26] E. Kokoris-Kogias, L. Gasser, I. Khoffi, P. Jovanovic, N. Gailly, and B. Ford. Managing Identities Using Blockchains and CoSi. Technical report, 9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016), 2016.

[27] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[28] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598, 2018.

[29] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[30] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719. ACM, 2015.

[31] P. McCorry, S. F. Shahandashti, and F. Hao. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375, 2017.

[32] A. Miller and I. Bentov. Zero-collateral lotteries in Bitcoin and Ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*, pages 4–13. IEEE, 2017.

[33] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[34] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125. ACM, 2001.

[35] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287. USENIX Association, 2017.

[36] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[37] T. Oberstein. EIP 665: Add precompiled contract for Ed25519 signature verification, Mar. 2018.

[38] R. Red. What is on-chain cryptocurrency governance? Is it plutocratic? 2018.

[39] A. Reutov. Predicting Random Numbers in Ethereum Smart Contracts. 2018.

[40] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[41] I. A. Seres. Implementing an e-voting protocol with blind signatures on Ethereum. 2018.

[42] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Advances in Cryptology — EUROCRYPT'98*, pages 1–16, 1998.

[43] J. Song. Attack on Pseudo-random number generator (PRNG) used in 1000 Guess, an Ethereum lottery game (CVE-2018–12454), July 2018.

[44] I. Stoica, R. T. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[45] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable Bias-Resistant Distributed Randomness. In *38th IEEE Symposium on Security and Privacy*, May 2017.

[46] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy*, May 2016.

[47] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

[48] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 63–76, 1987.

[49] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, Nov. 2016.

[50] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.