EPFL

Thèse n° 8858

Towards General-Purpose Decentralized Computing with Permissionless Extensibility

Présentée le 19 janvier 2024

Faculté informatique et communications
Laboratoire de systèmes décentralisés et distribués
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Enis Ceyhun ALP

Acceptée sur proposition du jury

Prof. C. González Troncoso, présidente du jury
Prof. B. A. Ford, directeur de thèse
Prof. A. Miller, rapporteur
Prof. J. Cappos, rapporteur
Prof. K. Argyraki, rapporteuse

École
polytechnique
fédérale
de Lausanne

2024

It was always me versus the world
Until I found it’s me versus me
— Kendrick Lamar Duckworth

No conception is immaculate, man
— Tarik Luqmaan Trotter

To my family,
and to the memory of my grandma, Özgülcüğüm. . .

Acknowledgments

After a 7-hour bike ride on a hot July day in 2022, I told my friends: *I think this ride was a great analogy of my PhD because: (1) it lasted longer than I expected, (2) halfway through it I regretted that I started it, (3) the last parts made me want to throw up, and (4) I felt a mix of happiness, confusion, and disgust in the end.* Although I was definitely exaggerating for humor purposes, it is safe to say that the last 2+6 years have not been easy as there were more downs than ups. The biggest reason why and how I was able to push through the hard times and get to the finish line is the people that surrounded and supported me during this process. It is my pleasure to express them my gratitude and remind them that this thesis is theirs as much as it is mine.

First, I would like to thank my advisor Bryan Ford. Bryan taught me how to do research, and clearly and effectively explain it to others, both verbally and in writing. There were many times where I showed up to our meetings feeling stuck and stressed, but I always left his office (or Zoom room) feeling relieved and much better about my research and myself thanks to his technical advice, positive attitude, and kindness. Thank you, Bryan, for patiently investing your time and energy in me to see me through my PhD. I cannot emphasize enough when I say this thesis would not have been possible without you. I am grateful to Paul Barford for accepting me as a research student at UW-Madison and offering me an internship at Comscore. During this internship, I was lucky to have Matt Malloy as a mentor with whom I had the opportunity to work closely on the Internet device graphs project. Thank you, Paul and Matt, for giving me an opportunity to do research with you and supporting my application to EPFL with your recommendation letters. I am indebted to Cem Ersoy for supervising my bachelor project and supporting my graduate school application not once, but twice. I would like to thank Katerina Argyraki, Justin Cappos, and Andrew Miller for serving on my thesis committee and providing feedback on this dissertation, and Carmela González Troncoso for presiding over my thesis committee.

During the six years I spent at DEDIS, I had the pleasure to meet and work with many great researchers and engineers: Jeff Allen, Ludovic Barman, Cristina Băsescu, Pierluca Borsò-Tan, Gaylor Bosson, Simone Colombo, Kelong Cong, Henry Corrigan-Gibbs, Vero Estrada-Galiñanes, Georgia Fragkouli, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Noémien Kocher, David Lazar, Louis-Henri Merino, Shailesh Mishra, Gaurav Narula, Pasindu Tennage, Jean Viaene, and Haoqian Zhang. Thank you all for all the discussions, feedback on my drafts and presentations, lunches, and group outings. A special thanks goes to Cristina, Ewa Syta, Georgia, Linus, Pasindu, and Philipp for contributing to this thesis and being wonderful collaborators. I

would like to especially thank Linus for helping me with my Cothority & Onet problems, even after leaving DEDIS. A big thanks to Margaret Church, Anouk Hein, Sandra Renata Hünsch, and Fara Malala Raharisolo for protecting us from getting crushed by the gears of the Swiss and EPFL bureaucracy, helping us with administrative tasks, and organizing social events. Finally, it was a delight to share the corridor and the floor kitchen with our sister labs LDS and SPRING. I would like to thank Sylvain Chatel, Kasra EdalatNejad, Bogdan Kulynych, Wouter Lueks, Mickaël Misbach, Christian Mouchet, Apostolos Pyrgelis, Sandra Siby, and Theresa Stadler for coffee breaks, beers at Satellite, and numerous other activities both on- and off-campus.

Kristina Gligorić, Novak Kaluđerović, and Panagiotis Sioulas were my first friends in Lausanne as we have started our PhDs at the same time. Even though I cannot prove it, I think our "Balkan-ness" brought us together and I am grateful to you for all the fun times we had, especially in the first years of our PhDs. I cannot thank Olivier Blanvillain, Mia Primorac, and Georg Schmid enough for introducing me to the wonderful game of Spikeball and various card games. Our regular game sessions at Parc de Milan were always fun and a much-needed distraction from the uncertainties of the COVID era. Hannah Casey and I bonded over our mutual love for black tea (Hannah, I know you are half-British so you cannot help it, but you really have to stop putting "liquid milk" in your tea!), cats, Jetée, and the conspiracy theory that the birds are not real. Thank you, Hannah, for your friendship and emotional support in the final years of my PhD and sorry for using you as a guinea pig in CS-234 :) Lukas Bischofberger and I crossed paths once again due to my moving to his home country. Thank you, Lukas, for crossing the Röstigraben to visit me in Lausanne, doing the Tour de Léman with me, and for hosting me in Zurich. I am looking forward to many more bike rides with you in the Swiss mountains. An ancient Turkish proverb says: *Turks find Turks in the foreign lands*. So it was no surprise that the Turks were a part of my social circle in Lausanne. I have countless memories of nights out, house parties, dinners, and us simply goofing around with Pınar Akyazı, Ahmet Alacaoğlu, Beril Beşbınar, Işınsu Katırcıoğlu, Ali Kavis, Sena Kıcıroğlu, Sinem Sav, Utku Şirin, Aslı Yörüsün, and Alp Yurtsever. Thank you all! I would like to thank the Swiss bureaucracy for making Can Yıldızlı visit Lausanne once every month so that I could have impromptu dinners and drinks with him to blow off steam. Thank you, Can, for cheering me up with your visits.

Lefteris Kokoris-Kogias and Kirill Nikitin were not just my colleagues but also good friends. Lefteris was my shadow advisor in the early years of my PhD and his guidance and support were crucial to this dissertation. One of my favorite moments from the past six years is when Lefteris showed up to the lab around 10pm with a bottle of tsipouro and some snacks to keep me company as I worked on the experiments of our Calypso submission. Thank you, Lefteris, for your advising and friendship. I am looking forward to our future encounters, which I am sure will happen either on ski slopes or at a Mediterranean city. It was a pleasure to work with Kirill for two main reasons: First, I could always trust him to do whatever is necessary to get the job done. Second, we managed to find a way to have fun no matter how boring/challenging/stressful/pointless the job at hand was. Outside of the lab, I had so many beautiful memories and experiences with Kirill, both in Switzerland and other countries, thanks to our mutual interest in numerous research-unrelated subjects. Thank you, *Monsieur*, for everything - these past six years would have sucked without you. It would be a blasphemy

if I did not mention David Froelicher and João Sá Sousa together. Becoming friends with these fine gentlemen is reason enough for me to feel glad that I did my PhD at EPFL. João, you are a great friend that I can always rely on. Thank you for cooking me delicious Portuguese meals, teaching me how to ski, introducing me to Renens Basket, explaining me how the Portuguese Bacalhau actually comes from Norway, and many other memories. I cannot wait for you to complete your transition to become a full-on Turkish person. David, *mon petit Malakoff*, you are the best cycling buddy that I could ask for. Thank you for the great rides, teaching me the local cycling routes, introducing me to the wonders of Malakoff, and being a constant source of fun. Now that my PhD is over, my next big goal in life is to do the Tour des Stations with you. The friendship of Okan Altıngövde has been a source of support and joy during this long and difficult journey. Okan, I cannot thank you enough for the mind-expanding conversations, cheering me up with your unique sense of humor, making sure that I do not miss out on the must-see movies and TV shows, the regular visits to Jetée, Tang Roulou, and Shanti, introducing me to rock and metal music, driving me crazy with your way too laid-back, nonchalant, and chaotic-neutral personality, and giving me a reason for visiting Ankara. I truly believe that you are the best thing that has come out of Central Anatolia.

The wise Dwight K. Schrute has once said: *Nostalgia is truly one of the great human weaknesses. Second only to the neck.* I was blessed to have parts of my Songs of Hope family close to me in Madison so that I could periodically succumb to my second greatest (and by far the favorite) weakness. Thank you, Rafael Ortega Castrellon and Nick Haley, for making me a part of your Thanksgiving celebrations. Thank you, Luis Ortega Nieves and David Haley & Becky Montgomery, for opening your houses to me for the said celebrations (and many other times in the past 20 years). Nick's pit stops in Madison on his way to St. Paul enabled us to catch up over lunches and dinners, which I always looked forward to. Jeanne Junge and Tom Surprenant are the reason why St. Paul is my home away from home, and the reason I have so many great people in my life, some of whom are mentioned in this text. For that, I am forever grateful to Jeanne and Tom.

Hakan Memişoğlu was my flatmate in Madison and was a source of support as I was adjusting to my new life and the grad school in Madison. Emre Besler is a great friend with whom I have been in a long-distance friendship for a while now. Having Emre within a Van Galder Bus distance during my time in Madison was a blessing. I fondly remember our trips, meet-ups in Madison and Chicago, and long conversations on various subjects where I always learned something new from you. Good job to both of us for doing our best to be there for each other during the difficult times of our PhDs. I am looking forward to our new adventures!

I cannot put into words how lucky I am to have Ali Uğur Aker, Mine Avcı, Buğrahan Ayanoğlu, Eralp Bayraktar, Gizem Burhanoğlu, Osman Orkan Durmaz, Ece Küçük, and Behçet Meyveci in my life. Thank you for always being there for me and being amazing friends for many years. I love you all! Zeynep Ece Atabay, you are one of the most selfless and caring people I know. I am sure that the last years of my PhD would have been much harder without your continuous support. Thank you for always being there for me and educating me on architectural elements. I cannot wait to see you become the doctor of chaos, destruction, and suffering. Emma Lofgren is a great friend with whom I can effortlessly talk about anything. Thank you for patiently

listening to my problems about my PhD and comforting me whenever I needed it during our somewhat-regular WhatsApp calls. Nigel Lofgren is a "special" person and a dear friend that I am lucky to have in my life. I cannot stress enough how valuable it was to have you close by during my first two years in Lausanne. Our meet-ups in Modena and Lausanne are some of the highlights of my PhD journey and our St. Patrick's weekend in Turin and Bologna is easily one of my best memories. Makwak Nigel! Ahmet Can Kan, *my twin*, thank you for always being by my side for the past 24 years. I cannot imagine my life without you.

Finally, I would not have been where I am today if it were not for my family's unconditional love and support. I am deeply grateful to my parents, Işın and Nedim, for doing everything in their power to make sure that I received a good education, being there for me when I need advice, and always believing in me no matter what happens; my grandpa, Yahya, for being my role model as an engineer and a human being; my grandma, Özgül, for raising me and being my first teacher; Gül and Elif, for not only being the coolest aunts but also being multifunctional and serving as moms/big sisters when necessary; my cousin, Zeynep, for being an amazing little sister; my uncle, Mehmet, for providing moments of comic relief; and my cat, Petit, for being the joy of my life. You are the bestest family ever and I love you all! I dedicate this thesis to you, and especially to your memory Özgülcüğüm.

Lausanne, October 3, 2023 -cey

Abstract

Smart contracts have emerged as the most promising foundations for applications of the blockchain technology. Even though smart contracts are expected to serve as the backbone of the next-generation web, they have several limitations that hinder their widespread adoption, namely limited computational functionality, restricted programmability, and lack of data confidentiality. Moreover, addressing these challenges manually in application-specific ways requires a lot of developer effort and time due to the monolithic architecture of smart contracts. In this dissertation, we start over with a novel architecture for building and deploying general-purpose decentralized programs. To this end, we first propose a new architecture that replaces the monolithic execution model of smart contracts with a modular one to support a rich set of functionality, which can be easily and permissionlessly extended at any time. Second, to support the efficient deterministic execution required by computationally-advanced smart contracts, we build a deterministic sandbox with floating-point arithmetic support that brings safe and deterministic execution together with general-purpose programming without having to sacrifice performance. Finally, we combine threshold cryptography and the blockchain technology to build a framework that enables mutually distrustful parties to share their confidential data in a fully auditable, transparent and decentralized manner. Through prototyping and evaluation using real-world applications, we demonstrate that it is possible and feasibly-practical to build a decentralized computing platform that can support general-purpose computations.
Keywords: decentralized computing, smart contracts, deterministic execution, confidentiality, post-quantum, data-flow graphs, extensibility, modular, blockchain, data sharing.

Résumé

Les contrats intelligents sont apparus comme les bases les plus prometteuses pour les applications de la technologie blockchain. Même si les contrats intelligents sont censés servir de l'épine dorsale du web de nouvelle génération, ils présentent plusieurs limites qui qui empêchent leur adoption à grande échelle, à savoir une fonctionnalité informatique limitée, une programmabilité restreinte et un manque de confidentialité des données, la programmabilité restreinte et le manque de confidentialité des données. En outre, De plus, relever ces défis manuellement en fonction de l'application nécessite beaucoup d'efforts et de temps de la part des développeurs en raison de la complexité de la programmation. de l'application nécessite beaucoup d'efforts et de temps de la part des développeurs en raison de l'architecture monolithique des contrats intelligents. l'architecture monolithique des contrats intelligents. Dans cette thèse, nous commençons par avec une nouvelle architecture pour construire et déployer des programmes décentralisés d'usage général. programmes décentralisés à usage général. À cette fin, nous proposons d'abord une nouvelle architecture qui remplace le modèle d'exécution monolithique des contrats intelligents par un modèle modulaire modulaire pour prendre en charge un ensemble riche de fonctionnalités, qui peuvent être facilement facilement et sans permission à tout moment. Deuxièmement, pour soutenir l'exécution déterministe efficace déterministe efficace requise par les contrats intelligents avancés sur le plan informatique, nous construisons un bac à sable déterministe avec support de l'arithmétique à virgule flottante qui apporte une l'exécution sûre et déterministe avec la programmation polyvalente sans avoir à sacrifier la performance. Enfin, nous combinons la cryptographie et la technologie blockchain pour construire un cadre qui permet aux parties qui se mutuellement méfiantes de partager leurs données confidentielles d'une manière totalement auditable, transparente et décentralisée, transparente et décentralisée. Grâce au prototypage et à l'évaluation d'applications réelles, nous démontrons qu'il est possible et faisable de construire une plateforme informatique décentralisée capable de prendre en charge des les calculs à usage général.

Mots-clés: informatique décentralisée, contrats intelligents, exécution déterministe, confidentialité, post-quantique, graphes de flux de données, extensibilité, modulaire, blockchain, partage de données.

Contents

Acknowledgments i

Abstract (English/Français) v

1 Introduction 1

1.1 Thesis roadmap . . . 3

2 PROTEAN: A Modular and Extensible Framework for General-Purpose Decentralized Computing 7

2.1 Introduction . . . 7

2.2 Motivating applications . . . 9

2.2.1 Decentralized lotteries . . . 9

2.2.2 Electronic voting (E-voting) . . . 10

2.3 PROTEAN Architecture . . . 11

2.3.1 System goals . . . 11

2.3.2 Key Concepts . . . 12

2.3.3 System Overview . . . 13

2.3.4 PROTEAN's programming model . . . 17

2.3.5 State management . . . 21

2.3.6 Executing workflows . . . 23

2.3.7 DFU management . . . 28

2.4 Security discussion . . . 29

2.5 Implementation . . . 31

2.5.1 Cryptographic primitives . . . 31

2.5.2 DFU implementations . . . 32

2.5.3 Applications . . . 34

2.6 Evaluation . . . 36

2.6.1 Experimental setup . . . 37

2.6.2 Microbenchmarks . . . 37

2.6.3 Modified Neff shuffle and threshold decryption protocols . . . 39

2.6.4 Application experiments . . . 40

2.7 Related work . . . 43

2.8 Conclusion . . . 44

3 DEJAFLOAT: A Deterministic JVM Sandbox with Floating-point Arithmetic Support 51
3.1 Introduction . . . 51
3.2 Background and motivation . . . 53
3.2.1 The IEEE 754 standard . . . 53
3.2.2 Why is floating-point arithmetic non-deterministic? . . . 54
3.2.3 Motivating applications . . . 56
3.3 Overview . . . 58
3.3.1 Challenges . . . 58
3.3.2 System goals . . . 59
3.3.3 Design rationale: Java and JVM . . . 59
3.4 DEJAFLOAT Architecture . . . 60
3.4.1 A static checker for language-level determinism . . . 60
3.4.2 Deterministic floating-point arithmetic . . . 62
3.4.3 Limiting program execution costs . . . 66
3.5 Implementation . . . 67
3.6 Evaluation . . . 67
3.6.1 Experimental setup . . . 67
3.6.2 Microbenchmarks . . . 68
3.6.3 Machine learning algorithms with deterministic transcendental functions 71
3.7 Related work . . . 74
3.8 Conclusion . . . 75

4 Decentralized Private-Data Sharing with On-chain Secrets 77
4.1 Data confidentiality in blockchains . . . 77
4.2 Motivating applications . . . 79
4.3 System overview . . . 80
4.3.1 Challenges . . . 80
4.3.2 System goals . . . 81
4.3.3 System model . . . 81
4.3.4 Threat model . . . 82
4.3.5 Architecture overview . . . 82
4.4 One-time secrets (OTS) . . . 83
4.4.1 OTS subprotocols . . . 84
4.4.2 Advantages and shortcomings . . . 87
4.4.3 Revisiting the system goals . . . 87
4.5 Post-quantum one-time secrets (PQ-OTS) . . . 88
4.5.1 PQ-OTS subprotocols . . . 90
4.5.2 Discussion . . . 91
4.6 Implementation . . . 91
4.7 Evaluation . . . 92
4.7.1 Experimental setup . . . 92
4.7.2 Level of decentralization vs. performance . . . 93

4.7.3 On-chain secrets vs. a semi-centralized system 94
4.7.4 A decentralized lottery application with on-chain secrets 97
4.8 Related work 98
4.9 Conclusion 99

5 Conclusion 101

A Workflows and FSM models of the PROTEAN applications 103
A.1 `randlotto` 103
A.1.1 FSM model 103
A.1.2 Workflows 103
A.2 `secretlotto` 106
A.2.1 FSM model 106
A.2.2 Workflows 107
A.3 `e-voting` 111
A.3.1 FSM model 111
A.3.2 Workflows 111

Bibliography 119

Curriculum Vitae 135

List of Figures

2.1 Architectural diagram of PROTEAN vs. Ethereum 11
2.2 JSON schema for the workflow definition 17
2.3 Visual representation of an example workflow 19
2.4 JSON schema for the FSM model 20
2.5 PROTEAN execution plan 24
2.6 PROTEAN receipt 25
2.7 JSON schema for DFU 29
2.8 Latency of verifying `KEYVALUE` and `OPCODE` dependencies 38
2.9 Latency of generating output receipts 39
2.10 Execution latencies for the Neff shuffle and threshold decryption protocols using their standard and modified implementations 40
2.11 Execution times of `randlotto` transactions 41
2.12 Execution times of `secretlotto` transactions 42
2.13 Execution times of `e-voting` transactions 46
2.14 Execution times of `randlotto` transactions with batching 47
2.15 Execution times of `secretlotto` transactions with batching 48
2.16 Execution times of `e-voting` transactions with batching 49

3.1 Execution times of the Java matrix multiplication program with and without NaN canonicalization 68
3.2 Execution times of the C matrix multiplication with and without NaN canonicalization in WASM 69
3.3 Relative slowdown in matrix multiplication (NaN rate = 0%) due to NaN canonicalization in Java and WASM. 69

4.1 The high-level architecture of on-chain secrets 83
4.2 One-time secrets subprotocols: (1) Write, (2) Read, (3) Share retrieval, (4) Secret reconstruction 85
4.3 Post-quantum one-time secrets subprotocols: (1) Write, (2) Read, (3) Share retrieval, (4) Secret reconstruction 89
4.4 Latency of on-chain secrets protocols for varying committee sizes 93
4.5 Average write and read workflow latencies under different loads in clearance-enforcing document sharing 95

4.6 Average write and read workflow latencies for the replayed real-world data trace in clearance-enforcing document sharing 96
4.7 Lottery execution times using the Fire Lotto workload 97

List of Tables

3.1 Execution times (ns) of various transcendental functions 70
3.2 Total time to run user-based collaborative filtering with 5-fold cross-validation using the MovieLens 100K dataset 71
3.3 Neural network model training and testing with MNIST 71
3.4 Neural network model training and testing with CIFAR-10 72
3.5 Execution times of a simple Java program with instrumentation for counting the number of executed JVM bytecode instructions 74

1 Introduction

A decentralized system is a type of system in which there is no single central entity that has complete control over the entire system. Although decentralized systems have been studied extensively over the past decades [35], [59], [171], [187], [223], their prominence has surged considerably in the recent years with the emergence of the blockchain technology [159]. Blockchain is a distributed ledger of transactions that is replicated across an open (permissionless) network of nodes where anyone can participate. Each transaction represents an action that creates a change in the ledger state upon its execution. Nodes run a distributed consensus protocol to validate transactions and group them into blocks in a mutually agreed-upon order. Each node executes the transactions in a block in this order on its own copy of the ledger to update its ledger state accordingly. This ensures that the copies of the ledger are kept consistent across the network. Since all the transactions and block contents of a blockchain are publicly visible, blockchains support features such as transparency and auditability, which are desirable in a trustless setting. Furthermore, since the blocks are chained to each other via cryptographic hashes, the state of the ledger is resistant to tampering.

One of the most promising applications of the blockchain technology is smart contracts [192]. A smart contract is a user-defined computer program that runs on top of a blockchain[1]. Since the program state is stored on the blockchain, each node in the network independently executes the program code to make sure that the copies of the ledger are kept consistent. Through the characteristics and the execution model of the underlying blockchain, smart contracts create a permissionless platform for automated, transparent, and correct execution of arbitrary programs with strong integrity guarantees without relying on a single trusted entity.

The transformative promise of smart contracts is best showcased by the role that they are expected to play in the evolution of the internet: serving as the backbone for Web 3.0 [121], [215], the third generation of the World Wide Web. The core principle behind Web 3.0 is to create a decentralized web where individual users control and operate the services that

[1]Throughout this thesis, we use the term *smart contracts* to refer to blockchain-based smart contracts.

they use instead of a few large technology companies. Anyone can access and participate in the services without requiring permission from an authorized party and without having to trust a single entity. Therefore, smart contracts fit perfectly with the Web 3.0 vision as they enable *permissionless innovation* in a decentralized setting: users have the freedom to build and deploy their general-purpose decentralized applications via smart contracts without restriction or censorship by central authorities.

Unfortunately, smart contracts have several limitations at the architecture level that need to be addressed before they can achieve widespread adoption and deliver their promise. In this thesis, we are going to specifically focus on the following three limitations:

Limitation #1: *Limited computing functionality*

The replicated execution model of smart contract systems such as Ethereum [217], which is the largest and most popular smart contract platform, requires every node in the network to store and execute every smart contract. This execution model has two major consequences in terms of the functional capabilities of smart contracts. First, smart contracts cannot support computations that have non-deterministic operations or use external (off-chain) data, which might be non-deterministic. The determinism requirement is essential to ensuring that nodes obtain the same outcome when they execute the same contract code on the same inputs. Second, since smart contracts are open to everyone, and both the program state and code are publicly visible, smart contracts cannot safely operate with/on private data.

Limitation #2: *Restricted programmability*

In order to ensure the correctness and safety of the smart contracts, smart contract platforms need to be able to execute arbitrary and potentially malicious or faulty programs across diverse hardware and software stacks in a deterministic and secure manner. Ethereum achieves this by using a deterministic sandbox called Ethereum Virtual Machine (EVM) [217], which is a specialized virtual machine that only supports a restricted instruction set. Additionally, Ethereum requires smart contracts to be written in domain-specific languages, such as Solidity [107] and Vyper [207], to further enforce deterministic and safe execution.

Unfortunately EVM and its programming languages considerably restrict the programmability of smart contracts. The limited instruction set and custom design of EVM prevent smart contracts from using common operations that are available in general-purpose programming languages [69], [70] and performing basic cryptographic computations, which are desirable in the context of smart contracts, in an efficient way [68], [71]. Moreover, the lack of programming tools and unintuitive semantics of the domain-specific languages increase the difficulty of programming smart contracts and lead to programming errors [8], [200].

The lack of floating-point arithmetic support is one of the representative examples of the restricted programmability of Ethereum smart contracts. Since the IEEE standard for floating-

point arithmetic does not guarantee that the same floating-point operation will produce the same result on all conforming systems, it is difficult to make floating-point arithmetic fully deterministic. As a result, existing smart contract platforms either omit floating-point arithmetic, and instead support fixed-point arithmetic, or only support a small subset of floating-point operations.

Limitation #3: *Data confidentiality*

In today's web, users' personal data are controlled by centralized companies and platforms, which can monetize or use personal data without users' consent [32], [63] or fail to keep the data safe [93]. One of the main promises of Web 3.0 [121], [215] is to achieve data sovereignty by giving users the control of their data. The transparency, integrity, and decentralized trust guarantees of smart contracts make them a good candidate for building decentralized data storage and sharing platforms. However, since smart contracts build on public blockchains, they cannot guarantee the confidentiality of sensitive personal data. Furthermore, the public nature of smart contracts necessitates protecting the long-term confidentiality of sensitive on-chain data *now* from the quantum threats of the *future*.

What makes the matters worse is that, even if there are solutions to some of the above limitations, making changes to smart contract platforms requires a slow and difficult governance process called *hard fork*. A hard fork is an upgrade to the underlying protocol software with the goal of introducing new functionalities, making performance improvements, or fixing security issues. The problem with hard forks is that they require all nodes in the smart contract network to agree on the changes and to upgrade to the new version of the protocol software. As a result, hard forks introduce significant delays to making system upgrades, hence hindering the evolution smart contracts.

We believe that applying the permissionless innovation principle at the architecture level can be crucial for the rapid advancement of smart contracts. Based on this idea, we ask the following question:

Is it possible to support permissionless innovation at the architecture level, namely permissionless extensibility, to enable easy, modular and unrestricted addition of new classes of core functionality to the system without requiring forklift upgrades?

1.1 Thesis roadmap

In this thesis, we affirmatively answer this question by designing and prototyping a general-purpose decentralized computing platform, which addresses the limitations of smart contracts that we have identified above. More specifically, the contributions of this thesis are the following:

PROTEAN: A Modular and Extensible Framework for General-Purpose Decentralized Com-

puting

In Chapter 2, we introduce PROTEAN, a permissionlessly-extensible decentralized computing platform that enables easy and modular addition of different classes of specialized functionality without forklifting the whole system. We argue that the root cause of the limitations of current smart contract systems is their monolithic execution model where consensus and computing are tightly coupled. Based on this observation, PROTEAN builds on two key ideas: (1) creating a modular architecture where consensus and computing are separated from each other, and (2) providing distinct specialized functionality via decentralized functional units (DFUs), which are standalone distributed systems that operate in isolation from each other. Since PROTEAN adopts a DFU-centric execution model that requires coordinating multiple distributed systems as part of executing a smart contract, we propose a workflow-based programming model that enables developers to define the data and execution dependencies between DFUs. We build a prototype of PROTEAN and implement real-world decentralized applications on top of it to show that PROTEAN can support applications that are not feasible or possible to implement on existing smart contract systems.

DEJAFLOAT: A Deterministic JVM Sandbox with Floating-point Arithmetic Support

In Chapter 3, we introduce DEJAFLOAT, a deterministic Java Virtual Machine (JVM) based sandbox that can efficiently provide smart contracts with deterministic floating-point arithmetic. By combining deterministic and safe execution with a general-purpose programming language, DEJAFLOAT aims at increasing the programmability of smart contracts, and many other applications that can benefit from reproducible general-purpose computations. DEJAFLOAT combines Java- and JVM-level techniques to eliminate sources of non-determinism in floating-point arithmetic with minimum performance overhead. We build a prototype of DEJAFLOAT and evaluate its performance using two computationally-intensive machine learning applications.

Decentralized Private-Data Sharing with On-chain Secrets

In Chapter 4, we address the problem of storing confidential data on blockchains. By combining threshold cryptography with the blockchain technology, we build on-chain secrets: a framework that enables mutually distrustful parties to share their confidential data in an auditable and transparent way. We design and implement two on-chain secrets protocols: one-time secrets and post-quantum one-time secrets that can protect the confidentiality of secret data against today's adversaries and the quantum adversaries of the future, respectively.

Bibliographic notes

This thesis is based on parts of the following co-authored publications:

- E. C. Alp, E. Kokoris-Kogias, G. Fragkouli, and B. Ford, Rethinking General-Purpose Decentralized Computing, in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019, pp. 105–112 [3].
- E. C. Alp, C. Băsescu, P. N. Tennage, N. Kocher, G. Bosson, and B. A. Ford, Efficient Deterministic Execution of Smart Contracts, Tech. Rep., 2022 [2].
- E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, CALYPSO: Private Data Management for Decentralized Ledgers, *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 586–599, 2020 [133].

2 PROTEAN: A Modular and Extensible Framework for General-Purpose Decentralized Computing

2.1 Introduction

In the previous section, we have introduced the replicated execution model of Ethereum-style smart contract platforms where every node in the system individually executes every smart contract in the system. This execution model ensures that the whole network can agree on the correct execution of the contracts and maintain a consistent view of the contract states. However, this requires smart contracts to be deterministic so that every node in the network, who potentially run on different software/hardware stacks, to get the same result when they execute the same smart contract with the same inputs. Unfortunately, this requirement severely restricts the functionality that can be supported by the existing smart contract platforms.

First of all, current smart contracts cannot support computations with non-deterministic data and operations. As a result, smart contracts cannot support programming constructs and computations that are commonly used in general-purpose execution environments, such as threads and other parallel programming constructs, random number generators, any time-based computation or initializing variables based on the time, and calling an external API that runs outside of the blockchain.

To enforce deterministic execution of smart contracts, current smart contract platforms typically rely on domain-specific languages and specialized execution VMs. Even though these programming environments prevent malicious or buggy programs to cause non-deterministic executions, they also restrict the programmability of smart contracts as they have limited and hard to extend instruction sets. These execution environments often sacrifice performance for deterministic execution and therefore perform poorly. Moreover, programming with domain-specific languages brings additional burden to programmers due to the additional effort of learning a new language and the the lack of libraries and development tools.

Finally, enforcing deterministic execution of smart contracts requires all nodes in the network

to operate on the same data. Due to this requirement, smart contracts cannot execute on external data from the real world. Since external data is off-chain, different parts of the network can potentially see different data, which would violate deterministic execution. For the same reason, smart contracts cannot safely operate with/on secret data (*e.g.*, cryptographic computations that rely on secrets). Since the whole network would need to have access to the secret data to be able to perform the same computation, it would violate the confidentiality of the secrets.

Based on the above discussion, we come to the conclusion that current smart contract platforms like Ethereum have limitations due to their monolithic architectures: Every node is part of a single network where consensus and computing are tightly coupled. This tight coupling does not only limit the computations that can be performed in smart contracts, but also creates an inflexible system where innovation and development cycles are slow.

We present PROTEAN, a modularly-extensible architecture for general-purpose decentralized computing. PROTEAN supports permissionless extensibility via *decentralized functional units (DFUs)*, which are standalone, special-purpose distributed systems that operate independently from each other. Each DFU supports a unique specialized functionality (*e.g.*, consensus, arbitrary code execution, randomness generation, verifiable anonymity) that is performed collectively by its nodes. DFUs can join PROTEAN and offer their services to applications without any restrictions and disrupting the operation of the other DFUs. Each DFU exposes a set of *opcodes*, which represent the specialized computations that can be performed by the DFU nodes. Developers represent the programmable logic of their smart contracts by creating a data-flow graph [1], [54], [122], [157] with the opcodes. In this programming model, clients can execute a contract by simply following the data-flow graph to send requests to DFUs to execute the opcodes.

One of the major benefits of the DFU-centric execution model of PROTEAN is that it enables a large set of computations to be used in decentralized applications. In the DFU-centric execution model, a computation (opcode) is only performed by the nodes of the DFU that supports it. Once the DFU nodes reach an agreement on the result of the computation, they generate a publicly-verifiable cryptographic proof that can be used by the other nodes to verify the correctness of a computation without having to re-perform it. As not all DFUs need to use replicated execution, but can instead use different techniques such as zero-knowledge proofs and secure multi-party computation to ensure correctness with no single point of failure, PROTEAN can support classes of computations that are currently not supported by smart contracts: non-deterministic computations, complex cryptographic computations that use secret data, and computations with confidential data.

Contributions. We make the following contributions:

- We present PROTEAN, a modular architecture for general-purpose decentralized computing that enables a richer set of computations and a larger class of decentralized

applications and achieves high extensibility via *permissionless upgradability* that enables adding new functionalities without forklifting the whole system.

- We define a programming model that enables developers to define the programmable logic of their contract in our DFU-centric execution model. Our programming model combines ideas from data-flow graphs and finite-state machine (FSM) models.
- We present an implementation of PROTEAN and build three decentralized applications on top of it to demonstrate the range of complex applications that it can support.
- We evaluate the performance of our system and show that PROTEAN can support a richer set of applications compared to the Ethereum-style smart contracts without incurring large performance penalties.

2.2 Motivating applications

There is a broad range of applications, such as automated market makers, auctions, games, insurance, non-fungible tokens, and voting, that have emerged as popular use cases of smart contracts. In this chapter, we take a closer look at two classes of decentralized applications, namely decentralized lotteries [49], [149] and electronic voting (e-voting) [147] to get a better sense of the shortcomings of Ethereum and how PROTEAN addresses them. We choose these applications as they embody a good mix of functional and performance demands to drive the development of our architecture and ensure its flexibility.

2.2.1 Decentralized lotteries

A decentralized lottery is a decentralized application that uses a smart contract to ensure the correctness, fairness, and transparency of the lottery. The smart contract automatically and transparently executes every step of the lottery process, which involves collecting funds from the participants, choosing the winner, and transferring the funds to the winner, without relying on a trusted intermediary.

There are typically two stages in a decentralized lottery. In the first stage, users participate in the lottery by depositing their funds to the lottery smart contract and purchasing lottery tickets. The contract keeps track of the participants and holds the a pool of the collected funds. Once the ticket acquisition period is over, the smart contract typically parses a source of public randomness, uses the randomness to select the lottery winner, and transfers the collected funds 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, contract developers create their own pseudorandom number generator implementations. A typical approach for generating randomness in Ethereum is parsing the randomness from a future block (*e.g.*, its timestamp, nonce or block hash).

However, this approach is vulnerable to attacks where a malicious miner can manipulate the mining process to bias the values of the block variables, and therefore the randomness [8], [168], [180].

Another approach for generating randomness in smart contracts is using a *commit-reveal* protocol. In this approach, each user chooses a secret and shares their commitment to the secret with other users by storing it to a smart contract. Later, users reveal their secrets and a random value is calculated by combining the secrets. However, this approach is susceptible to bias-via-abort attacks where a user can choose not to reveal their secret to bias the randomness generation to their advantage [190]. Solutions that address the bias-via-abort attacks either require participants to lock collaterals [5] or take multiple rounds to finish the protocol [149].

Finally, oracles can be used for generating random values off-chain and sending it to the on-chain contract. Previous oracle solutions required trusting a third party service for the quality of the randomness. Chainlink's Verifiable Random Function (VRF) [120] has recently emerged as a viable solution for providing Ethereum smart contracts with provably fair and verifiable random numbers without relying on a single trusted entity.

Although there are secure multi-party computation (SMPC) protocols for generating fair and verifiable randomness [31], [91], [190], 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 §2.5.3, we present two decentralized lottery applications. The first lottery uses an SMPC-based randomness beacon that produces publicly-verifiable, unpredictable, and unbiasable random numbers. The second lottery relies on threshold cryptography to implement a commit-reveal protocol that does not require collaterals or multiple rounds to finish.

2.2.2 Electronic voting (E-voting)

Remote electronic voting (e-voting) has become increasingly popular in the recent years as it has been used as an alternative to the traditional voting schemes in various countries [87], [189], [202]. E-voting has the potential to make the voting process more convenient for voters, which can increase voter turnout, and less costly for the organizers (*e.g.*, governments). Additionally, it is expected to deliver more reliable results by reducing human errors and preventing fraud in ballot stations. To reach its potential and see wider deployment, e-voting systems have to provide strict security guarantees about the voting process, such as secrecy and integrity of the votes, unlinkability of voter and vote, and verifiability and transparency of the voting process. Moreover, the voting system should not be controlled by a single entity to avoid a single point of failure and compromise.

Smart contracts can potentially serve as a platform for building decentralized e-voting systems that provide the above guarantees. McCorry et al. provide the first implementation of a decentralized and self-tallying e-voting protocol that runs on Ethereum [147]. Their system is

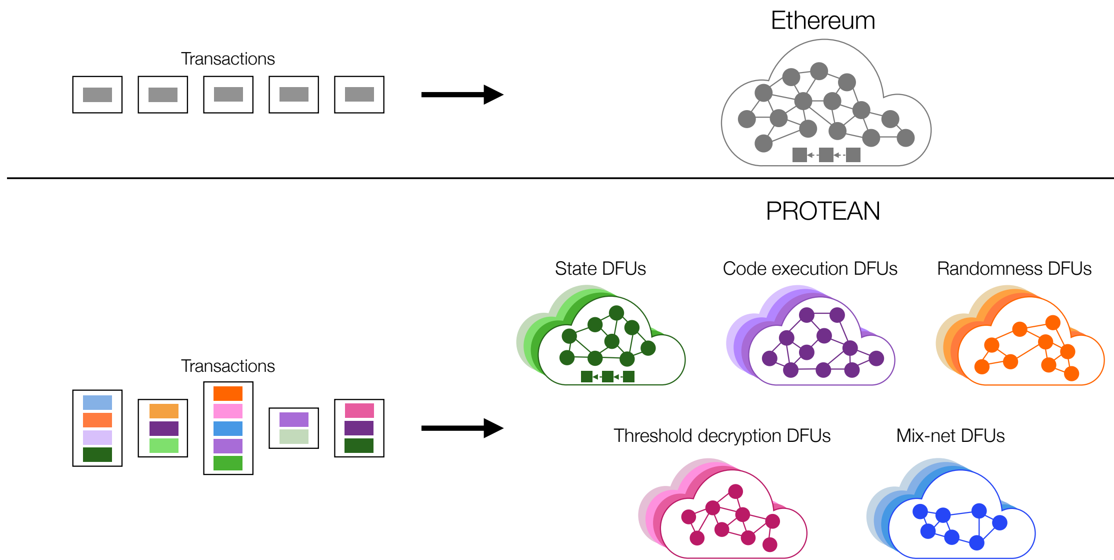


Figure 2.1: Architectural diagram of PROTEAN vs. Ethereum

based on the Open Vote Network protocol and provide desirable properties, such as maximum voter privacy and public verifiability. The downside of their system is that it can only scale up to 50 – 60 voters due to the limitations of Ethereum. A follow-up work [175] addresses the scalability problem of [147] by using an off-chain party to tally the votes. However, this solution has reduced security guarantees as it requires active participation of users to make sure that the off-chain party has not violated the correctness of the tallying process. Finally, Provotum [132] combines well-known cryptographic protocols, such as distributed key generation and homomorphic encryption, to implement a blockchain-based e-voting system with end-to-end verifiability. A prototype of Provotum is implemented on Ethereum but the research paper lacks performance measurements. Therefore, it is difficult to assess how Provotum compares to other work in terms of scalability. In §2.5.3, we present an e-voting application that combines threshold cryptography with verifiable shuffles to provide the above security guarantees and scale up to 1000 voters.

2.3 PROTEAN Architecture

Figure 2.1 shows the architecture of PROTEAN and how it compares to Ethereum.

2.3.1 System goals

PROTEAN has the following primary goals:

1. *Permissionless extensibility*: New classes of functionalities can be added modularly

without any restrictions or requiring major upgrades to the system.
2. *General-purpose computing*: Users can implement and execute arbitrary smart contracts that use the available functionalities in the system.
3. *Decentralization*: There is no single point of failure or compromise in the system.

2.3.2 Key Concepts

We describe the three key concepts of PROTEAN's architecture:

Extensible functionalities via DFUs: PROTEAN is an ecosystem of decentralized functional units (DFUs), each of which is a standalone and special-purpose distributed system that supports a specific class of functionality (*e.g.*, threshold cryptography). DFUs operate independently from each other and have their own mechanisms for node membership management and governance processes. A DFU can join PROTEAN in a permissionless manner and start providing its service to smart contracts without interfering with the rest of the system. There can be many DFUs that support the same class of functionality using different underlying protocols (or implementations of the same protocol) with different performance and security guarantees. Additionally, DFUs can upgrade the underlying protocol to newer versions and support multiple versions concurrently without sacrificing backward compatibility. As a result, PROTEAN achieves *permissionless extensibility* where new classes of functionalities can be introduced to the system in an easy and modular way by anyone.

Separation of consensus and computing: Ethereum employs a *homogeneous execution model* where every node in the network performs the same computational task: reaching consensus on the order of the transactions and executing them in that order to maintain a consistent blockchain state. Due to this execution model, every node has to execute every smart contract in Ethereum, which limits the classes of functionality that can be supported by smart contracts. PROTEAN separates consensus from computing by employing a *heterogeneous execution model* where each DFU executes only the computational tasks of their specialized functionality in isolation from the rest of the network. *Computing DFUs* perform their computations on given external inputs (*e.g.*, contract state, outputs of other computing DFUs, user-owned data) to produce state changes for the contracts. *State DFUs* are responsible for maintaining the contract states by reaching consensus on the correctness of the state changes and committing them. State DFUs use cryptographic attestations that are produced by the computing DFUs to verify the correctness of a state change without having to re-execute the computations that produced it. It is the contract developer's responsibility to choose the computing DFUs and the state DFU that will be used by the smart contract based on the requirements of the contract and the characteristics of the DFUs.

Thanks to this separation, it is possible to introduce new functionality (or upgrade the existing) without disrupting the operation of the system. For example, if there is a new high-

performance consensus algorithm, a set of nodes can create a state DFU and deploy it alongside the existing state DFUs. Similarly, if, for example, there is a new code execution VM that is more efficient than the existing VMs, it can be deployed in the PROTEAN as a new computing DFU. This can keep happening for any specialized functionality, without different DFUs disrupting each other's operation or breaking anything.

Client-driven contract execution: The heterogeneous execution model of PROTEAN has two consequences: First, because computing DFUs do not maintain contract state, they need to retrieve the most recent contract state from the state DFU to be able to correctly execute their computations. Second, as executing a smart contract involves multiple DFUs, the computation of a DFU might depend on the output of another DFU's computation. Therefore, executing a smart contract requires coordination among different DFUs. PROTEAN relies on clients for orchestrating the execution of contracts by sending requests to DFUs to invoke their computations. As clients provide DFUs with the necessary input data, DFUs can simply return the results of their computations to clients without having to communicate with other DFUs. In a way, clients serve as the communication conduit between the DFUs. An alternative design is to have the DFUs orchestrate the execution of contracts. In this design, DFUs would have to understand the steps of a contract and communicate with each other to be able to correctly execute the contract. We choose the client-driven execution model over the DFU-driven model for two reasons: First, it simplifies the design of DFUs as they do not need to be aware of or keep state about each other. Second, it fits well with our motivating applications (*i.e.*, decentralized lottery, e-voting) that require clients to interact with the smart contracts.

2.3.3 System Overview

A PROTEAN smart contract is a combination of programmable logic and state. The programmable logic of a smart contract comprises three components: *workflows* that define the steps and rules for executing the contract, a *finite state machine (FSM) model* of the contract, and the *contract code* that is written in a general-purpose programming language.

A *workflow* describes a series of tasks that are executed together as part of a specific, self-contained process (*e.g.*, vote tallying in the e-voting application). Each task of the workflow is represented by a *transaction* that causes a change in the contract state upon its execution. A transaction comprises a series of operations called *opcodes*, each of which corresponds to a specialized computation of a DFU. We are not using the term opcode to mean the same thing that it would in a traditional instruction set architecture, but rather an analogous but different thing in the decentralized architecture of PROTEAN. Opcodes serve as a well-defined API through which a DFU exposes the computations of its specialized functionality to smart contracts. Contract developers can make use of a functionality in their smart contracts by simply including the corresponding opcodes in the transactions. From the developer's perspective, using an opcode is analogous to using a third-party software service as it acts as a black box and hides the complexities of its computations. Since there can be many

DFUs that support the same functionality with different underlying implementations and security/performance guarantees, it is the developer's responsibility to choose the appropriate opcode-DFU combinations based on the requirements of their smart contract.

Transactions are represented as data-flow graphs where the nodes are opcodes and the directed edges are the data dependencies between opcodes. For example, the edge $op_i \rightarrow op_j$ indicates that the output of the *parent* opcode op_i is used as an input to the *child* opcode op_j. Therefore, the data-flow graph also serves as an execution dependency graph as it imposes the order in which the opcodes have to be executed: an opcode can only be executed if all of its parent opcodes have already been executed. The final opcode of every transaction is for committing the state changes that are created by the execution of the transaction to the state DFU that stores the contract state. To execute an opcode, a client simply sends a request with the necessary inputs to the DFU that exposes the opcode. The DFU nodes collectively execute the opcode and produce a *receipt* that contains information about the execution of the opcode (*e.g.*, input and outputs). The receipt is collectively signed by the DFU nodes and serves as a cryptographic attestation for other DFUs to verify that the data dependencies are satisfied without having to re-execute the computations.

Representing smart contracts as FSMs is a common design pattern in Ethereum [45], [146]. In PROTEAN, we adopt the same approach by requiring contract developers to create an *FSM model* of their contract to represent its behavior. The FSM states represent the stages that a contract moves between during its life cycle. A transition between two FSM states is based on the current state, which transaction is executed, and the output of the transaction. Consequently, the FSM model of the contract dictates how clients can interact with the contract: a transaction can be executed if and only if it can trigger a transition from the current FSM state.

The final component of the programmable logic of a PROTEAN smart contract is the *contract code*, which is distinguished from the contract code in Ethereum smart contracts in several ways. First, the contract code is executed only by the DFUs that support the functionality of executing and agreeing on the output of arbitrary code. In other words, code execution is not replicated across the whole network as in Ethereum. Second, the contract code can be written in general-purpose programming languages (*e.g.*, Go, Java, C/C++, Rust). Finally, since computing DFUs provide contracts with general-purpose computations, we expect the contract code to mainly serve as a glue code that performs contract-specific tasks, such as managing the contract data, performing computations that are not supported by the DFUs, and formatting data to adapt the inputs and outputs of the opcodes. We note that the developers still have the freedom to implement all the computations in the contract code without using the specialized functionalities of the DFUs.

State model

The state of a contract is stored in a key-value store that builds on top of a blockchain. Each contract has a logically-separate namespace where both the data and the programmable logic of the contract are stored publicly and are tied together. A contract's state can only be modified by executing the transactions that are defined in the workflows of the contract. Every transaction ends with an opcode that commits the state changes, which is a set of key-value pairs, produced by the transaction to the contract state. This opcode is similar to a `put` operation in the traditional key-value stores. In §2.3.5, we discuss in detail how DFUs handle state in PROTEAN.

Security model

Each DFU in PROTEAN defines its own threat model and security assumptions that are consistent with its functionality and the underlying protocol implementation. Our only requirement is that the state DFUs run a Byzantine fault-tolerant consensus protocol and follow its trust assumptions. DFUs make this information publicly available when they join PROTEAN and operate in accordance with their assumptions. It is the developers' responsibility to use the publicly-available information about the DFUs to choose the DFUs that they want to deploy in their contracts, thereby creating its trust base. Therefore, we can think of using a DFU as using a third-party software in your system. To be able to tolerate a threshold number of compromised nodes within each DFU, and thereby avoid single points of compromise, we require DFUs to collectively sign [191] the output of their computations. To this end we need each DFU to define their signing threshold. Each DFU specifies a signature verification threshold t, which indicates that if a DFU's output has at least t different valid signatures on it, then the verifier can decide that the output is produced correctly. Finally, we assume malicious and/or unreliable clients and do not trust them for the correct execution of the contracts.

DFU types

Below, we list different classes of DFUs that we have prototyped in PROTEAN. We note that this is not an exhaustive list as DFUs with arbitrary functionalities can be added to the system.

- **State DFUs:** These DFUs are responsible for maintaining the contract states. Each state DFU has three components: (i) a key-value store that stores each contract's state in a separate namespace, (ii) a distributed consensus algorithm that can tolerate Byzantine failures [33], [159] to guarantee that its nodes can agree on the contract states, and (iii) a blockchain to maintain an immutable log of the state changes of the contracts. Each contract picks one of the state DFUs in the system for storing its state.
  State DFUs do not execute contract-specific code or perform general-purpose computations to compute the state changes for a contract. Instead, they verify that the states changes, which are produced by transactions, are correct and consistent to decide

whether they can be committed or not. State DFUs use the cryptographic attestations that are produced by the computing DFUs to perform the verifications without having to re-execute the computations.

- **Computing DFUs:** These DFUs are responsible for performing the computations of smart contracts. Even though these DFUs can perform computations using replicated execution as in Ethereum, PROTEAN additionally allows DFUs that can support computations using different techniques such as secure multi-party computation or specialized cryptographic protocols whose correct execution can be verified using zero-knowledge proofs. Unlike state DFUs, computing DFUs do not maintain contract states.
  - **Code execution DFUs:** These DFUs provide an execution environment for contract code that is written in a general-purpose programming language (*e.g.*, Go, Java, C/C++, Rust). The execution environment can be implemented on top of a virtual machine such as Java Virtual Machine (JVM) or WebAssembly (WASM) to support different programming languages. The execution environment should ideally enforce deterministic execution of the contract code so that the nodes can agree on the output of the execution. If the execution environment does not support deterministic execution and the contract code has non-deterministic operations, either due to an implementation bug or a malicious developer, the DFU nodes will not be able to agree on the output of the execution and the contract will not be able to make progress. However, thanks to the heterogeneous execution model of PROTEAN, the rest of the system and the other contracts will not be affected.
  - **Mix-net DFUs:** These DFUs implement a mix-net [35] using a verifiable shuffling protocol. A shuffle operation permutes and re-encrypts a given list of ciphertexts to produce a new list of ciphertexts. The new ciphertexts decrypt to the same plaintexts as the original ciphertexts but in a different order. A verifiable shuffle scheme additionally produces zero-knowledge proofs to prove without revealing any information about the permutation or the re-encryptions that the shuffle was performed correctly. Given a list of ciphertexts, a mix-net DFU performs a series of verifiable shuffles, where each shuffle is performed by one of the DFU nodes, and guarantees that the final list of ciphertexts cannot be linked to the original list of ciphertexts as long as at least one honest node have performed a shuffle.
  - **Threshold decryption DFUs:** These DFUs implement a $(t,n)-$threshold decryption scheme [24], [130], [166], [179] where n shares of a collective private key are distributed among n DFU nodes such that the private key can be recovered only if a threshold number (t) of nodes combine their private-key shares. Therefore, a ciphertext that is obtained by encrypting data under the corresponding collective public key can be decrypted only if t DFU nodes collaborate. Any combination of less than t shares is not enough to recover the private key, and therefore to decrypt the ciphertext.
  - **Randomness DFUs:** These DFUs implement a decentralized randomness beacon that produces public randomness that is publicly-verifiable, unbiasable, and unpredictable [31], [91], [190].

```
1  {
2    "<wf-name>": {
3      "<txn-name>": {
4        "opcodes": [
5          {
6            "dfu-id": "<dfu-id>",
7            "dfu-version": "<dfu-version>",
8            "name": "<opcode-name>",
9            "inputs": {
10             "<input-name>": {
11               "src": "OPCODE|KEYVALUE|PRECOMMIT|CONST",
12               "src-name": "<src-name>",
13               "idx": <idx>,
14               "value": "<value>"
15             }, ...
16           }
17         }, ...
18       ]
19     }, ...
20   }, ...
21 }
```

Figure 2.2: JSON schema for the workflow definition

2.3.4 PROTEAN's programming model

In PROTEAN's DFU-centric execution model, a smart contract can use many DFUs to perform its computations, which requires a mechanism to coordinate the execution of smart contracts across independent networks of nodes. More specifically, programmers need a convenient way to specify which computations and DFUs that they want to use in their smart contracts and how they work together to perform a task. This is new challenge that Ethereum did not have to worry about because the execution of smart contracts take place in a single network. In this section, we introduce the three components of the programmable logic of a PROTEAN smart contract, namely workflows, a finite state machine (FSM) model, and the contract code; and describe how they work together to help with the coordination of DFUs.

Workflows

Smart contract developers write workflow definitions in JSON as shown in Figure 2.2. The `workflows` field holds a set of key-value pairs where keys are workflow names (`wf_name`) and values are the corresponding workflow definitions. A workflow definition consists of one or more transaction definitions that are keyed by transaction names (`txn_name`). Workflow and transaction names are unique within a smart contract and are specified by the smart contract developer.

The opcodes field defines the opcodes that are going to be executed as part of a transaction. An opcode definition includes the following fields:

- **dfu-id:** The globally-unique and persistent identifier (ID) of the DFU that exposes the opcode.
- **dfu-version:** The version number of the DFU. It is an optional field that defaults to the latest available version of the DFU.
- **name:** The name of the opcode as specified by the DFU.
- **inputs:** The input parameters that are required to execute the opcode.

Going back to our software service analogy from earlier, we can think of the (dfu-id, dfu-version) pair as the service endpoint and name as the API call to the service. Since all DFU-related information is publicly available in PROTEAN, both the developers and users of smart contracts can verify that name is a valid opcode that is supported by dfu-id in version dfu-version. In §2.3.7, we explain in detail how DFUs can join PROTEAN and publish new versions of their services. The inputs field is used for expressing the data dependencies of the opcode. Each field of inputs represents an input parameter that is required to execute the opcode. Fields are keyed by the parameter name (input_name), which, like opcode names, are specified by the DFU that exposes the opcode. The src field specifies the data dependency type using one of the default values: OPCODE, KEYVALUE, PRECOMMIT, and CONST. The remaining fields of the input parameter definition become required based on the value of src.

An OPCODE dependency specifies that the output of an opcode is required as an input. The parent opcode is listed at index idx in the opcodes field and has to be in the same transaction as the child opcode. Since an opcode can have multiple outputs, src_name denotes the name of the output parameter whose value is going to be used. An opcode can have one or more OPCODE dependencies and can only be executed after all of its parent opcodes have been executed.

A KEYVALUE dependency indicates that one or more key-value pairs that are stored in the contract state are required as an input to the opcode. The key names whose values are going to be passed to the opcode are listed as a comma-separated string in the value field. Since it is only the state DFUs that maintain the contract states, other DFUs need be able to verify the correctness of the provided key-value pairs. Therefore, for each KEYVALUE dependency, clients need to submit a cryptographic proof that enables DFUs to verify the correctness of the provided data without having to interact with the state DFUs.

Unlike the OPCODE and KEYVALUE dependencies, the data sources for the remaining dependency types are the clients instead of the DFUs. A PRECOMMIT dependency represents a client-provided data that needs to be stored to the contract state when the current transaction commits. The key name under which the client-provided data will be stored is specified in the value field. Even though the data is defined as a dependency in the current transaction, it is actually used as an input by an opcode in a subsequent transaction. Therefore, we refer to this data as *pre-committed data*: from the subsequent transaction's perspective, it will be using a

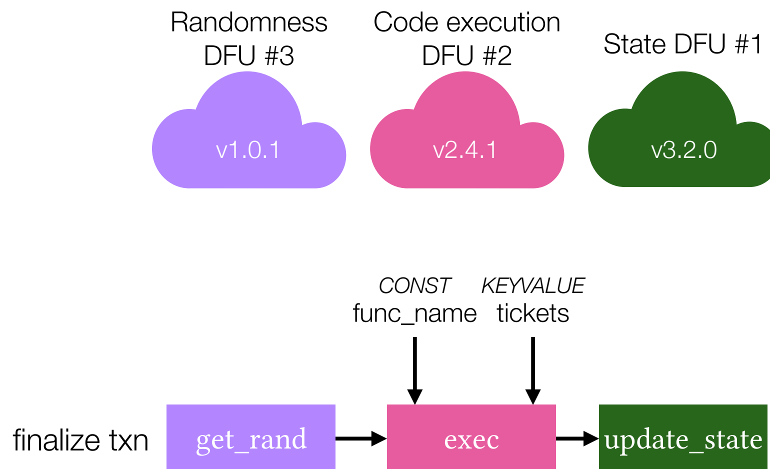


Figure 2.3: Visual representation of an example workflow

previously-committed data. In §2.3.5, we explain why we need pre-committed data. Finally, the input for a `CONST` dependency is hard-coded in the workflow definition in the `value` field.

Figure 2.3 visually represents an example workflow that finalizes a decentralized lottery. This workflow consists of a single transaction (`finalize`) that has three computation steps: (1) parsing a distributed randomness beacon, (2) executing contract code to pick the winner using the parsed randomness, and (3) committing the result to the contract state. Each step corresponds to an opcode, which are represented by colored rectangles, that is executed by a particular version of a DFU, which are represented by clouds with the same colors as opcodes. In the workflow definition corresponding to this figure, values of the `dfu-id` and `dfu-version` fields in the first opcode definition (*i.e.*, `get_rand`) are set to “Randomness DFU #3” and “v1.0.1”, respectively. The directed arrows represent the data and execution dependencies between the opcodes. For example, `exec` can be executed only after `get_rand` is executed since the former requires the output of the latter (*i.e.*, the parsed randomness). `exec` has two additional data dependencies: “func_name” that specifies the name of the function that is going to be executed in the contract code, and “tickets” that specifies the key name that stores the lottery tickets. The corresponding workflow definition has two entries in the `inputs` fields that have `CONST` and `OPCODE` as the values of the `src` field, and “func_name” and “tickets” as the values of the `value` field, respectively.

FSM model

The FSM model is used for representing contract behavior. At any given time, a contract can be in one of its finite set of *stages*[1], which dictates how clients can interact with the contract. The FSM model specifies the set of transactions that are allowed to execute in each stage. Execution of a transaction triggers a stage transition based on the current stage and the outcome of the

[1] We use *stage* to refer to a FSM state to avoid confusion between an FSM state and the contract state.

```
{
  "stages": ["<stage>", ...],
  "initial_stage": "<initial_stage>",
  "transitions": {
    "<txn_name>": {
      "<from>": ["<to>", ...],
    }, ...
  }
}
```

Figure 2.4: JSON schema for the FSM model

transaction. Figure 2.4 shows the JSON schema used for defining the FSM model. The schema contains the following fields:

- **`stages`:** The list of stages that the contract can be in at any given time.
- **`initial_stage`:** The stage that the contract starts in when it is initialized. It has to match one of the stage names listed in `stages`.
- **`transitions`:** The set of valid stage transitions. Each field in `transitions` is keyed by the name of the transaction (`txn_name`) that triggers the stage transition upon its execution. Transaction names have to match the names used in the workflow definitions. Since a transaction can be executed in multiple stages, the `txn_name` field contains a key-value pair for each contract stage in which the transaction can be executed. Executing `txn_name` in stage `from` triggers a transition to one of the stages (`to`) in the list of valid next stages based on the outcome of the execution. Note that `txn_name` is allowed to execute only in the stages specified by `from`. All stage names have to match the names listed in `stages`.

Our decision of using FSMs to model contracts is due to the natural fit between the characteristics of our motivating applications and how FSMs work. Our applications consist of a finite number of stages in which certain actions can be performed. The outcome of the actions determines the next stage of the application. For instance, the e-voting application begins at a stage where users can cast their votes without affecting the contract stage. After a certain amount of time passes, voting is closed and the application advances to a new stage where no one can cast a vote. In this stage, votes are tallied to reveal the results of the election, which would not be permitted in the previous voting stage.

We can also observe that the FSM model complements the workflows: workflows specify the operations that can be performed on the contract, and the FSM model defines when these operations can be performed and how they affect the life cycle of the contract. In §2.3.6, we explain how the FSM model is used to ensure the correct execution of the workflows of a contract.

Contract code

The final component of the programmable logic of a PROTEAN smart contract is the contract code. Developers can write their contract code in any general-purpose programming language that is supported by one of the code execution DFUs in the system. Even though the code execution DFUs do not run consensus, they still need to be able to get the same output when they execute the contract code so that they can produce collectively signed receipts. It is up to the developers to either choose a code execution DFU that enforces deterministic execution of programs or write their programs carefully to avoid non-determinism.

As we have discussed before, we expect the contract code to mainly serve as a glue code that ties together the inputs and outputs of the opcodes. Since DFUs operate independently from each other, they might have incompatible communication protocols, data formats, or APIs. Contract codes enable DFUs to be able to "talk" to each other by making sure that the correct data transformations are applied. In addition to the data management tasks, the contract code is also responsible for preparing the state change of a transaction based on the outputs of the opcodes and performing contract-specific computations that are not supported by the DFUs (*e.g.*, XORing a list of random byte arrays in the decentralized lottery).

2.3.5 State management

PROTEAN smart contracts rely on state DFUs to store and manage their states. State DFUs provide the contracts with a key-value store where each contract has a separate namespace to store its data. The contract developer deploys the contract to a state DFU instance of their choosing to make it available to the users. To do so, they send the workflow and FSM model definitions, and the cryptographic hash of the contract code to the state DFU. The state DFU creates a key-value namespace for the contract and generates a contract ID (`CID`) that uniquely identifies the namespace within the DFU. The `CID` is generated using the hash of the definitions and the contract code to cryptographically bind them together. Every contract namespace has a *header* that consists of six reserved keys: `CID` stores the unique ID of the contract, `workflows` stores the workflow definitions, `fsm` stores the FSM model definition, `code_hash` stores the hash of the contract code, `curr_stage` stores the name of the current FSM stage that the contract is in, and `lock` stores a binary value that shows if the contract is locked.

Since computing DFUs do not maintain contract state, they need to be able to verify the authenticity and integrity of the contract state that the clients provide them. To this end, we require state DFUs to produce publicly-verifiable state proofs that enable any party to verify that a given set of key-value pairs belong to a valid contract state. Verification of a state proof should not require interaction with the state DFU that produced it. Additionally, each state proof should contain the state version number so that DFUs can detect stale contract state, which we are going to discuss in the next section. We assume that clients can retrieve key-value pairs and the associated state proofs from state DFUs without having to use PROTEAN

transactions.

In contrast, a contract's state can only be updated by its programmable logic and via an explicit opcode execution at the end of a transaction. Because of PROTEAN's DFU-centric and client-driven execution model, there can be multiple transactions that try to concurrently execute on the same contract state. To guarantee that transactions can execute safely without leaving the contract in an inconsistent state, we employ an approach that is based on optimistically executing transactions against the contract state and resolving conflicts at commit time by aborting transactions.

In our optimistic approach, a transaction *T* executes in three phases. In the *initialize phase*, *T* is marked with an identifier that indicates the current version of the relevant contract state. The state DFU updates the identifier after any change to the key-value pairs in the contract state. Therefore, we can think of each identifier as a version number that operates at contract level. State DFU instances can use different methods for generating an identifier (*e.g.*, monotonically-increasing counter, system clock). In the rest of this chapter, we are going to assume that state DFUs use Merkle trees [148] to store the contract state and the hash of the Merkle root is used as the version number. During the *execute phase*, DFUs execute the opcodes of *T* and create the state updates. Finally, in the *commit phase*, *T* tries to commit its state updates to the state DFU. If the current Merkle root hash of the contract state matches the Merkle root hash observed by *T* at the initialize phase, *T* successfully commits. If the hashes do not match, it means that at least one other transaction *T'* has successfully committed and changed the contract state after *T* started. In this case, *T* aborts to avoid any potential conflicts and retries execution by going back to the initialize phase.

Contract locking: The optimistic approach is a good fit for short-lived workflows with a single transaction that can be aborted and retried without creating side-effects. However, there are cases where using the optimistic approach can violate the correctness or safety of a contract. Consider a workflow that is composed of multiple transactions to divide a large computational task into subtasks. Transactions of a multi-transaction workflow need to be executed serially on the contract state to successfully perform the task. However, in the optimistic approach, execution of a multi-transaction workflow can be interleaved with transactions from other workflows, which can leave the contract in an incorrect state and prevent the successful execution of the multi-transaction workflow. Moreover, if a transaction involves irreversible operations (*e.g.*, decryption, revealing a secret), using the optimistic approach can undermine the safety of the contract since the effects of these operations cannot be reverted if the transaction aborts. For example, a malicious client can reveal a secret data by executing the transaction and then wait until the contract state to change to try to commit the transaction. Since the transaction will abort, the contract state will not show that the transaction has executed, even though the malicious client has executed the transaction and obtained the secret.

PROTEAN supports a contract-level locking primitive that can work together with the optimistic

transaction execution approach to provide workflows with an alternative concurrency control mechanism that can be used in the above scenarios where using the optimistic approach alone is not a viable option. A workflow that uses contract-level locking consist of multiple transactions and begins its execution by locking the contract state. To do so, it uses a *lock* transaction to set the value of the `lock` key in the header of the contract namespace. Once the contract is locked, only the transactions that are defined in the locking workflow can be executed optimistically as described before. The final transaction of the workflow unlocks the contract state by unsetting the `lock` key when it commits.

Since we assume a Byzantine setting, there is the risk that a contract remains locked if a malicious or unreliable client disappears without driving the execution of a locking workflow to completion. PROTEAN's client-driven execution model addresses this risk by allowing other clients to fill in for an unresponsive client to drive the execution of the workflow. To be able to do so, the new client needs to have access to all the data that will be required by the transactions in the workflow, including any non-public data that the disappearing client possessed and was supposed to input to the transactions. To this end, PROTEAN implements a mechanism that forces clients to *pre-commit* their inputs to the contract state when they execute the lock transaction. The contract developer specifies the data that needs to be pre-committed by using the `PRECOMMIT` dependency (§2.3.4) in the definition of the lock transaction. Later, when a client tries to commit the lock transaction, the state DFU verifies that the state updates of the transaction include the data specified by the `PRECOMMIT` dependencies. If the verification is successful, the state DFU atomically commits the data and locks the contract state. In this way, if the original client disappears, a different client will have access to all the necessary data to be able to push the workflow through completion until the contract is unlocked again.

An alternative mechanism relies on DFUs for ensuring the liveness of workflow execution. This approach is based on a garbage collection type process that causes the DFU nodes themselves to periodically check for any uncompleted workflows. The DFU nodes act temporarily as "surrogate clients" to get the uncompleted workflows run to completion. In our PROTEAN prototype, we implement the first approach where clients are responsible for ensuring the liveness of workflow execution. We leave the implementation of the DFU-based mechanism as a future work.

2.3.6 Executing workflows

In our client-driven execution model, clients are responsible for orchestrating the execution of a workflow by communicating with the DFUs. The main challenge that we need to address in this model is to guarantee the correct and uninterrupted execution of workflows in the face of malicious and unresponsive clients. Since transactions are the building blocks of workflows, we are going to discuss in detail the steps involved in executing a transaction. As in §2.3.5, we break down the life cycle of a transaction to three phases (*i.e.*, initialize-execute-commit) and describe the challenges involved in each stage and how we address them.

```
type ExecutionPlan struct {
  CID          []byte
  StateVersion []byte
  CodeHash     []byte
  TxnName      string
  Txn          Transaction
  DFUData      map[string]DFU
  DFUID        string
  Sig          []byte
}
```

Figure 2.5: PROTEAN execution plan

Initializing a transaction

To execute a transaction, clients first need to create an *execution plan*. An execution plan is a cryptographic data structure that is created in accordance with the workflow definition to serve clients and DFUs as a roadmap during transaction execution. Clients attach the execution plan to the opcode requests that they send to the DFUs as part of orchestrating the execution of the transaction. An execution plan contains information, such as the data-flow graph of the transaction, current state of the contract, and the cryptographic identities of the DFU nodes, that is necessary for verifying the correct execution of the transaction. Every code execution DFU that is registered to PROTEAN is required to support the functionality of generating execution plans. However, clients have to choose one of the code execution DFUs that is specified in the workflows of the contract. The reason behind this restriction is to make sure that clients do not use a DFU that is outside of the trust base of the contract.

A code execution DFU generates an execution plan for a given transaction if and only if the contract is in a state where the transaction can be executed. To this end, the client sends a request to the code execution DFU with the current header of the contract and the state proof associated with it. Additionally, the client specifies the name of the transaction that they want to execute and the name of its workflow. Upon receiving a request, the code execution DFU first verifies the state proof for the contract header to make sure that it is part of a valid contract state. Then, it verifies that it is one of the DFUs that is specified in the workflows, and it is therefore eligible to create the execution plan for this transaction. Finally, using the FSM model of the contract, it verifies that the transaction can be executed in the current contract stage. Recall that the contract header contains all the information that is needed (*i.e.*, `workflows`, `fsm`, and `curr_stage`) to be able to perform the above verifications.

If all verifications are successful, the code execution DFU creates the execution plan, which contains the following fields as shown in Figure 2.5:

- **`CID`:** The ID of the contract that contains the transaction. It is extracted from the contract header.

```
1 type Receipt struct {
2   EPID []byte
3   Idx  int
4   Name string
5   Hash []byte
6   Sig  []byte
7 }
```

Figure 2.6: PROTEAN receipt

- **`StateVersion`:** The current version number of the contract state. It is extracted from the state proof of the contract header.
- **`CodeHash`:** The cryptographic hash of the contract code. It is extracted from the contract header.
- **`TxnName`:** The name of the transaction.
- **`Txn`:** A data structure that stores the information about the transaction, as specified in the workflow definition.
- **`DFUData`:** A collection of key-value pairs where each key is the ID of a DFU (`dfu-id`) that is used in the transaction, and the value is a data structure that stores the public keys of the DFU nodes and the threshold of the DFU. The client uses the information in the transaction definition (`dfu-id` and `dfu-version`) to retrieve the DFU data and the accompanying proof from the DFU registry. The code execution DFU receives the data and the proofs from the client as part of the request.
- **`DFUID`:** The ID of the code execution DFU that created this execution plan.
- **`Sig`:** The collective signature of the code execution DFU over the cryptographic hash of the execution plan.

To summarize, an execution plan serves multiple purposes. First, it links the transaction execution to a particular version of the contract state that is identified by `CID` and `StateVersion`. As we have discussed in §2.3.5, state DFUs need this information later to decide whether a transaction can commit or needs to abort. Second, it specifies the cryptographic hash of the contract code to prevent a malicious client from executing a different, unauthorized code that does not belong to the contract. Finally, it securely provides DFUs with data structures (`Txn` and `DFUData`) that are created using the workflow of the transaction. Using these data structures, DFUs can ensure that the client orchestrates the execution of the transaction correctly, without deviating from the workflow. The code execution DFU collectively signs the execution plan (`Sig`) so that DFUs can verify that it is generated correctly and has not been tampered with by a malicious client.

Executing a transaction

The responsibility of the client at this stage is to drive the execution of the transaction. To

do so, they send requests to DFUs to execute the opcodes of the transaction in an order that is consistent with the data-flow graph of the transaction. The client is also responsible for providing DFUs with the correct input data that are required for executing the opcodes as prescribed in the workflow. Since clients can be malicious, DFUs need to be able to check that the parent opcodes of their opcode have already been executed. Additionally, they need to be able to verify the authenticity and integrity of the opcode outputs that they receive via the client.

To address these challenges, DFUs produce an *output receipt* for each output of their opcode. The receipt binds the output of the opcode to the ongoing execution of the transaction and is collectively signed by the DFU nodes. Figure 2.6 shows the data structure that represents an output receipt:

- **`EPID`:** The cryptographic hash of the execution plan (excluding the collective signature).
- **`Idx`:** The index of the opcode in the `OPCODE` list.
- **`Name`:** The name of the output parameter.
- **`Hash`:** The cryptographic hash of the output.
- **`Sig`:** The collective signature of the DFU over the cryptographic hash of the output receipt.

The client includes the relevant output receipts in their opcode request so that the DFU can verify that the parent opcodes of its opcode have been executed correctly. The `EPID` enables the DFU to verify that the parent opcodes have been executed using the same execution plan. The `Idx` and `Name` enable the DFU to match the output receipt with the `OPCODE` dependency in the workflow. The DFU can use the value of `Hash` to confirm that the client has not tampered with the output of the parent opcode. Finally, the collective signature (`Sig`) of the DFU proves the authenticity and integrity of the output receipt.

Although collectively-signed output receipts ensure that a malicious client cannot execute opcodes out of order or tamper with the opcode outputs, there is still a type of attack that they cannot prevent. Consider a transaction where the output of an opcode (op_i) is used as an input of two opcodes (op_j and op_k). If op_i produces a different output at every execution, a malicious client can re-execute op_i multiple times to obtain different outputs. The client can then equivocate by providing op_j and op_k with different results. To prevent this equivocation attack, DFUs produce an *input receipt* for each of their input with an `OPCODE` dependency. Input receipts use the same data structure as the output receipts by replacing the value of `Name` and `Hash` fields with the name of the input parameter and the cryptographic hash of the input data, respectively. State DFUs check the input receipts at the commit phase to detect any equivocation and abort the transaction if they do so.

Output receipts guarantee that the `OPCODE` dependencies of an opcode are satisfied correctly. However, there are three other dependency types (`KEYVALUE`, `PRECOMMIT`, and `CONST`) that DFUs need to make sure that are correctly satisfied. DFUs can verify that the `PRECOMMIT` and `OPCODE` are satisfied correctly by using only the information in the execution plan (`Txn`). For

the KEYVALUE dependencies, DFUs rely on state proofs to verify that clients provide them with the correct contract state. One potential attack that a malicious client can attempt is to provide the computing DFUs with stale contract state and state proofs. Since computing DFUs do not maintain contract state and do not communicate with the state DFUs, they cannot distinguish between an old state and the current state of the contract as long as the accompanying proofs are valid. However, DFUs can thwart this attack by confirming that the state version in the state proofs matches the state version in the execution plan.

To summarize, DFUs perform the following steps before they accept an incoming opcode request:

1. Compute the cryptographic hash of the received execution plan and verify the collective signature on it. Use the DFUID in the execution plan to retrieve the public keys and the threshold information from DFUData.
2. Check that the DFU ID, DFU version, and the opcode name that are specified in Txn match with this DFU's.
3. If this is a code execution DFU and the opcode request is for executing the contract code, confirm that the cryptographic hash of the to-be-executed code matches the value of CodeHash.
4. Verify that the client has provided the necessary inputs that satisfy the data dependencies of the opcode. For each input, perform one of the following checks based on the data dependency type:
   (a) OPCODE: Check that the values of Idx and Name in the client-provided output receipt match the values in Txn. Compute the cryptographic hash of the received data and confirm that it matches the value of Hash in the receipt. Verify the collective signature on the receipt.
   (b) KEYVALUE: Verify that the CID and version number in the client-provided state proof match the values in the execution plan. Check that the state proof contains an inclusion proof for all the key-value pairs that are listed in the definition of the dependency. Verify the state proof.
   (c) PRECOMMIT: Check that the client has provided a value for each key that is listed in the definition of the dependency.
   (d) CONST: Compute the cryptographic hash of the client-provided data and the hard-coded value in the workflow and confirm that they match.

Committing a transaction

The final opcode of every transaction is for committing the state changes that are created by the execution of the transaction. To do so, the client sends an opcode request to the state DFU that maintains the state of the contract to commit the transaction. In their request, the client includes all the input receipts that are generated during the execution of the transaction so that the state DFU can verify that the client have not performed an equivocation attack. Since committing a transaction is performed via an opcode, state DFUs perform the verifications

that we have described above. However, they also perform additional checks before deciding whether a transaction can commit or not.

First, as we have described in §2.3.5, the state DFU has to make sure that the contract state, which the transaction has been executing on, has not changed after the initialization of the transaction. To do so, the state DFU simply compares the state version information in the execution plan to the current version of the state. If they do not match, the transaction aborts without committing its changes to the contract state. In that case, the client can retry the transaction if the contract is still at a stage where the transaction can be executed. Otherwise, the client can try to execute the next transaction based on the current stage of the contract. Second, the state DFU needs to ensure that the client has not performed a equivocation attack during the transaction execution. The state DFU uses the data-dependency graph of the transaction and the input receipts to verify that if two opcodes have the same `OPCODE` dependency, then they should have received the same data for this dependency. If the state DFU finds an inconsistency in the input receipts, the transaction aborts. If all the checks are successful, then the transaction commits its state changes and the contract moves to its next stage.

2.3.7 DFU management

The core idea of PROTEAN is to create a permissionlessly extensible platform where DFUs can freely join and offer their specialized functionalities to smart contracts. Therefore, we need a mechanism that enables DFUs to register to PROTEAN and subsequently manage their services. To this end, PROTEAN employs a *DFU registry* that facilitates the registration/deregistration, management, and discovery of the DFUs. DFUs interact with the registry to join/leave the system or release a new version of their service. Developers use the information in the registry to inspect various aspects of a DFU, such as the underlying protocol and its implementation, security assumptions, API documentation, and which nodes run the DFU, to decide whether they want to use its services in their smart contracts. Clients fetch DFU information from the registry so that they can provide the DFUs with this information when they orchestrate the execution of a contract. The DFU registry maintains a database that builds on top of a blockchain that uses Byzantine fault-tolerant consensus to handle the requests from DFUs and store the DFU information. It provides proofs that can be used by any party to verify that the information of a DFU is stored on-chain.

To register to PROTEAN or release a new version, a DFU sends a request to the registry with a JSON file, which is shown in Figure 2.7, that contains the following information:

- **`dfu_id`:** The unique identifier of the DFU. If the request is for registering for the first time, the value of this field is `null`. The ID is the same for all versions of the DFU.
- **`version_num`:** The version number of the DFU. For each version, the DFU needs to provide the following information:
  - **`num_nodes`:** The number of nodes in this version.

```
1   {
2     "dfu-id": "null" | "<dfu_id>",
3     "<version_num>": {
4         "num_nodes": <num_nodes>,
5         "threshold": <threshold>,
6         "node_ids": {
7             "<ipaddr>": "<public_key>", ...
8         },
9         "opcodes": ["<opcode_name>"],
10        "resources": "<resource-address>",
11    }, ...
12  }
```

Figure 2.7: JSON schema for DFU

- **`threshold`:** The verification threshold for the collective signatures of this version.
- **`node_ids`:** The IP addresses and the public keys of the participating nodes.
- **`opcodes`:** The list of opcodes that are supported in this version.
- **`resources`:** The reference to the resources about this version (*e.g.*, source code, API documentation).

When registering for the first time, the DFU sends a collective signature on the cryptographic hash of the JSON file. The registry uses the public keys and the threshold in the JSON file to verify the signature. If everything checks out, registry generates a unique ID for the DFU and creates a record for the DFU in its database. DFUs use semantic versioning [176] to manage and communicate the changes in their versions. Therefore, version numbers take the form `MAJOR.MINOR.PATCH`, where each element is a non-negative integer. The request for releasing a new version is collectively signed by both the nodes that are participating in the new version and the nodes that are participating in the previous version [162]. In addition to the underlying software, the set of participating nodes and signature verification thresholds can change across the versions. It is the DFU's responsibility to remove a version from the registry if it cannot be supported anymore.

2.4 Security discussion

The permissionlessly-extensibility principle of PROTEAN enables any set of nodes to freely join our system as DFUs and start providing their services to the smart contracts. The advantage of this architecture is that we can introduce many specialized functionality to our system in an easy and modular way. However, since DFUs run specialized computations with a diverse set of threat and network models, we cannot realistically define a security model that works for every DFU that can ever join the system. Therefore, as we have discussed earlier, each DFU in PROTEAN self-declares their threat model and security assumptions (*e.g.*, how many faulty nodes they can tolerate), and we assume that they operate in accordance with them.

Therefore, it is the developers' responsibility to select the DFUs that they want to trust and use in their smart contracts based on the publicly-available information about the DFUs. In this way, using the services of a DFU in a smart contract is analogous to using third-party software in an application.

The publicly-available information about a DFU includes the source code that it runs and its performance metrics, the description of its threat model, the identities of its nodes, the detailed description of its membership and governance policies, and any other data that the DFU chooses to disclose to help developers in their decision making. Developers would use a combination of the security requirements of their smart contracts and their judgment of which DFUs are trustworthy based on the publicly-available information about DFUs to choose the DFUs that they want to trust and use in their smart contracts.

While deciding whether a DFU can be trusted or not is a challenging and subjective task, we present a few examples of what can influence the decision-making process. For example, other things being equal, a developer might choose a DFU that runs its service on a trusted execution environment (*e.g.*, Intel SGX **anati2013**) over a DFU that does not, since the former can provide additional confidentiality and integrity guarantees and also enable any party to cryptographically verify that the DFU is running the correct software. Another example is related to the membership and governance policies of a DFU. Even though PROTEAN is a permissionless system, there is no restriction on how DFUs internally manage their memberships. Therefore, developers might prefer permissioned DFUs over permissionless DFUs in their smart contracts. One concrete example of a permissioned system that can potentially serve as a DFU in PROTEAN is the League of Entropy consortium [62]. It is composed of 16 independent organizations, which involves companies, research labs, and universities, that collectively run a verifiable, decentralized randomness beacon. Since the consortium has strict membership requirements and comprises reputable organizations from different countries, we believe that systems like League of Entropy would be preferred by smart contract developers in PROTEAN.

Even though the previous examples demonstrate how certain characteristics of a DFU can make it more trustworthy than the other DFUs, a developer can still choose malicious DFUs to use in their smart contract. In this scenario, PROTEAN cannot guarantee the correct execution of the smart contract. Below, we discuss two approaches that we can employ in PROTEAN to create secure DFUs:

- We can use a bias-resistant sharding protocol that is similar to that of OmniLedger [135] to sample from a large population of nodes to create DFUs with given security thresholds (*e.g.*, at most 1/3 of the DFU nodes are malicious). The population can be the nodes of every DFU that is registered to PROTEAN. Given an adversarial model over the whole population, we can create state DFUs and code execution DFUs of various sizes that can tolerate different numbers of malicious nodes. In this way, PROTEAN can create a trusted computing base that supports distributed consensus and replicated code execution. We restrict the trusted computing base to state and code execution DFUs

since randomly assigning nodes to DFUs that perform more specialized computations (*e.g.*, cryptographic protocols) would not be feasible because nodes might not have the adequate resources for performing these specialized computations. Moreover, providing secure state and code execution DFUs means that PROTEAN can provide the same functionality that Ethereum does in a secure way. If developers want to use more specialized DFUs in their smart contracts, they are responsible for choosing the honest DFUs as we described before.

- In addition to registering to PROTEAN collectively as DFUs, we can also enable nodes to register individually. Developers can then choose from these nodes to create ad hoc DFUs that can perform any functionality that they want for their smart contracts. In this case, it would be the developer's responsibility to provide the nodes with the software that they are supposed to execute as part of providing the specialized functionality. The advantage of this approach over the previous one is that it can enable developers to create specialized DFUs with the nodes that they trust.

2.5 Implementation

We implemented a prototype of PROTEAN with a specific implementation of each DFU type that are listed in §2.3.3. Our implementation is in Go [94], [198] and has ~4900 LoC. Additionally, we implemented the three motivating applications that are described in §2.2. We use Onet [196], an open-source overlay network library for simulating and deploying distributed protocols, as the underlying communication layer for DFUs. Onet sets up point-to-point TCP connections between the DFU nodes and uses Protobuf for data serializing. For cryptographic primitives and operations, we use Kyber [55], an open-source advanced cryptographic library for Go, and the Go standard library.

2.5.1 Cryptographic primitives

We use the Boneh-Drijvers-Neven (BDN) multi-signature scheme [25], which is based on the Boneh-Lynn-Shacham (BLS) signature scheme [26], with a 256-bit Barreto-Naehrig (BN256) curve for collectively signing the input and output receipts in DFUs. BDN prevents rogue-key attacks without requiring the parties to prove knowledge of their private keys.

We use the signature aggregation property of BDN that allows many signatures on the same message to be combined into a single signature (*i.e.*, a multi-signature). This property has two performance benefits: First, since both the regular and aggregated BDN signatures have the same size, aggregation reduces the space complexity. Second, since verifying a single BDN signature and the aggregation of n BDN signatures use the same number of costly pairing operations, aggregation improves the verification performance.

Clients use the ElGamal encryption system [86] to encrypt their data and Schnorr signatures [173] to sign their inputs. We use the Edwards25519 elliptic curve implementation in

Kyber for both operations. Finally, we use the SHA-256 cryptographic hash function [160] in the Go crypto package.

2.5.2 DFU implementations

State DFU: We implement a state DFU on top of a programmable blockchain that runs an implementation of ByzCoin [134], which is a scalable Byzantine consensus protocol. In the rest of the thesis, we are going to use ByzCoin to refer to the blockchain. We implement the key-value store on top of ByzCoin as a global smart contract. The blockchain uses skipchains as its underlying data structure for storing the blocks. Skipchains combine blockchains with skiplists to enable traversal of the blockchain both forward and backward in time. Skipchains use multi-hop links in both directions to efficiently traverse arbitrary distances along the timeline in a logarithmic number of steps. The multi-hop forward links are created via cryptographic collective signatures and enable any party to efficiently prove (regardless of where its block is in the timeline) that a transaction is correctly stored in the blockchain.

Forward links are an integral component of a ByzCoin proof, which has three components: (1) a Merkle proof that proves the presence or absence of a key, (2) the latest block where the Merkle tree root that is used in the Merkle proof is stored, and (3) a list of forward links to prove that the latest block is part of the blockchain. Since any party can use the forward links to verify the proof without having to follow the blockchain, ByzCoin proofs are a great fit for PROTEAN 's architecture where DFUs do not interact with each other. In our prototype, we build state proofs on top of ByzCoin proofs without making changes to the underlying blockchain implementation for simplicity purposes. However, our decision of using ByzCoin as it is comes at a cost: state proof size is proportional to the contract state size since the proof contains all the key-value pairs. It is necessary to emphasize that this is a limitation of our prototype implementation and not PROTEAN 's architecture.

Code execution DFU: We implement a code execution unit that can execute programs written in Go. Since our execution environment does not enforce deterministic execution, we assume that the contracts are written carefully to avoid non-deterministic computations.

Mix-net DFU: We implement a mix-net DFU using Neff's verifiable shuffle protocol [161]. A common way of building a mix-net using verifiable shuffling is to have a set of servers perform a shuffle one after another. Each server shuffles the output of the previous shuffle (except for the first server that shuffles the input ciphertexts) and creates a zero-knowledge proof of correct shuffling. The output of the mix-net contains the output of each shuffle step, and assuming that at least one server is honest, the output of the mix-net cannot be linked to its input.

In our implementation, we modify the above setting to make it compatible with PROTEAN's

requirement of DFUs creating and collectively signing output receipts. Our scheme also starts with the DFU nodes performing shuffles in a sequential order. Assuming that the DFU can tolerate up to f faulty nodes, the DFU first performs $f+1$ shuffles. Once these shuffles are done, nodes verify the output of each shuffle to make sure that there is at least one correct shuffle. If that is the case, nodes use the output of the mixing process to create the corresponding output receipt and collectively sign it.

Threshold decryption DFU: Our threshold decryption DFU implementation uses Pedersen's distributed key generation (DKG) scheme [166] to create a collective public-private key pair. Clients encrypt their data with the collective public key of the DFU using a variant of Shoup and Gennaro's TDH2 threshold encryption scheme [179] as described by Leuks [145]. For decrypting a ciphertext, one approach would be to have each node produce their decryption share and the corresponding non-interactive proof of correct decryption. Then, anyone can collect these shares and proofs, and if they can verify that at least t shares are decrypted correctly, they can recover the original plaintext.

We modify the above approach to make sure that it is compatible with PROTEAN. We use a two-phase protocol, as we do in the mix-net DFU, where in the first phase nodes create their decryption shares and proofs, and in the second phase, they all verify the same set of shares and recover the plaintext. In this way, nodes can create an output receipt for the plaintext. Reconstructing the plaintext at the DFU does not violate the confidentiality of the data, since in our applications, the recovered plaintexts are stored at the publicly-visible state DFU right after the decryption. However, it is also possible to implement this DFU such that the nodes only verify the proofs, and create the output receipt for the set of shares and proofs.

Randomness DFU: We implement a drand-style [62] distributed randomness beacon that builds on two threshold cryptography primitives: DKG and threshold BLS signatures. As part of the setup phase, the randomness DFU runs Pedersen's DKG protocol to create a collective public-private key pair where every node in the DFU holds a share of the collective private key. After the setup phase, the randomness DFU operates in rounds, and in every round, it generates a random number. We implement the chained mode of the drand protocol where the random number of the current round r builds on the random number created on round $r-1$. More specifically, in round r, each node in the DFU uses their private key share to create a partial BLS signature on $\mathrm{H}(r \parallel sig_{r-1})$ where sig_{r-1} is the collective BLS signature from round $r-1$ and H is a cryptographic hash function. The collective BLS signature is obtained by combining at least t valid partial BLS signatures and its hash is the random number for round r. The protocol and the underlying cryptographic primitives guarantee that the generated randomness is publicly-verifiable, unbiased and unpredictable.

2.5.3 Applications

In this section, we describe the implementation of our motivating applications (§2.2) on top of our PROTEAN prototype. More specifically, we present two variations of a decentralized lottery and an e-voting application.

Decentralized lottery with a distributed randomness beacon

Our first application is a decentralized lottery that uses a publicly-verifiable, unpredictable and unbiased randomness beacon to pick a winner. In the rest of the thesis, we are going to refer to this application as `randlotto`. We provide the FSM model and workflows for `randlotto` in Appendix A.1. The `randlotto` contract is initially at the `lottery_open` stage where participants can enter the lottery by registering themselves with the contract. Later, if the predefined necessary conditions are met, the contract transitions to the `lottery_closed` stage where the lottery is closed for new participants. Finally, the winner is picked by using the randomness beacon that is provided by the randomness DFU and the contract terminates by reaching the `lottery_finalized` stage.

Participants execute the `join` transaction to enter the lottery. Each participant sends their public key and a signature over the hash of the public key to the `join_randlotto` function of the contract code. If the contract code can verify the signature, it prepares a `writeset` to store the public key-signature pair, which represents a ticket, in the contract state. The participant then contacts the state DFU with the `writeset` to update the contract state. In Ethereum-based lotteries, participants enter a lottery by depositing a specific amount of money to an Ethereum account to buy a lottery ticket. Since our PROTEAN prototype does not have a payment system, we assume that there is an external system where participants can make payments and receive a proof of their payment. We replace the verification of the proof of payment with the verification of the participant's cryptographic identity.

Any interested party (*e.g.*, the lottery organizer or a lottery participant) can execute the `close` transaction once the predefined necessary conditions are met. In our application, we use the block height of the underlying blockchain in the state DFU to specify a barrier point. More specifically, the `CONST` dependency `barrier` of the `exec` opcode specifies a future block height after which the lottery can be closed. The value of `barrier` is hard-coded in the workflow definition by the contract developer and is publicly visible so that participants can make sure that they have enough time to enter the lottery. The `close_randlotto` function of the contract code checks that the barrier point is reached by checking the block height information that is included in the state proof. Based on the outcome of this check, the client can execute the subsequent `update_state` opcode to trigger the transition to the `lottery_closed` stage.

Finally, as with the `close` transaction, any interested party can execute the `finalize` transaction to pick the winner of the lottery. The `get_randomness` opcode is executed to get a random value from the randomness DFU. As we explained before, our implementation of

the randomness DFU periodically generates random numbers in rounds. The CONST round dependencies of get_randomness and exec opcodes specify the round number whose randomness will be used. The finalize_randlotto function verifies both the correctness of the generated randomness and that the randomness is parsed from the correct round. If everything is correct, finalize_randlotto uses the randomness to pick the winning ticket and prepares the writeset to update the contract state with the result of the lottery.

Decentralized lottery with threshold decryption

The second decentralized lottery application combines threshold cryptography with a commit-reveal scheme to generate a random value, which is produced by combining the encrypted inputs of lottery participants. In the rest of the thesis, we are going to refer to this application as secretlotto. We provide the FSM model and workflows for secretlotto in Appendix A.2. The secretlotto contract starts from the lottery_setup stage where the nodes in the threshold decryption DFU create a collective public-private key pair. The stages that are subsequent to lottery_setup are identical to the stages of randlotto. As we explain below, the main difference is in the transactions that are used for triggering the stage transitions.

The setup transaction begins with the run_dkg opcode of the threshold decryption DFU. run_dkg runs the DKG protocol and outputs a collective public key that will be used by the participants to encrypt their lottery tickets. The output of run_dkg, which is pk, is passed to the setup_secretlotto function of the contract code, which produces a writeset to store the collective public key in the contract state.

Participants execute the join transaction to enter the lottery. Each participant picks a random value and encrypts it with the public key generated by the DKG protocol. The ciphertext serves as a commitment to the generated random value and is sent to the join_secretlotto function to be stored in the contract state. As in randlotto, we assume that there is an external payment system where participants can make payments.

The close transaction works the same way as the close transaction in randlotto, as they both use block height as the barrier point. However, even though the finalize transaction performs the same high-level task of picking the lottery winner as in randlotto, it is composed of different opcodes that perform different computations. The first exec opcode of the finalize transaction is for executing the prepare_decrypt function that outputs a request message with the stored ciphertexts. The client forwards the request to execute the decrypt opcode, which requires the output of exec for its ciphertexts parameter. The output of decrypt is the plaintext random values that were originally generated by the participants. The second exec opcode is for executing the finalize_secretlotto function that XORs the plaintext random values to generate an unbiased and unpredictable random value and uses this value to select the winning ticket.

E-voting

The final application is the implementation of an e-voting system that protects the privacy of the votes and provides unlinkability of voters and their votes. We use the threshold decryption DFU to store the votes in encrypted format and the mix-net DFU to verifiably shuffle the encrypted votes so that the plaintext votes cannot be linked to the original encrypted votes. In the rest of the thesis, we are going to refer to this application as `e-voting`. We provide the FSM model and workflows for `e-voting` in Appendix A.3. The `e-voting` contract starts from the `election_setup` stage where the threshold decryption DFU runs DKG to create a collective public-private key pair. Once the setup is over, the contract moves to the `election_open` stage where clients use the collective public key to encrypt their votes and submit their encrypted votes to the contract state. After the contract moves to the `election_closed` stage, the election can be finalized by shuffling and decrypting the encrypted votes in succession to calculate the election results. During this process, the contract first moves to the `election_shuffled` stage and then to the `election_finalized`, which is the final stage.

The `setup` and `vote` transactions are very similar to the `setup` and `join` transactions of the `secretlotto` so we are not going to discuss them in detail. Instead, we are going to take a closer look at the `finalize` workflow, which contains three transactions: `lock`, `shuffle`, and `tally`.

The `lock` transaction both locks the contract state and triggers a stage transition that ends the voting phase. Similar to the previous applications, `e-voting` uses block height as the barrier point. The `PRECOMMIT` dependency of the `exec` opcode requires the client, who executes this transaction, to submit a generator value (`h`), which is later used by the `shuffle` transaction. The code execution DFU adds the generator value to the `writeset` so that it can be committed to the contract state at the end of this transaction.

The `shuffle` transaction begins with executing the `prepare_shuffle` function of the contract code to prepare a shuffle request with the encrypted ballots and the pre-committed generator value. The client sends the request to the mix-net DFU to execute the `shuffle` opcode. The final opcodes, namely `exec` and `update_state`, process the output of `shuffle` and prepare the state change that is going to be committed to the contract state. The final transaction `tally` is similar to the `finalize` transaction of `secretlotto` in that it is for collectively decrypting a list of ciphertexts and processing the plaintexts in the contract code (*i.e.*, `tally_votes` function of the contract code) to finalize the application.

2.6 Evaluation

In our experiments, we evaluate the performance of PROTEAN to answer the following questions:

1. What is the overhead of verifying the data dependencies and generating the output

receipts at DFUs?
2. What is the overhead of the modifications made to the Neff shuffle and threshold decryption protocols to make them compatible with PROTEAN?
3. Can PROTEAN run general-purpose decentralized applications with acceptable performance and level of decentralization?

The first two questions are important questions to answer to understand the overhead due to PROTEAN's modular and DFU-centric execution model where execution of a smart contract takes place across multiple DFUs. More specifically, the first question helps us understand the overhead of the techniques that we employ to be able to correctly and securely coordinate the execution of smart contracts across DFUs. The second question also aims at understanding the consequences of PROTEAN's execution model. However, this time, we want to quantify the cost of adapting the specialized protocols of the two DFUs that we implement in our prototype to make them compatible with PROTEAN's execution model. Finally, the final question tries to answer the question of whether PROTEAN is a feasibly-practical architecture where decentralized applications can be deployed.

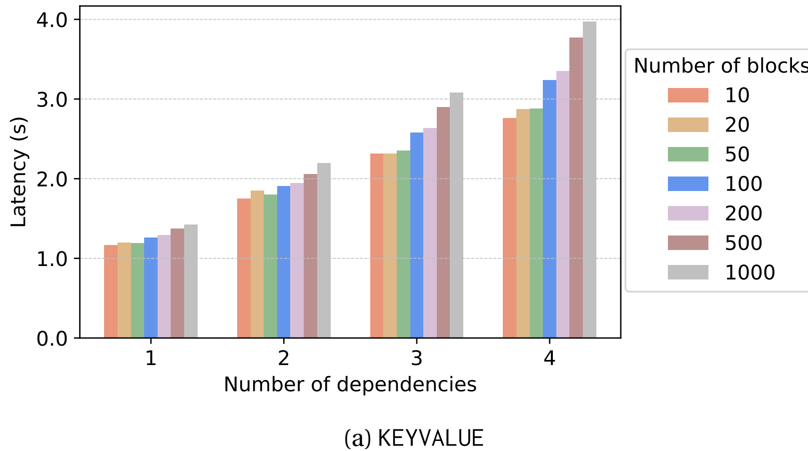
2.6.1 Experimental setup

We run our experiments on a Ubuntu 20.04 VM instance that is equipped with 16 vCPU (Intel Xeon Silver 4216 CPU @ 2.10 GHz) and 64 GB RAM. We use Mininet to simulate a realistic network where the link delay is 100 ms and the link bandwidth is 100 Mbps.

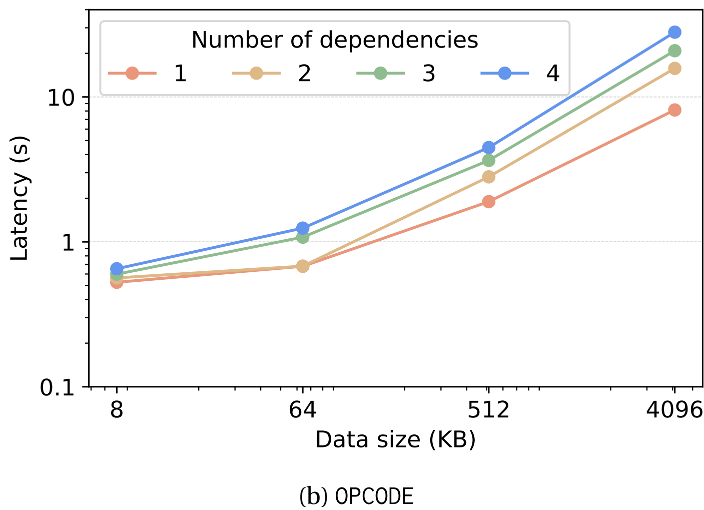
2.6.2 Microbenchmarks

These experiments evaluate the two core tasks that are performed by every DFU as part of executing an opcode: verifying the data dependencies and generating the output receipts. We choose to measure the overhead of these tasks since they are essential to coordinating the execution of smart contracts across DFUs. For the first experiment, we only consider the overhead of verifying the `KEYVALUE` and `OPCODE` dependencies since verifying the `PRECOMMIT` and `CONST` dependencies is trivial and creates negligible overhead. For both experiments, we consider DFUs, both the verifying DFU and the source DFU of the data, with 19 nodes and a collective signature threshold of 13. We choose these values since they are the maximum values for any DFU that we use in our applications.

Figure 2.8a shows the latency results of verifying `KEYVALUE` dependencies. Each DFU node verifies the state proof associated with each `KEYVALUE` dependency. This task involves verifying the inclusion proofs for the input key-value pairs and the collective signatures of the forward links. We vary the number of dependencies and the number of blocks in the blockchain. The maximum values for both parameters are selected to match the maximum values from the applications. Our results show the advantage of using multi-hop forward links in state proofs. Even though the difference between the maximum and minimum number of blocks is 100×,



(a) KEYVALUE



(b) OPCODE

Figure 2.8: Latency of verifying KEYVALUE and OPCODE dependencies

the difference between their verification latencies does not exceed 2×. This is because the multi-hop links enable traversing the blockchain in a logarithmic number of steps. For the same reason, we can observe that there are three levels in the latency values that correspond to block numbers [10, 50], [100, 200] and [500, 1000]. In the first two levels, latencies are especially close to each other since the corresponding state proofs are comparable in size and the number of forward links.

Figure 2.8b shows the latency results of verifying OPCODE dependencies. Each DFU node verifies the collective signature over the output receipt for each OPCODE dependency. To this end, nodes compute the hash of the input data and the hash of the output receipt for each dependency. We vary the number of dependencies and the size of each input data. Our choice of data sizes, which range between 8KB and 4MB, is representative of the opcode outputs that we observe in our applications. Our results show that for smaller data sizes (*i.e.*, 8KB and 64KB), verifying OPCODE dependencies is faster than verifying KEYVALUE dependencies for the *same number of dependencies*. In this experiment, the number of signatures that needs

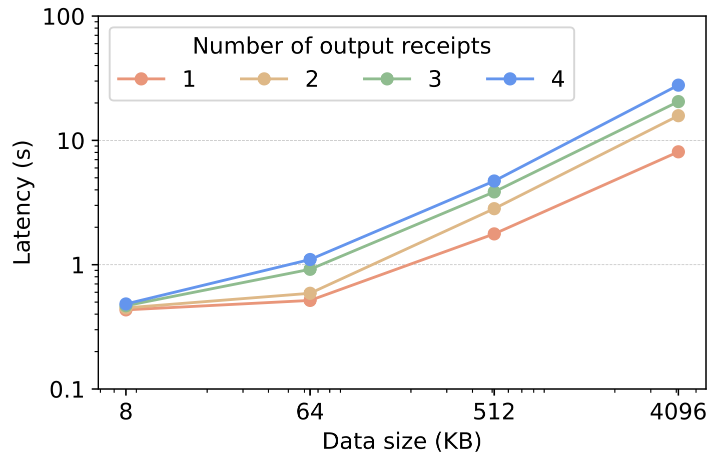


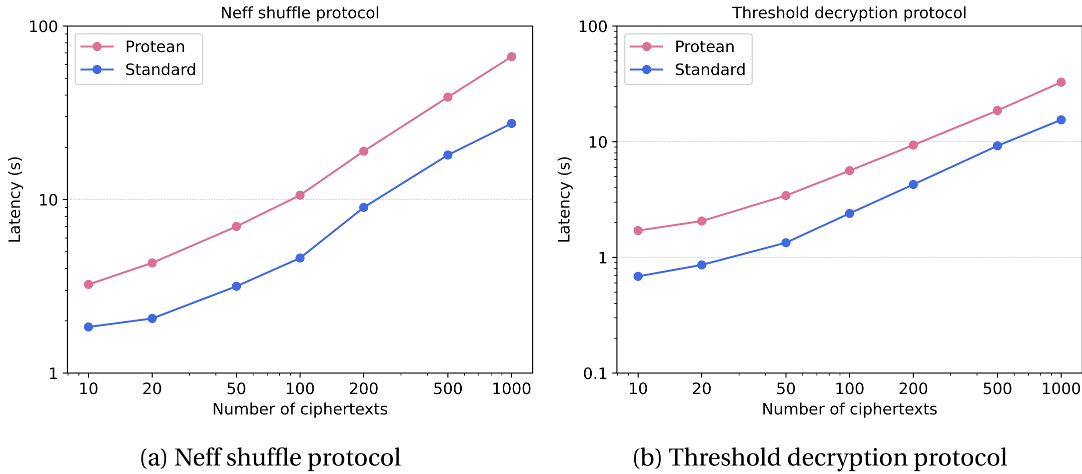
Figure 2.9: Latency of generating output receipts

to be verified equals the number of dependencies, which is at most four. However, in the previous experiment, there are more signatures to verify for every number of dependencies-block number pair, and hence more computational overhead. For larger data sizes (*i.e.*, 512KB and 4MB), we observe the opposite scenario as verifying `OPCODE` dependencies is slower than verifying `KEYVALUE` dependencies for the same number of dependencies. For example, for a data size of 4MB, we observe an order of magnitude slowdown for each number of dependencies compared to the previous experiment. The reason behind these results is that for larger data sizes, transmission delay dominates the latency and the overhead of signature verification becomes negligible.

Figure 2.9 shows the latency results of generating output receipts. Each DFU node individually prepares an output receipt for each output data they have collectively produced and signs the output receipt. Therefore, the main computational tasks are computing hashes and producing cryptographic signatures. We vary the number of output receipts and the size of the output data. We compare the results to the previous experiment since they use the same parameters. For smaller data sizes, where the transmission delay does not dominate latency, we observe that signing is marginally faster than signature verification. DFUs use aggregated BDN signatures to collectively sign the output receipts. Therefore, it is expected that producing a BDN signature is faster than verifying an aggregate BDN signature. Like the previous experiment, for larger data sizes, transmission delay dominates the latency and therefore we do not observe the same pattern consistently.

2.6.3 Modified Neff shuffle and threshold decryption protocols

In this experiment, we quantify the performance penalty caused by our modifications to the Neff shuffle and threshold decryption protocols to make them compatible with PROTEAN. We implement the standard and PROTEAN-style protocols as they are described in §2.5.2. We set



(a) Neff shuffle protocol (b) Threshold decryption protocol

Figure 2.10: Execution latencies for the Neff shuffle and threshold decryption protocols using their standard and modified implementations

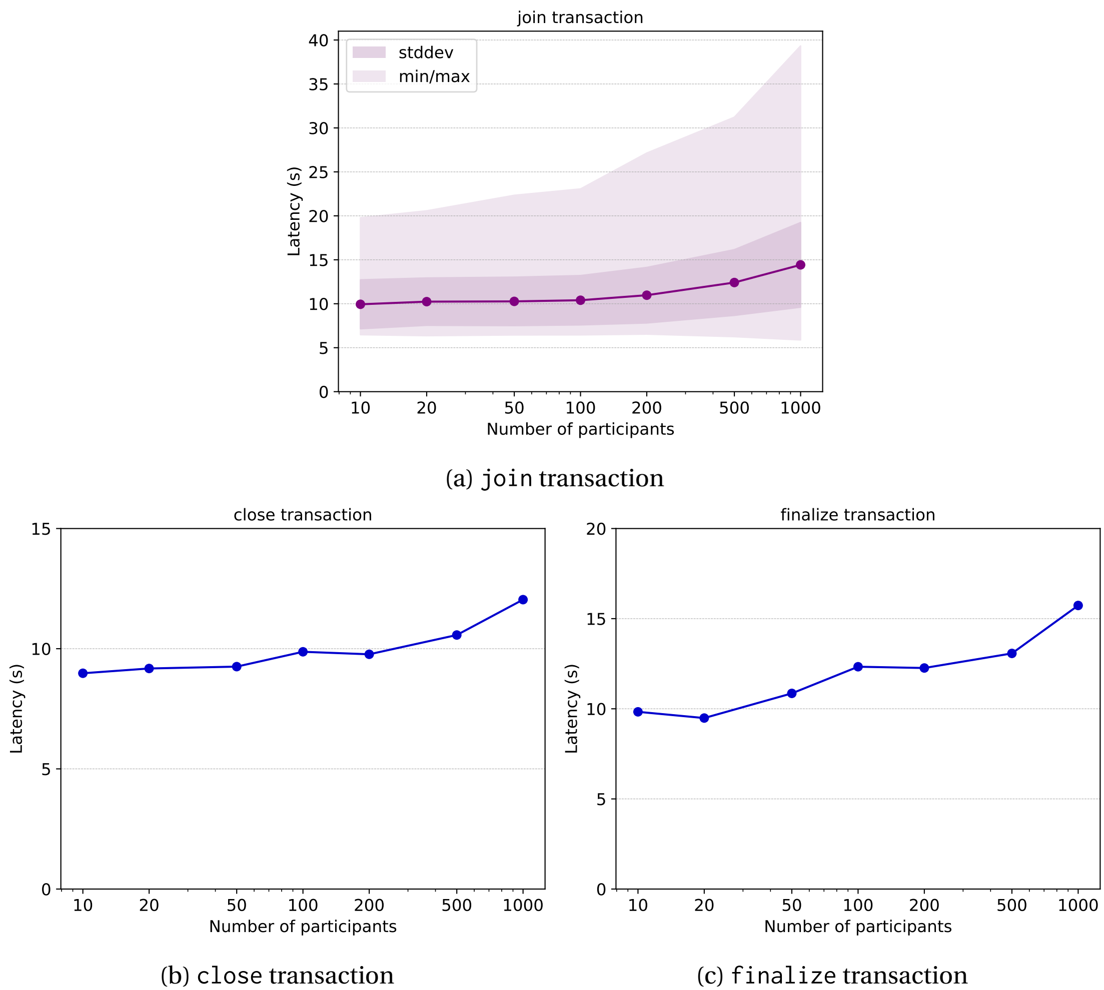
the number of nodes in the mix-net DFU and threshold decryption DFU to 13 and 19, and their collective signature thresholds to 7 and 10. We send a single request that contains a list of ciphertexts to the DFUs. We vary the number of ciphertexts from 10 to 1000. We report the average latency results over 10 experiment runs.

Figures 2.10a and 2.10b show the results for the Neff shuffle and the threshold decryption protocol implementations, respectively. Our modification to the Neff shuffle protocol results in up to 2.44× slowdown when there are 1000 ciphertexts. For the threshold decryption protocol, the slowdown goes up to 2.1× for the same number of ciphertexts. These results are expected as our modifications add a second phase to both protocols that requires coordination between a threshold number of the DFU nodes.

2.6.4 Application experiments

In this experiment, we evaluate PROTEAN 's performance using the three motivating applications, namely `randlotto`, `secretlotto` and `e-voting`, that we described in §2.5.3. We simulate different workloads by varying the number of participants in the applications between 10 and 1000. Assuming n denotes the number of nodes in a DFU and t denotes its collective signature threshold, we use the following (n, t) pairs in our experiment: (19, 13) for the state DFU, (13, 7) for the code execution DFU, (19, 10) for the randomness DFU, (13, 7) for the mix-net DFU, and (19, 10) for the threshold decryption DFU.

An important consideration in generating the workloads is the scheduling of participants' transactions. Since participants can execute the join and vote transactions without having to lock the contract state, if there are multiple participants who read the same contract state and try to concurrently execute their transactions, only one client is going to be able to successfully commit their transaction. The remaining concurrent participants will fail at transaction



(a) `join` transaction

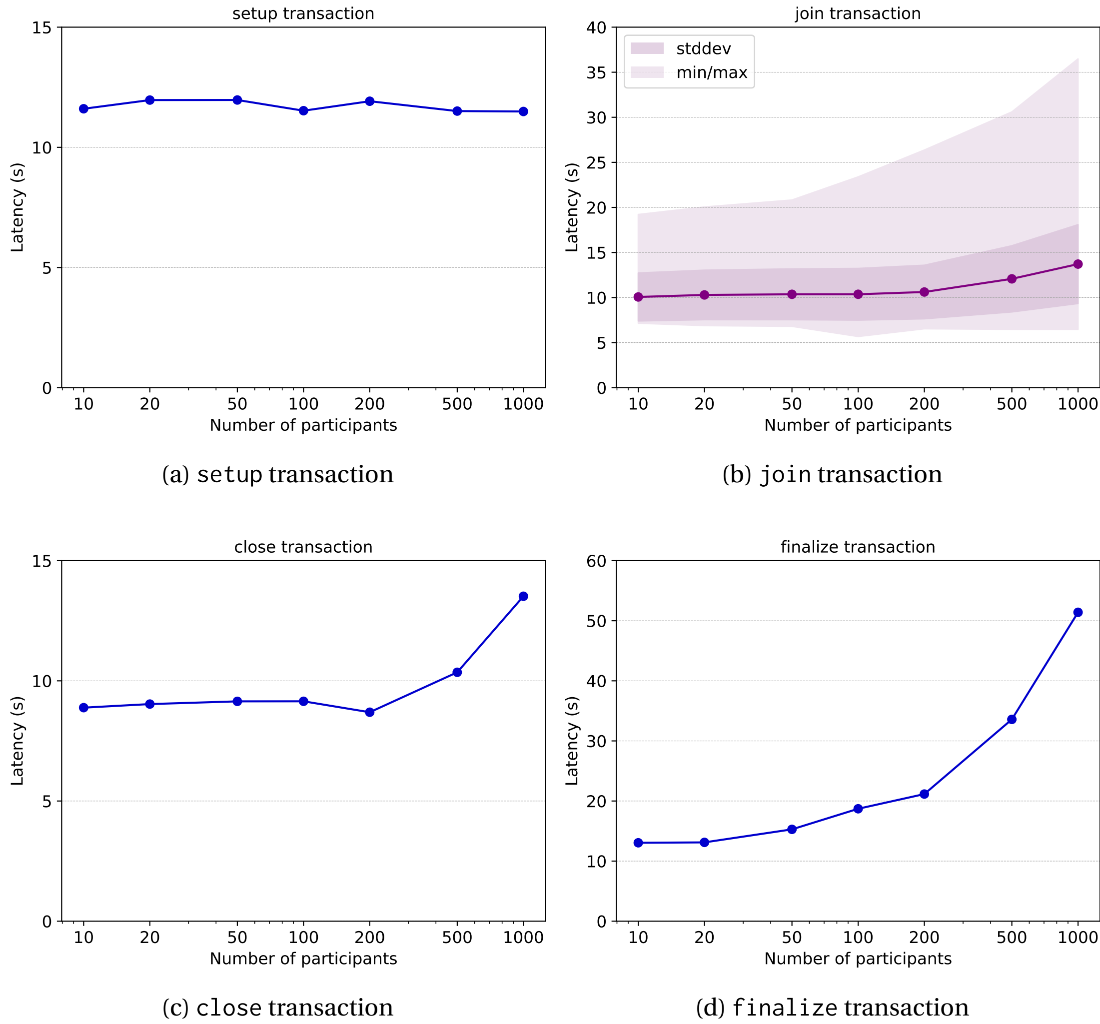
(b) `close` transaction

(c) `finalize` transaction

Figure 2.11: Execution times of `randlotto` transactions

commit and have to re-execute their transaction. In our experiment, we assume a constant retry rate of 0.1, which means that 10% of the participants will fail to commit their transactions at the first try and will be able to successfully commit upon re-execution. Our choice of a conservative retry rate is motivated by a common characteristic of our applications: There is a long period of time (*e.g.*, many hours, days) where the application is open for participants' inputs (*i.e.*, tickets or ballots). Therefore, we assume that it is likely to have a small number of concurrent participants.

Figures 2.11-2.13 show the execution latencies of the transactions of `randlotto`, `secretlotto` and `e-voting`. For the `join` and `vote` transactions, we report the statistics (*i.e.*, avg, min, max, std) by combining the measurements from all experiment runs. Therefore, for these transactions, the number of data points equals to the x-axis value times the number of experiment runs, which is 10. For other transactions, we report the average results over 10 runs. We make the following observations based on our experiment results:



(a) `setup` transaction (b) `join` transaction

(c) `close` transaction (d) `finalize` transaction

Figure 2.12: Execution times of `secretlotto` transactions

1. `join` and `vote` transactions (Figures 2.11a, 2.12b, 2.13b) have similar average latencies as they are composed of the same opcodes, namely `exec` and `update_state`. Moreover, the executed application code in these transactions both have low computational overheads. The average latencies increase as the number of participants increases due to higher computation load and network overhead caused by a longer blockchain and a bigger contract state. For the same reason, we see an increase in both the maximum latency and standard deviation values as the number of participants grows.
2. `secretlotto` and `e-voting` use `setup` transactions (Figures 2.12a and 2.13a) to run the DKG protocol at the threshold decryption DFU to create a collective public key. Both transactions contain the same three opcodes and the computations in these opcodes are independent of the number of participants and the size of the contract state. For example, execution time of the DKG protocol depends on the number of nodes in the threshold decryption DFU, which is the same for both applications. Therefore, we

observe similar latencies for these transactions. Similarly, close and lock transactions (Figures 2.11b, 2.12c, 2.13c) exhibit comparable latencies as they contain the same opcodes whose execution latencies do not depend on the number of participants. That being said, since the size of the on-chain data increases with the number of participants, and therefore the state proofs become larger, we see a small increase in the execution times of these transactions when more participants are involved.

3. shuffle and tally transactions of e-voting (Figure 2.13d and 2.13e), and finalize transaction of secretlotto (Figure 2.12d) are the most computationally-intensive transactions as they are $3-6\times$ slower than the other transactions for the maximum number of participants. These results are due to the two computationally-intensive computations, namely the Neff shuffle in shuffle, and threshold decryption in tally and finalize, and are consistent with our findings in §2.6.3.
   Additionally, we can observe the impact of state proof sizes by comparing Figure 2.11c and Figure 2.13e. As we mentioned in §2.5.2, a state proof includes all the key-value pairs stored in a contract. The additional storage overhead due to the shuffled ballots and shuffle proofs in e-voting causes larger state proofs, which increases the network load. Therefore, even though tally and finalize transactions are composed of the same opcodes that contain lightweight computations, tally is $1.3\times$ slower than finalize for 1000 participants.

Batching optimization

In our evaluation of the applications, we assume that the number of clients that try to concurrently execute the join and vote transactions are low, which reduces the number of transaction aborts and retries. In this experiment, we consider an optimization that can increase the throughput of join and vote transactions. We assume that a client who wants to participate in one of these applications submits their input to a *proxy client* instead of executing the join or vote transaction. The proxy batches the inputs from many participants and commits them to the contract state in a single transaction. In this way, we can avoid aborts due to concurrent transactions and achieve a higher throughput.

Figures 2.14-2.16 show the execution latencies for our applications with batched join and vote transactions. We assume a fixed batch size of 10. We note that all the transactions except for vote and join are executed once as in the previous experiment. Our results show improved execution latencies for join and vote transactions. These results are expected since there are no aborts and retries due to concurrent clients.

2.7 Related work

Privacy-preserving smart contracts: A long line of work improves the privacy of data and computations on smart contracts by performing computations off-chain and verifying the proofs on-chain. Unfortunately, these solutions are either domain-specific and do not gener-

alize to all decentralized applications [15], [29], have limited expressiveness [27], [186], relies on a trusted party for a privacy or security guarantee [38], [129], [137], or have significant performance overheads [185], [220].

Oracle networks: Oracles try to address a particular shortcoming of smart contracts: reliably and securely accessing to data about the real-world. TownCrier [221] is one of the earliest examples of oracle solutions for smart contracts. However, since it builds on top of trusted hardware, it is essentially relies on a trusted party. Deco [222] solves the problem of proving the authenticity of data that is accessed via TLS came from a particular website. Finally, Chainlink is an exciting system that have certain architectural similarities with PROTEAN. Chainlink uses Decentralized Oracle Networks (DONs) that offer off-chain computing resources to smart contract. Chainlink's VRF solves the problem of parsing verifiable, unpredictable, and unbiasable randomness in Ethereum smart contracts without relying on a trusted party. Chainlink's OCR protocol can also be used for providing smart contracts with a rich set of computations.

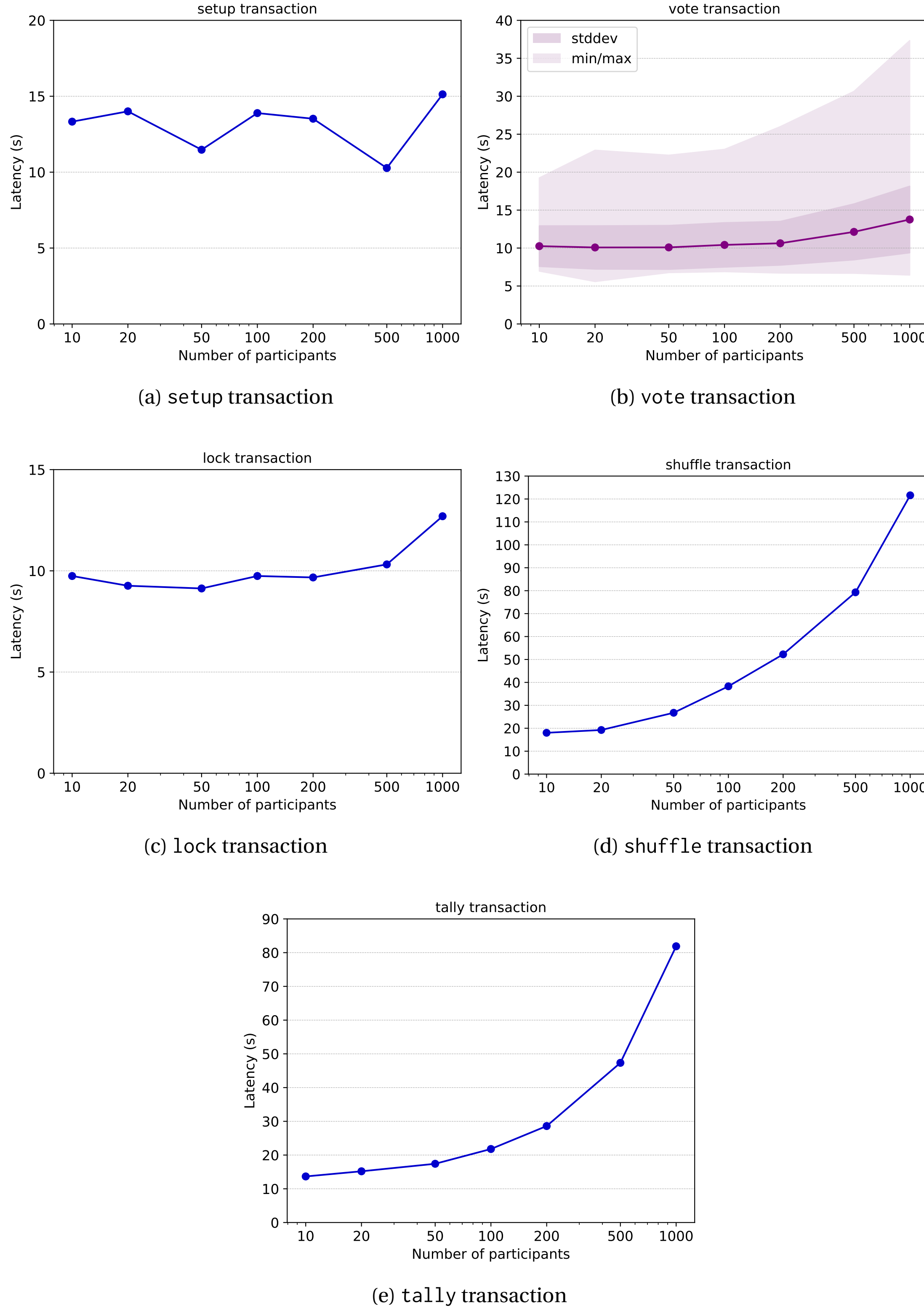
Extensible blockchains: Aspen [90] introduces service-oriented sharding builds on a multi-blockchain structure where each blockchain exposes a different service and stores only the transactions that belong to its service. In addition to high scalability, Aspen also achieves extensibility by enabling users to introduce new services without disrupting the operation of others. However, Aspen does not support smart contracts. Hyperledger [4] is a modular and extensible smart contract platform for building distributed applications. The core idea of Hyperledger is separating transaction execution and validation from consensus, which is similar to PROTEAN's separating of consensus from computing. Hyperledger uses multi-version concurrency control (MVCC) to concurrently execute transactions. However, it is not clear how Hyperledger handles aborted transactions that involve irreversible operations.

Data-flow graphs: Using data-flow graphs for performing tasks is a common pattern in many systems. Dryad [122] and CIEL [157] provide distributed execution engines for executing data-parallel tasks using data-flow graphs. in-toto [201] is a framework that ensures the integrity of software supply chains. Developers prepare layouts to declare the steps that needs to be performed in the supply chain and how different steps interact with or depend on each other. Layouts define data-flow graphs that resemble the transactions in PROTEAN. When a step is performed, the entity who performed that step produces a cryptographically signed statement to prove that the step is performed, which is similar to our output receipts.

2.8 Conclusion

We have presented PROTEAN, a permissionlessly-extensible framework for general-purpose decentralized computing. PROTEAN can support a richer set of functionality compared to

current smart contract platforms (*e.g.*, Ethereum) thanks to its permissionless design that enables easy and modular addition of specialized functionality via special-purpose standalone distributed systems called decentralized functional units (DFUs). We introduce a workflow-based programming model with cryptographically-secured data structures to define and coordinate the execution steps of smart contracts across multiple DFUs. The evaluation of our prototype using two classes of real-world decentralized applications has shown that PROTEAN can support applications that are currently not possible or feasible to implement with Ethereum-style contracts without incurring large performance penalties.



(a) `setup` transaction

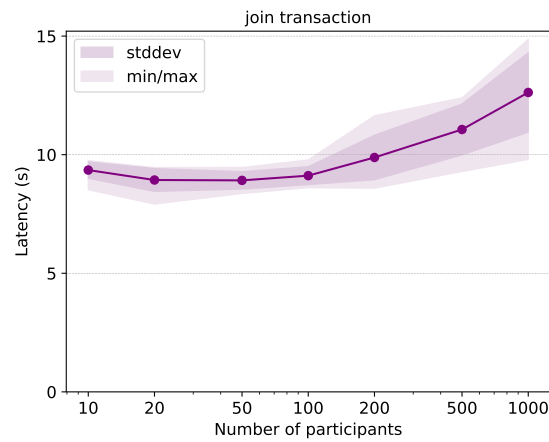
(b) `vote` transaction

(c) `lock` transaction

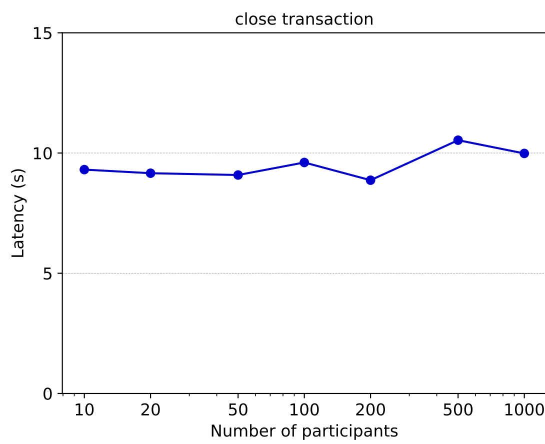
(d) `shuffle` transaction

(e) `tally` transaction

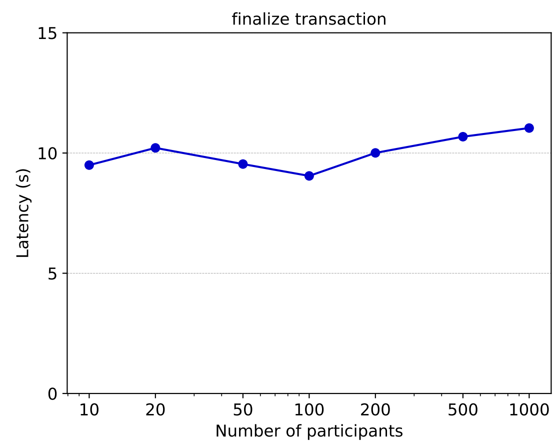
Figure 2.13: Execution times of `e-voting` transactions



(a) `join` transaction

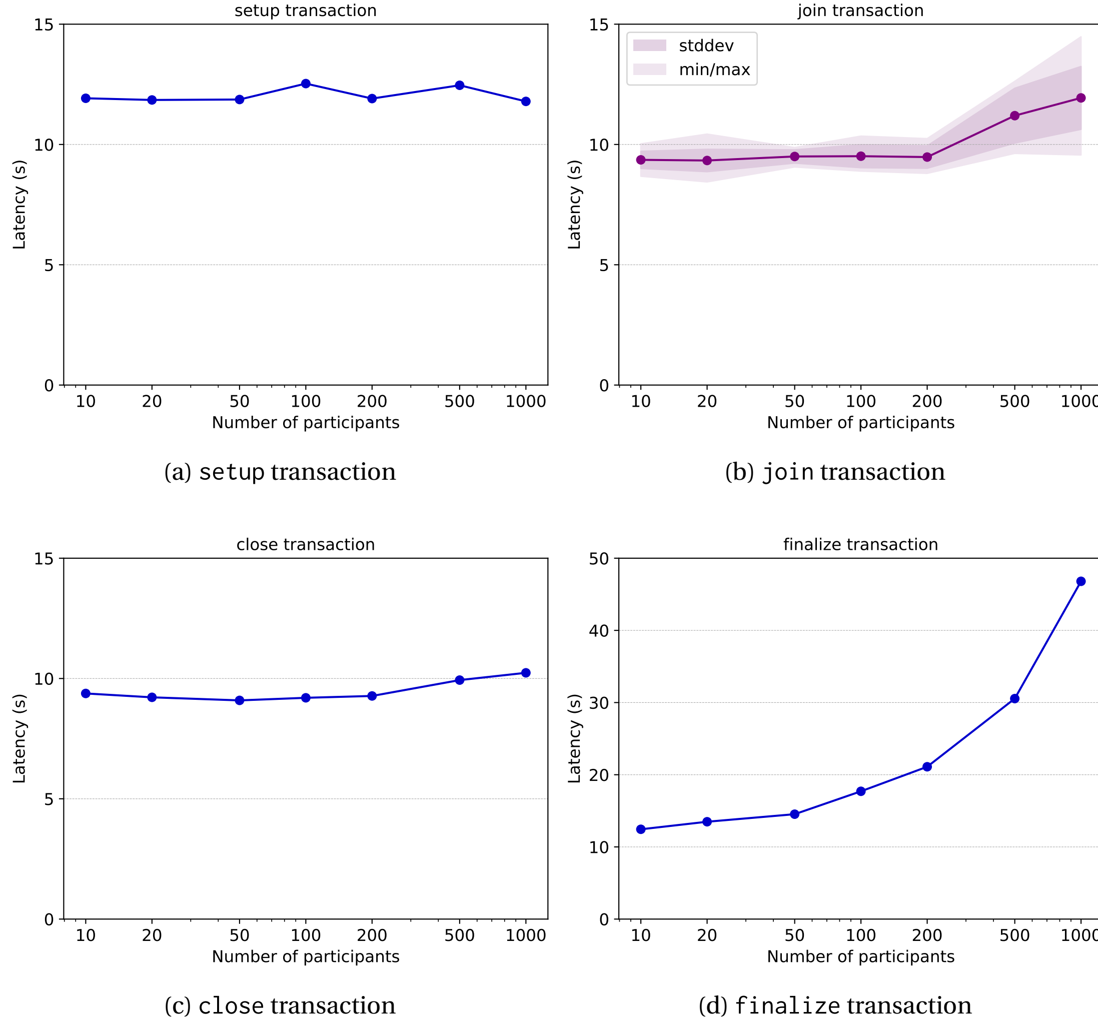


(b) `close` transaction



(c) `finalize` transaction

Figure 2.14: Execution times of `randlotto` transactions with batching



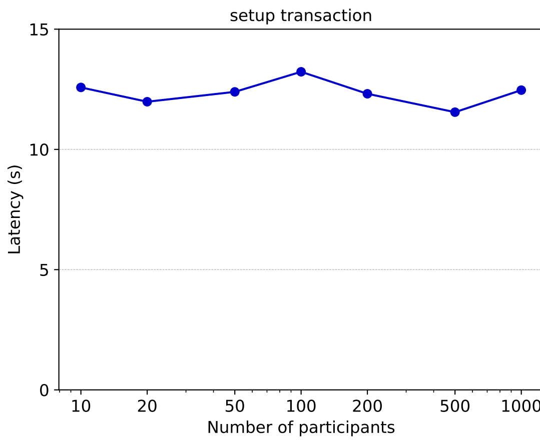
(a) `setup` transaction

(b) `join` transaction

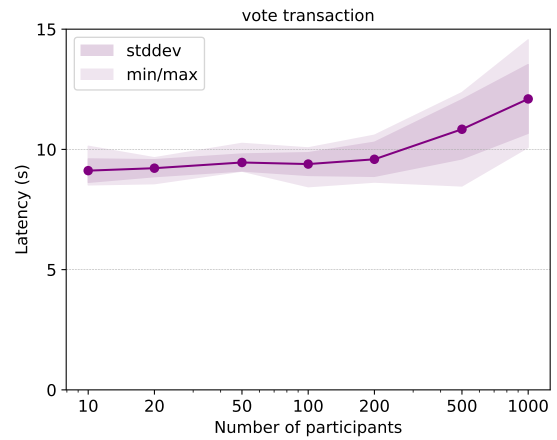
(c) `close` transaction

(d) `finalize` transaction

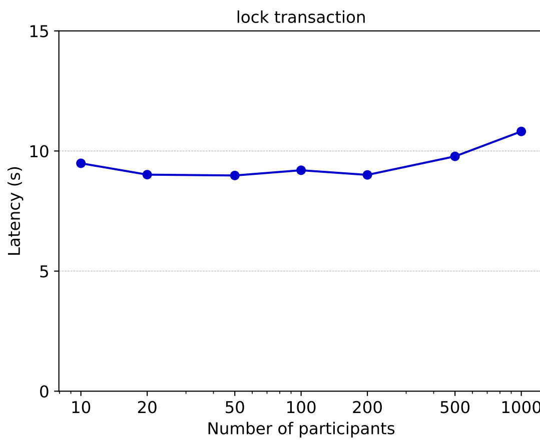
Figure 2.15: Execution times of `secretlotto` transactions with batching



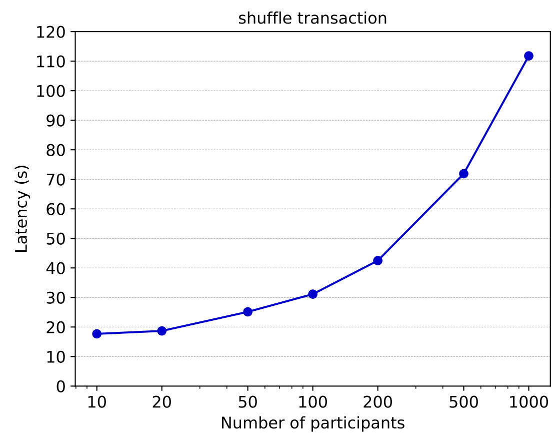
(a) `setup` transaction



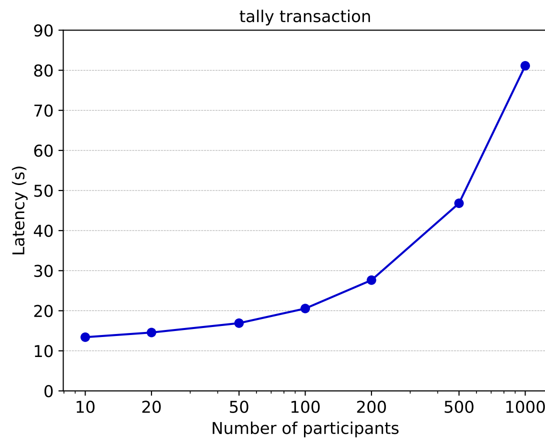
(b) `vote` transaction



(c) `lock` transaction



(d) `shuffle` transaction



(e) `tally` transaction

Figure 2.16: Execution times of `e-voting` transactions with batching

3 DEJAFLOAT: A Deterministic JVM Sandbox with Floating-point Arithmetic Support

This work is part of a larger collaborative determinism project that other PhD students and research engineers from the DEDIS lab participated in as well. This thesis chapter focuses primarily on my contributions to that project.

3.1 Introduction

Deterministic execution guarantees that, given the same input, a program produces the same output across different executions. The ability to deterministically execute software simplifies program debugging [89], [206], testing [18], [48] and program analysis [41]. Additionally, enforcing deterministic execution of untrusted programs is essential for the correctness and security of fault-tolerant systems [34], [44], [143], prevention of timing channel attacks [10], [84], [218], and intrusion analysis [65].

More recently, smart contracts have emerged as one of the most prominent systems that have strict determinism requirements due to the underlying consensus protocols. To be able to reach consensus on the state of smart contracts, each node in the network have to obtain the same output upon executing the same smart contract function with the same inputs. Non-deterministic execution of smart contracts can break consensus as nodes can fail to agree on the same program state, thereby violating the consistency and safety of the system.

Deterministic code execution is required not only by Ethereum-style smart contract platforms where consensus and computing are tightly coupled, but also by our novel smart contract platform PROTEAN, which separates consensus from computing. Even though PROTEAN can support specialized computations via DFUs that use different techniques such as secure multi-party computation or zero-knowledge proofs to guarantee the correctness of a computation, it still requires DFUs (*i.e.*, code execution DFUs) that support replicated code execution to be able to run application-specific programs.

One of the most challenging sources of non-determinism in programs is floating-point arithmetic. The IEEE 754 standard [105] specifies formats and methods for floating-point arithmetic and has become the de-facto standard for many hardware and software systems. However, the standard does not specify every aspect of floating-point arithmetic, and therefore it allows for different conforming implementations [95], [105]. For instance, IEEE 754 specifies a set of valid bit-patterns of "not a number" (NaN) values but it is up to the hardware architecture to decide which particular bit pattern is used. Additionally, propagation of NaN payloads are also left to the implementations [83]. Transcendental functions and extended-precision formats are other well-known reasons why programs can produce different results on conforming systems [95], [151].

Systems that require determinism for correctness and security often forgo floating-point arithmetic altogether or only support a limited subset of the operations. In the context of smart contracts, Ethereum, which is the largest smart contract platform, uses a deterministic runtime environment called Ethereum Virtual Machine (EVM) that does not have native floating-point support. A WebAssembly (WASM) [99], [210] based execution engine called Ethereum WASM (eWASM) [77] is designed to replace EVM to improve the performance, scalability, and programmability of smart contracts. Even though WASM has floating-point support, it is not deterministic due to NaN payloads [163], [211]. Therefore, eWASM disables floating-point support in the interest of full determinism. Other popular smart contract platforms such as EOSIO [106], Solana [164], and Algorand VM [195] support either fixed-point arithmetic or a limited set of floating-point operations via software libraries.

We present DEJAFLOAT, a Java Virtual Machine (JVM) based sandbox that enforces deterministic execution of untrusted Java programs. By providing a restricted runtime environment, DEJAFLOAT guarantees that programs can only have deterministic executions in the sandbox. As part of the sandbox, DEJAFLOAT deterministically supports floating-point arithmetic with minimum performance penalty. We address the following challenges to build DEJAFLOAT: (1) identifying the deterministic methods and classes in the Java standard libraries, (2) eliminating the sources of non-determinism from floating-point operations (*e.g.*, NaN payloads, transcendental functions, extended precision formats) with minimum performance overhead, (3) securing the sandbox against malicious or buggy programs that can have infinite executions or finite executions with high resource consumption.

We address the first challenge by carefully reading the specifications and source codes of the Java standard libraries to determine classes and methods that are deterministic. We create a deterministically-safe subset of the Java standard libraries and restrict the programs in our sandbox to use only these libraries. We make the floating-point operations deterministic by: (1) canonicalizing NaNs at escape points where their bit patterns can become visible, (2) using libraries that produce correctly-rounded results for transcendental functions, and (3) relying on Java's strict floating-point semantics to avoid precision errors due to extended precision formats. We address the final challenge by limiting the number of Java bytecode instructions that can be executed as part of a program in our sandbox.

We implemented a prototype of DEJAFLOAT and evaluated its performance using a combination of microbenchmarks and prototyped applications. Our microbenchmarks show that the overhead of canonicalizing NaNs is almost negligible thanks to the strong type system of Java. To demonstrate the performance benefit due to the strong type system of Java, we implement NaN canonicalization in WASM, which has an untyped linear memory model, and compare its performance to our prototype. Our experiment results show that canonicalizing NaNs in WASM can result in up to a 1.5× slowdown. We evaluate the cost of enforcing deterministic transcendental functions in DEJAFLOAT using two machine learning applications, namely image classification and recommendation system. Our results show that enforcing determinism via correctly-rounded floating-point arithmetic libraries can cause a 2× slowdown compared to Java math libraries. However, we show that pure Java libraries can be a feasible alternative for applications that prefer performance over correctness without sacrificing deterministic execution. Finally, we show that by using a JVM bytecode instrumentation technique, we can efficiently bound the execution of untrusted programs to protect our sandbox from malicious programs.

Our contributions are the following:

- The design and implementation of DEJAFLOAT, a JVM-based sandbox that can enforce deterministic execution of untrusted Java programs and support deterministic floating-point operations.
- An experimental evaluation of DEJAFLOAT's performance using two real-world machine learning applications that rely heavily on floating-point arithmetic.

3.2 Background and motivation

3.2.1 The IEEE 754 standard

The IEEE 754 standard [105] is the most widely-used standard for implementing floating-point arithmetic in software and hardware systems (*e.g.*, programming languages, compilers, microprocessors). The standard specifies a floating-point number by four parameters: a base (b), a sign (s), an exponent (e), and a significand (m). The significand, which is also called the mantissa, is a digit string in base-b with length p. The length of the significand is called the *precision* and it determines the number of significant digits that can be represented in a format.

The standard defines five basic formats for representing floating-point numbers in binary and decimal bases using different bit lengths for their encodings. An encoding consists of three fields: the sign bit, the exponent field, and the trailing significand field. Bit lengths of the exponent and trailing significant field depend on the format of the representation. We focus on the two most common binary formats for representing floating-point numbers: 32-bit single precision (binary32) and 64-bit double precision (binary64). These formats have 24 and 53 bits of precision, respectively.

Since it is not possible to exactly represent all real numbers as floating-point numbers, they need to be rounded so that they can be represented in a valid format. IEEE 754 defines five rounding modes that determine how the rounded result is calculated. The default mode is rounding to the nearest floating-point value. Finally, the IEEE 754 standard requires the results of the five basic arithmetic operations (*i.e.*, addition, subtraction, multiplication, division, and square root) to be *correctly rounded*. A function is correctly rounded for a given rounding mode if it produces the same rounded result that would be obtained by rounding the result of the same function computed with infinite precision. An implication of correct rounding is that there is a bound on the maximum rounding error, which is commonly measured by *unit in the last place (ulp)*. The original definition of ulp by Kahan defines `ulp(x)` as "*the gap between the two floating-point numbers nearest x, even if x is one of them*" [128]. Based on this definition, in the rounding to the nearest mode, a correctly rounded result is within 0.5 ulp of the exact result. Correctly rounded operations are deterministic by default since there is only one possible correctly rounded result.

3.2.2 Why is floating-point arithmetic non-deterministic?

Unfortunately, there are situations in which the IEEE 754 standard does not guarantee that the same floating-point operation produces the same result across all compliant implementations [95], [151]. In the case of smart contract systems, which is our main motivating application with strict determinism requirements, getting different results on different systems can cause security vulnerabilities by breaking consensus: All the nodes running on one architecture might decide that one block is correct, and another set of nodes running on another architecture might decide that another block is correct. In this section, we discuss three main sources of non-determinism in IEEE 754 floating-point arithmetic:

Transcendental functions. A transcendental function is a non-algebraic functions that does not satisfy a polynomial equation. The most common transcendental functions are exponential, logarithmic and trigonometric functions. Since transcendental functions cannot be exactly computed in a finite number of steps, they are instead approximated. To avoid results with poor accuracy, it would be ideal to use the correctly-rounded implementations of these functions.

However, correct rounding of transcendental functions is difficult because of the Table Maker's Dilemma [128], which is the fact that there is no way to know how much precision is required for the computation to obtain the correctly-rounded result and that the computation can take an arbitrarily long time. Therefore, IEEE 754 only recommends and does not require correct rounding of transcendental functions so that conforming systems can support implementations with lower accuracy but better performance. As a result, IEEE 754 compliant systems support transcendental functions via different software libraries or hardware implementations, which causes getting different results on different systems [53], [95].

NaNs. Not a Number (NaN) is a special floating-point value that is used for representing the result of an invalid arithmetic operation, such as 0/0 or $(+\infty) - (+\infty)$. There are two kinds of NaNs: a signaling NaN (sNaN) raises an exception as a result of an invalid operation and a quiet NaN (qNaN) that propagates through operations without raising any exception.

The IEEE 754 standard specifies the encoding of NaNs as binary bit strings as follows: all bits in the exponent field are ones, the significand field contains at least one non-zero bit, and the sign bit can either be a zero or one. The first bit of the significand field of a qNaN (sNaN) is a one (zero).

Since there are many different bit strings that can be a valid encoding of a NaN, NaNs are a source of non-determinism in floating-point arithmetic. For example, implementations might not give guarantees about the value of the sign bit [108]. Another source of non-determinism is due to the propagation of NaN payloads. The standard specifies that an operation with one NaN input should output a NaN with the same payload as the input NaN. However, if the operation has multiple NaN inputs, then the output NaN can have the same payload as any one of the input NaNs. Therefore, different implementations can have different behaviors in propagating NaNs [83], [109].

Extended precision. IEEE 754 specifies extended precision formats that extend the basic formats to support greater precision and range. For example, the double-extended format has at least 64 precision bits, which is 11 bits more than the double-precision format. The motivation behind supporting extended precision is to perform intermediate computations with higher precision so that premature overflows/underflows can be avoided and final results can be computed with higher accuracy [95].

The downside of using extended precision is that it can lead to non-deterministic behaviors. For instance, the x87 floating-point unit (FPU) of Intel uses 80-bit registers (64-bit precision) to perform floating-point operations in the double-extended format. When the value in an x87 register has to be stored in memory, it is rounded down to single- or double-precision format. However, it is up to the compiler to decide whether temporary values of intermediate computations will be stored in the registers or spilled to memory. Therefore, a computation can produce different results on the same hardware architecture depending on the compiler version, options, and optimizations [46], [52], [219].

Use of extended precision can also cause non-determinism due to *double rounding* [151], [155], which is rounding a floating-value twice in a decreasing level of precision. For example, in x87, the result of a floating-point operation is first rounded down to 80-bits, then to 64-bits to be stored in a double-precision variable. In contrast, in a system that does not support extended precision, there is only single rounding to the precision of the variable. Since double rounding may affect the final result, the same operation can produce different results across different hardware architectures.

3.2.3 Motivating applications

Smart contracts

One of the most prominent applications that has strict deterministic execution requirements is smart contracts. A smart contract is an arbitrary user-defined program that is stored and automatically executed on top of a blockchain network. Blockchain nodes (miners) run a distributed consensus algorithm to execute the smart contract. To reach consensus, each miner individually executes the contract code and computes the output of the execution. If a quorum of miners obtain the same output, then the contract state is successfully updated by the miners based on the result of the execution. Therefore, execution of smart contracts has to be deterministic across all miners, which can potentially run on different software/hardware stacks, to guarantee the correctness of smart contracts.

An important consideration in enforcing deterministic program execution is addressing floating-point non-determinism. Ethereum Virtual Machine (EVM) [217], which is Ethereum's execution environment, does not natively support floating-point arithmetic. In the future, Ethereum is planning to transition from EVM to a deterministic subset of WebAssembly (WASM) called Ethereum WASM (eWASM) [77]. Since WASM does not address non-determinism due to NaNs, the design of eWASM leaves out floating-point arithmetic support.

Other smart contract platforms such as EOSIO and Solana take a different approach than Ethereum. EOS VM is a WASM engine that uses a software implementation of IEEE 754 to deterministically support a small subset of floating-point operations [106]. Solana supports floating-point operations via LLVM's float built-ins, which is also implemented in software, and also fixed-point operations [164]. Unfortunately fixed-point arithmetic has less precision than floating-point arithmetic.

Reproducible scientific computing

Reproducibility is one of the major tenets of scientific computing [167], [225]. However, non-deterministic floating-point arithmetic represents a major challenge in achieving reproducibility in scientific computing applications, such as climate modeling and simulation, long-term simulations of the solar system, and dynamical systems [6], [13], [39], [156]. Reproducibility has also become a major concern for machine learning (ML) applications [37], [144], [158]. ML algorithms are widely deployed for making automated decisions in various domains, such as education, healthcare, and employment. Since the decisions made by these algorithms can have significant sociological and economical consequences, it is essential that the algorithms are transparent and accountable [47].

Even though reproducibility of scientific computations is a desirable property, there are certain cases where bit-for-bit reproducibility might be undesirable or provide a stronger guarantee than what we want. For example, if a simulation is numerically unstable, then the output might be meaningless because it is mostly the result of many-times-amplified

roundoff errors throughout the calculation without carrying any "real" information. Bit-for-bit reproducibility would obscure the results of numerically-unstable calculations, since re-running the unstable computation would produce the exact same meaningless result. However, with inexact reproduction (*e.g.*, allowing different seeds for random numbers the computation might use, different schedules for parallel computations that might affect the result, *etc.*), the instability might become apparent more immediately because each run of the calculation would return a wildly different result, as opposed to "almost the same" results.

Despite this limitation, however, bit-for-bit reproducibility is still a very useful tool to support scientific reproducibility. In particular, it gives users control over when we want "exact reproduction" and when we want to explore semantically-equivalent (but potentially numerically distinct) variations. For example, if the relevant calculation incorporates randomness in any way, like many do, then we can use bit-for-bit reproducibility with the same random seed when we want exact reproduction of a prior calculation. But when we want to test numerical stability or otherwise explore similar-but-not-exactly-identical runs, we can simply re-run the calculation several times with different seeds.

An efficient runtime environment with deterministic floating-point arithmetic can provide control over when to reproduce identically and when to inject variation into calculations. The particular run with any particular seed and input set is precisely reproducible, but we can get as many different variations as we might want for exploring the numerical result space. This can facilitate the verification, testing, and debugging of scientific computing applications. Furthermore, being able to reproduce the results of decision-making algorithms can enable auditing these algorithms and bring transparency to the decision-making processes.

Decentralized ML

A deterministic sandbox that can provide smart contracts with deterministic and efficient floating-point arithmetic support can improve the accountability and trustworthiness of machine learning (ML) applications. For example, a third party service can train an ML model on participants' aggregated input data within our sandbox. Later, any participant who wants to verify that the model was correctly trained by re-running the training phase in the sandbox. Therefore, our deterministic sandbox can bring accountability to ML training. Similarly, participants can verify that the output of a particular query on the trained model is correct by re-running the prediction stage, thereby ensuring the trustworthiness of the predictions. As a result, many decentralized finance (DeFi) applications, such as automated market-making, lending protocols, and asset management [81], [112], [170], [224], can benefit from running predictive ML models on smart contracts. Additionally, combining smart contracts and ML algorithms can create decentralized and collaborative AI marketplaces [101], [102], [224] that can collect large amounts of data from various sources and achieve better models thanks to the incentivized participation of many contributors.

3.3 Overview

3.3.1 Challenges

We first discuss the challenges involved in building a deterministic sandbox that can run untrusted general-purpose programs:

How to enforce deterministic execution: The first challenge is determining the techniques that we want to employ enforce deterministic execution of programs in our sandbox. *System-level techniques* can enforce deterministic native execution of programs written in different programming languages at runtime. For instance, these techniques can be used for enforcing the deterministic execution of multithreaded programs [9], [17], [56], [57] or preventing timing attacks [139], [218]. The downside of these approaches is that they can have significant runtime costs. *Language-level techniques* trade off the generality of the system-level techniques for avoiding runtime overheads [82], [165]. These techniques are based on restricting the primitives and features of a particular programming language so that it does not allow non-deterministic executions. These approaches are lightweight due to statically enforcing determinism, but they only work for a particular language. A final approach, *virtual instruction set architecture (ISA)-level techniques*, operates above the hardware instruction architecture sets (*e.g.*, x86, ARM) and below the programming languages [17], [20]. For example, they can enforce determinism at the instruction level in architectures such as JVM [140], WASM [99], [210], LLVM IR [142], or EVM [217]. These techniques can generalize better than language-level techniques without incurring the high overheads of the system-level techniques.

Supporting deterministic floating-point arithmetic: The second challenge is to provide programs with deterministic floating-point arithmetic support in our sandbox. As we have discussed in the previous section, there are different ways in which floating-point operations can exhibit non-deterministic behavior. The challenge is to prevent programs from observing these behaviors without restricting the operations that can be used by the programs and without incurring high performance overhead. More specifically, we need to make sure that the bit patterns of NaNs become observable in a controlled manner so that the platform-dependency of NaN values do not affect the programs. Additionally, we need to guarantee that transcendental functions produce reproducible results within our sandbox independent of the platform on which our sandbox runs. Finally, we need to eliminate non-determinism due to extended precision by enforcing all computations to be rounded to the single or double precision.

Limiting program execution costs: The third challenge is to protect the sandbox from malicious or buggy programs that execute infinitely or that execute in a finite amount of time but include lengthy or computationally-intensive operations. If there is no mechanism to limit the execution time or resource usage of a smart contract in a deterministic way, a program

can cause a denial-of-service (DoS) attack by using up the resources of a sandbox and prevent it from servicing other programs. Ethereum solves this problem by charging a fee for every instruction that is executed in a smart contract. Each EVM opcode has a fixed and deterministic fee, which is referred to as *gas*. The user has to pay enough gas fee to cover the computational cost of executing a transaction. If the user does not specify enough gas for their transaction, EVM halts the execution when it runs out of gas. We need a similar mechanism that enables defining halting conditions in a deterministic and flexible manner. Moreover, this mechanism should introduce as little runtime overhead as possible.

Supporting applications with different requirements: Our sandbox is designed to enforce deterministic execution of untrusted programs and to operate in malicious settings, both of which are exemplified by our main motivating use case of smart contracts. However, as we have discussed in the previous section, there are many other applications that can benefit from our sandbox. Since these applications can have different computational requirements and security assumptions than smart contracts, our sandbox should have the flexibility to cater the needs of different applications. For example, if a ML application needs to access a source of randomness in our sandbox, it should be able to do so at its own risk. In this scenario, our sandbox cannot guarantee that the consecutive executions of this program will produce the same results if the programmer has not written the application code carefully (*e.g.*, forgetting to fix the seed of the random number generator).

3.3.2 System goals

DEJAFLOAT aims at achieving the following goals:

1. *Deterministic execution:* The sandbox enforces the deterministic execution of untrusted Java programs across all underlying software/hardware stacks and implementations of DEJAFLOAT.
2. *Limiting program execution costs:* The sandbox deterministically prevents a program from executing infinitely. To do so, the sandbox enables measuring the duration of program execution and limiting it to a specific deterministic execution quota.
3. *Efficient general-purpose computation:* The sandbox should support commonly-used programming constructs and primitive types to make sure that a wide variety of applications can be run in the sandbox.

3.3.3 Design rationale: Java and JVM

Before we dive into the details of DEJAFLOAT's architecture, we first discuss our decision to build a deterministic sandbox on top of Java and JVM. While doing so, we focus on the features of Java and JVM that help us addressing the challenges and achieving the system goals that we have presented above.

First, building our sandbox on top of Java and JVM enables us to use both language- and virtual ISA-level techniques to enforce determinism, thereby striking a good balance between performance and flexibility. Since Java is a mature and well-defined language [97], [140], it simplifies identifying the well-known sources of non-determinism in its language constructs and standard libraries. We demonstrate how this information helps us building a static checker to eliminate non-determinism without introducing runtime overheads (§3.4.1). While the static checker is a good start, since it operates only at the source-code level and before the program execution, we cannot rely solely on it to achieve our system goals. Therefore, we also use JVM-level bytecode instrumentation to efficiently limit the execution cost of a program (§3.4.3) and to enforce programs to use deterministic floating-point operations in a way that is transparent to the programmers (§3.4.2). Second, both Java and JVM have desirable features that facilitate deterministic support of floating-point arithmetic due to their primary design goals of portability and safety [67], [98]. We make use of these features to efficiently eliminate non-determinism due to NaNs and extended precision (§3.4.2). Finally, since Java is one of the most popular programming languages, there is a large ecosystem of libraries and development tools that are available to the developers and can be supported by our sandbox. We utilize this ecosystem for addressing non-determinism due to transcendental functions (§3.4.2).

We have also explored WASM as an alternative to Java/JVM for the purposes of building a deterministic sandbox. Since WASM is designed for serving as a compilation target for various programming languages, it can support a diverse set of languages (*e.g.*, C/C++, Rust) that JVM cannot readily support. However, WASM has a flat memory model where programs load and store values from/to an untyped array of bytes. This is in contrast with Java/JVM that is strongly typed at the bytecode level, which is a useful property for enforcing deterministic execution. In §3.4.2, we discuss how the strong typing of Java/JVM enables us to efficiently eliminate non-determinism due to NaNs. In §3.6.2, we experimentally evaluate the overhead of eliminating NaN non-determinism in both Java/JVM and WASM to demonstrate the benefit of strong typing of Java/JVM.

3.4 DEJAFLOAT Architecture

In this section, we describe how DEJAFLOAT combines static analysis and JVM bytecode instrumentation to create a deterministic sandbox with floating-point arithmetic support.

3.4.1 A static checker for language-level determinism

We build a static checker that inspects the source code of a Java program to confirm that it only uses whitelisted Java classes and methods that are known to behave deterministically. We curate a whitelist of Java classes and methods by manually sifting through API documentations, source code, and the Java language and JVM specifications. Since the Java runtime environment comes with hundreds of classes and thousands of methods, the search space is prohibitively large to manually check every package and class. Therefore, our whitelist is

limited to the classes and methods that are defined in `java.lang`, which contains the classes that are fundamental to Java, and `java.util`, which provides widely-used utility classes that make programmers' life easier. Although we try to whitelist as many classes and methods as possible, we still try to keep our list minimalistic to avoid false positives.

Our static checker parses the source code of the Java program to build an Abstract Syntax Tree (AST)[1]. Then, it traverses the AST to collect various information about the source code: class instance creations, method and constructor invocations, field variable accesses, variable declaration statements, return statements, type casting operations, and import statements. Using this information, the checker generates a list of all the Java classes and methods that are used in the code. By comparing this list to our curated whitelist, the checker decides whether the code is safe to run or not.

Some classes are inherently non-deterministic due to the computations that they perform. `java.util.Random`, `java.util.Timer`, and `java.lang.Thread` are a few examples of such classes, as they involve random number generation, time-sensitive operations, and multi-threaded execution, respectively. Therefore, we can easily exclude these classes from our whitelist. Some classes are not fundamentally non-deterministic but they contain operations that exhibit non-deterministic behavior. `java.util.HashSet` and `java.util.HashMap` are two well-known examples of these classes. Most of the fundamental operations of these classes (*e.g.*, adding/removing elements, checking for element membership) are deterministic. However, since neither of them makes any guarantees about the iteration order of their elements, we exclude these classes from our whitelist. Luckily, `java.util` provides alternative hash set and hash map implementations, namely `LinkedHashSet` and `LinkedHashMap`, that are backed by linked lists, and therefore have deterministic iteration orders. We support both of these classes in our sandbox.

The `hashCode` and `toString` methods of the `Object` class are fundamental constructs of Java that present an interesting non-determinism challenge. Since every class in Java is a descendant of the `Object` class, these methods are automatically inherited by all Java programs. However, the `Object` class implementations of these methods are not guaranteed to return the same value across different executions since `hashCode` uses the memory address of the object in its calculations and `toString` uses `hashCode`. Therefore, if a program wants to use these functions in our sandbox, it should override them. Our static checker uses the AST of the program to verify that these functions are overridden and reject a program if they are not.

Exception and error handling is another point of consideration in building our sandbox. We allow programs to define variables that belong to the `java.lang.Exception` and `java.lang.Error` classes for more accurate error handling. However, we disallow all the methods that are in these classes since the information about exceptions and errors (*e.g.*, the stack trace) can be different across executions.

[1] We use an existing third-party Java library [66] for parsing Java source code and building the AST.

Finally, we discuss how the static checker handles the classes that contain methods for performing arithmetic operations, namely `java.lang.Math` and `java.lang.StrictMath`. As we discuss in detail in the next section, we cannot rely on these classes to support deterministic floating-point arithmetic in our sandbox. However, we do not blacklist these classes for two reasons. First, these classes contain methods that operate on non-floating-point types as well, which does not violate our determinism guarantees. Second, since we can use bytecode rewriting techniques to efficiently replace calls to the methods of these classes, we can make this process transparent to the programmers.

3.4.2 Deterministic floating-point arithmetic

In this section, we discuss how DEJAFLOAT eliminates non-determinism due to NaNs, transcendental functions, and extended precision to deterministically support floating-point arithmetic.

NaNs

As we have discussed earlier, IEEE 754 allows many distinct bit patterns for representing a NaN and does not specify exactly how NaN payloads should propagate. Therefore, the same floating-point operation can produce NaNs with different bit patterns on different hardware implementations of IEEE 754, thereby causing non-determinism. One solution to eliminate NaN-based non-determinism is to *canonicalize* NaNs by replacing their bit patterns with a fixed well-known bit pattern. Consequently, all NaNs can be represented by a single bit pattern to avoid the NaN-specific behavior of the underlying hardware architectures.

Java does not distinguish between NaN values, as it treats all float and double NaNs as if they are represented by a single canonical value [97]. In fact, Java internally collapses NaNs into a canonical value in certain floating-point operations (*e.g.*, comparing two float/double values). That being said, neither the Java nor the JVM specification mandates that all floating-point operations have to collapse NaNs to a canonical value. Furthermore, Java standard library provides methods, namely, `Float.intBitsToFloat` [124] and `Double.longBitsToDouble` [123], that allow programmers to create NaNs using different valid bit patterns. However, Java does not guarantee that these functions are going to return a NaN that encodes the exact bit pattern as the provided bit pattern, because the underlying hardware implementation can change the NaN bit patterns during execution (*e.g.*, converting an sNaN to a qNaN).

We have explored three approaches for canonicalizing NaNs:

- ***Canonicalize after every operation:*** The most naive approach is canonicalizing after every floating-point operation. However, performing a NaN check after every operation (and replacing its value when necessary) would add significant runtime overhead to program execution.
- ***Canonicalize on memory store:*** Instead of canonicalizing NaNs after every floating-

point operation, a more efficient approach is to canonicalize on every store operation that writes the value of a floating-point type to the memory. Unlike the previous approach, this does not require canonicalizing the results of intermediate computations that are stored in registers, and therefore has less overhead. We would need to implement this approach in WASM due to its flat memory model where a floating-point value that is stored to memory can later be reinterpreted as a different type when loaded from memory, which means its bit pattern can be observed. By canonicalizing NaNs before memory stores, we can ensure that the observed bit pattern is deterministic.

- ***Canonicalize on type conversion:*** JVM's strongly-typed memory provides an important opportunity for a smarter and more efficient approach for canonicalizing NaNs. Since a float/double type cannot be assigned to a variable or a memory location of a different type without an explicit conversion, we do not need to canonicalize when storing to a floating-point typed memory. Instead, we only need to canonicalize at a smaller set of operations that explicitly externalize the bit pattern of a floating-point value by changing its type. In our sandbox, we implement this approach to minimize the performance overhead due to NaN canonicalization.

There are two operations in Java that can externalize the bit pattern of a floating-point value by explicitly changing its type. First, Java standard library provides two sets of methods that return the bit representation of a given floating-point number: `Float.floatToRawIntBits` & `Double.doubleToRawLongBits` and `Float.floatToIntBits` & `Double.doubleToLongBits`. The first set of methods preserves the NaN bit patterns, and therefore can produce different results across different architectures. However, the second set of methods collapses all NaN values to a canonical NaN value, and is therefore deterministic. Our sandbox replaces the method calls in the first set with their deterministic alternatives by rewriting the bytecode of the user programs to change the method calls. Second, the narrowing type-casting operation that can be used to convert a float/double type to another primitive type, such as an int, long, or byte. This operation is also safe to use with NaNs since the JVM specification guarantees that the result of converting a float/double type with a NaN value to an int or long is always a 0.

The final operation that can externalize NaN bit patterns is a more subtle one, since, unlike the previous operations, it does not involve a type change. The `copySign` function in `java.Math` takes two floating-point arguments and returns the first argument with the sign of the second argument. Based on what we have discussed in the previous paragraph, this operations should be safe to use since the returned value is still a floating-point type. However, since the sign bit of a NaN is non-deterministic, the resulting floating-point can have different values depending on the underlying hardware architecture. To address this problem, our sandbox replaces calls to `copySign` with calls to a deterministic variant of it that is implemented in our sandbox. Our method checks if the second argument is a NaN and, if it is, replaces it with the same constant positive value.

Transcendental functions

The most recent IEEE 754 standard recommends, but does not require, that the results of the transcendental functions are correctly rounded. This is because producing correctly-rounded results for these functions could impose a performance penalty on the systems due to the Table Maker's Dilemma [128]: the problem of knowing how accurately a function needs to be approximated so that rounding the approximated result gives the same result as rounding the exact value of the function. Since it might require an arbitrarily large number of approximation steps to ensure that the obtained result is correctly rounded, most systems trade off correctness for performance by approximating these functions with a larger error than that required by correct rounding. For instance, given the term unit in the last place (ulp) [128], [154], which defines the distance between two consecutive floating-point numbers and is used commonly for measuring rounding error, a correctly rounded result is within 0.5 ulp of the exact result. However, most hardware and software implementations of IEEE 754 produce results that are between 0.5 and 1 ulp of the exact result to achieve better performance. Consequently, the result of the same transcendental function can vary across IEEE 754 conforming systems.

Existing approaches in Java: The Java standard library has two classes that contain methods for performing arithmetic operations: `Math` and `StrictMath`. The reason behind Java's design decision of providing two classes with the same arithmetic operations is to enable programmers to choose between performance and reproducibility.

Methods in the `Math` class can be implemented by platform-specific native libraries or hardware instructions to deliver high performance at the cost of reproducibility. For the transcendental functions, the `Math` class allows for implementations to return results with errors as large as 1 or 2 ulps. Since these functions are not required to be correctly rounded, or even rounded in the same way across different `Math` implementations, different implementations of the same `Math` function can return different results.

The `StrictMath` class guarantees reproducibility and portability of floating-point operations across different platforms by sacrificing performance. The floating-point operations in `StrictMath` are based on fdlibm [78], which is a C library that supports IEEE 754 floating-point arithmetic. fdlibm does not implement transcendental functions with correct rounding but it can still guarantee reproducible results since it is written in pure C, and is therefore platform-independent.

Even though `StrictMath` provides strict reproducibility of the floating-point operations, including transcendental functions, there are several reasons why our sandbox does not rely on it to support deterministic floating-point arithmetic. At the moment, Java uses version 5.3 of fdlibm as the de facto library that underlies `StrictMath`. However, there is no explicit requirement in the Java language or JVM specifications for how `StrictMath` should be implemented. Therefore, a later version of `StrictMath` can potentially round transcendental functions in a different way than that of fdlibm 5.3, thereby creating non-determinism across

different versions of otherwise the same Java/JVM implementation. Similarly, even a change in the underlying version of the fdlibm can potentially cause unreproducible floating-point arithmetic results across different Java versions. Moreover, the Java language or JVM specification cannot realistically prescribe that all future `StrictMath` implementations must compute their transcendental functions in exactly the same way as the particular version of a particular library does (*e.g.*, fdlibm 5.3), since it could eliminate the possibility of future improvements to how the transcendental functions are calculated.

Given that, even with `StrictMath`, we cannot rely on Java's native libraries for deterministically supporting floating-point arithmetic, we explore the following two solutions:

Solution #1: We use a library that supports correctly-rounded transcendental functions. More specifically, we use the MPFR library [85], [197], which is a C library that supports multiple-precision floating-point computations with correct rounding, in the prototype of our sandbox. Since the correctly-rounded result of a computation is independent of how it is computed, using a correctly-rounded library guarantees determinism. The limitation of this approach is that correct rounding can incur a high performance penalty. Therefore, using correctly-rounded floating-point operations can be infeasible for computationally-intensive applications.

Solution #2: An alternative approach is using a floating-point arithmetic library that does not implement correctly-rounded transcendental functions but is written in pure Java. The results of these functions are not going to be correctly rounded but since we run the library within our sandbox, it will at least behave identically and deterministically across all DEJAFLOAT sandbox implementations and underlying platforms. This approach provides better performance compared to the previous approach by sacrificing the accuracy of the computed functions.

In both approaches, our sandbox rewrites the Java class files of user programs to replace calls to transcendental functions of the `Math` and `StrictMath` classes with their deterministic counterparts that are provided by our sandbox. As a result, we eliminate non-determinism due to transcendental functions within the boundaries of our sandbox in a truly platform- and software-independent way. In §3.6, we compare MPFR to a pure Java floating-point arithmetic library without correct rounding support to quantify the performance benefits of using the latter.

Extended precision

Java's handling of the platform-specific extended precision format is one of the reasons why we consider it to be a good fit for building DEJAFLOAT. Originally, Java required strict evaluation of floating-point expressions, which means that the results of all floating-point computations (including the intermediate computations) are required to be values that can be represented

using the IEEE single- or double-precision format. As a result, Java guaranteed that floating-point computations would not produce different results due to the extended precision support of the underlying hardware. Between Java 1.2 and 16, Java allowed intermediate computations to use, by default, the extended precision formats, since enforcing strict semantics caused performance degradation. Therefore, programs had to explicitly use the `strictfp` modifier to achieve strict floating-point semantics. However, starting with Java 17, strict floating-point semantics is once again the default behavior in Java [51]. Therefore, we build our sandbox on top of Java 17. Note that if Java did not restore the strict floating-point semantics, our static checker could use the AST of the user program to make sure that the `strictfp` keyword was used.

3.4.3 Limiting program execution costs

To protect DEJAFLOAT from infinite and finite but computationally-costly program executions, the sandbox enforces a limit on the program execution based on the number of executed JVM bytecode instructions. The sandbox internally keeps a counter of the number of executed JVM instructions and if the counter goes above a pre-defined and fixed threshold value, it halts the execution of the program. To this end, we rewrite the Java class file of the program to insert our instrumentation code, which maintains the counter and reports it back to the sandbox at runtime.

A naive approach for counting the executed bytecode instructions is to increment the counter by one after every instruction. While this approach is simple and gives an accurate count, it requires executing four additional instructions due to our instrumentation for every instruction in the original class file. A more efficient approach is identifying the basic blocks in the program and incrementing the counter at the beginning of each basic block to execute fewer instrumentation instructions. A basic block is a sequence of instructions that has a single entry point at the beginning and a single exit point at the end. The instructions within a block are executed one after another, without branching out to another part of the program.

To implement the basic-block based approach, we first need to parse the class file to identify the basic blocks. Each block starts with a leader instruction so identifying the basic blocks is equivalent to identifying the leader instructions. A leader instruction has to satisfy one of the following: (1) it is the first instruction in the program, (2) it is the target of branch/jump instruction, (3) it is the instruction that immediately comes after a branch/jump instruction. Once we identify the basic blocks, we count the number of instructions in each basic block and increment the counter by that count once at the beginning of each basic block. As a result, this approach reduces the instrumentation overhead since it executes four additional instructions for each basic block, as opposed to each instruction.

3.5 Implementation

We implement a prototype of DEJAFLOAT on top of Java SE 17, which is the latest long-term support version. The static checker uses the Eclipse Java development tools [66](Eclipse JDT v3.30.0) to parse Java source files and create ASTs. For modifying the Java classes (*i.e.*, bytecode rewriting), we use the ASM library v9.4 [7]. Our prototype uses MPFR v4.1.0 with GMP v6.2.1. To be able to call MPFR functions from Java, we use an open-source library [153] that provides Java bindings to MPFR. We modify the library to make it compatible with MPFR v4.1.0 and to support a larger set of floating-point operations. For the pure Java floating-point arithmetic library, we use an open-source project called MicroFloat [43]. Although MicroFloat does not support correctly-rounded transcendental functions, because we run it within the deterministic sandbox, it will behave identically across all DEJAFLOAT sandbox implementations and underlying platforms to produce the same not-correctly-rounded results.

To better quantify how DEJAFLOAT benefits from the strong type system of Java in the context of NaN canonicalization, we implement and evaluate the performance of NaN canonicalization in WASM. To this end, we use `wasm-opt`, which is one of the tools in the Binaryen toolkit [212] for WASM. `wasm-opt` is normally used for applying wasm-to-wasm transformations for optimizing a WASM binary. However, we use its wasm-to-wasm transformation capability to insert NaN canonicalization routines before memory store operations. We use the WASI SDK v16.0 toolchain [213] to compile C files to WASM and Wasmer v3.3.0 [209] as our WASM runtime.

3.6 Evaluation

In this section, we experimentally evaluate the performance of DEJAFLOAT to answer the following questions:

1. What is the overhead due to NaN canonicalization?
2. What is the performance cost of supporting transcendental functions with correct rounding in computationally-intensive applications?
3. What is the runtime overhead of our instrumentation for counting the number of executed JVM instructions?

3.6.1 Experimental setup

We run our experiments on a Ubuntu 20.04 VM instance that is equipped with 16 vCPU (Intel Xeon Silver 4216 CPU @ 2.10 GHz) and 64 GB RAM. In all of our experiments, we use 64-bit double precision floating-point values. We collect our measurements after running many iterations of the measured code to warm up the JVM.

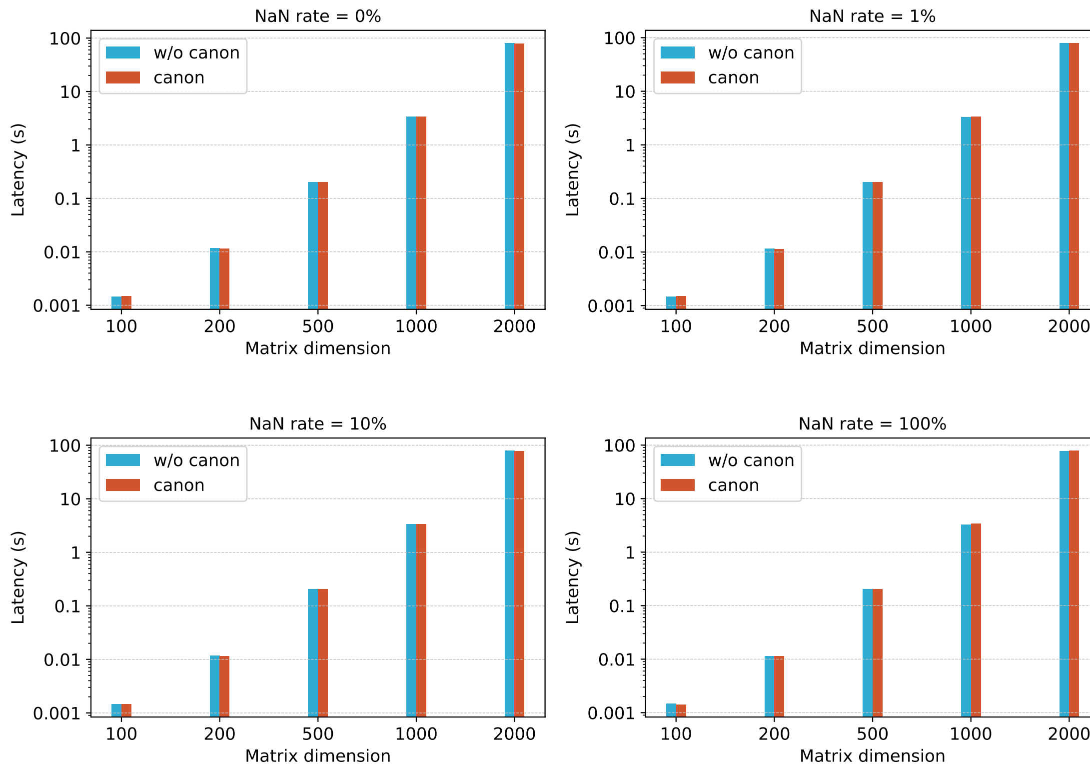


Figure 3.1: Execution times of the Java matrix multiplication program with and without NaN canonicalization

3.6.2 Microbenchmarks

NaN canonicalization microbenchmarks

In this experiment, we measure the overhead of NaN canonicalization. To demonstrate how DEJAFLOAT benefits from the strong typing of Java, we implement NaN canonicalization in WASM and measure its overhead for comparison. For evaluation, we use a program that first performs matrix multiplication and then converts the resulting matrix to an array of bytes so that we can force the externalization of floating-point bit patterns, and therefore the canonicalization routine. We choose matrix multiplication as our benchmarking program since it is a common computation in scientific computing and machine learning algorithms.

Figure 3.1 shows the execution times of the matrix multiplication program in our sandbox for varying matrix dimensions and percentage of NaN values in the matrices (*i.e.*, the NaN rate). Since we use square matrices, the x-axis values indicate both the row and column counts. Our results show that there is no discernible runtime overhead due to NaN canonicalization in Java. Thanks to Java's strong type system, our sandbox does not need to run the canonicalization routine during the matrix multiplication. Therefore, the difference between canonicalizing and not canonicalizing is the Java library function that is used to externalize the bit patterns of the double values, namely `doubleToLongBits` and `doubleToRawLongBits`. However, both

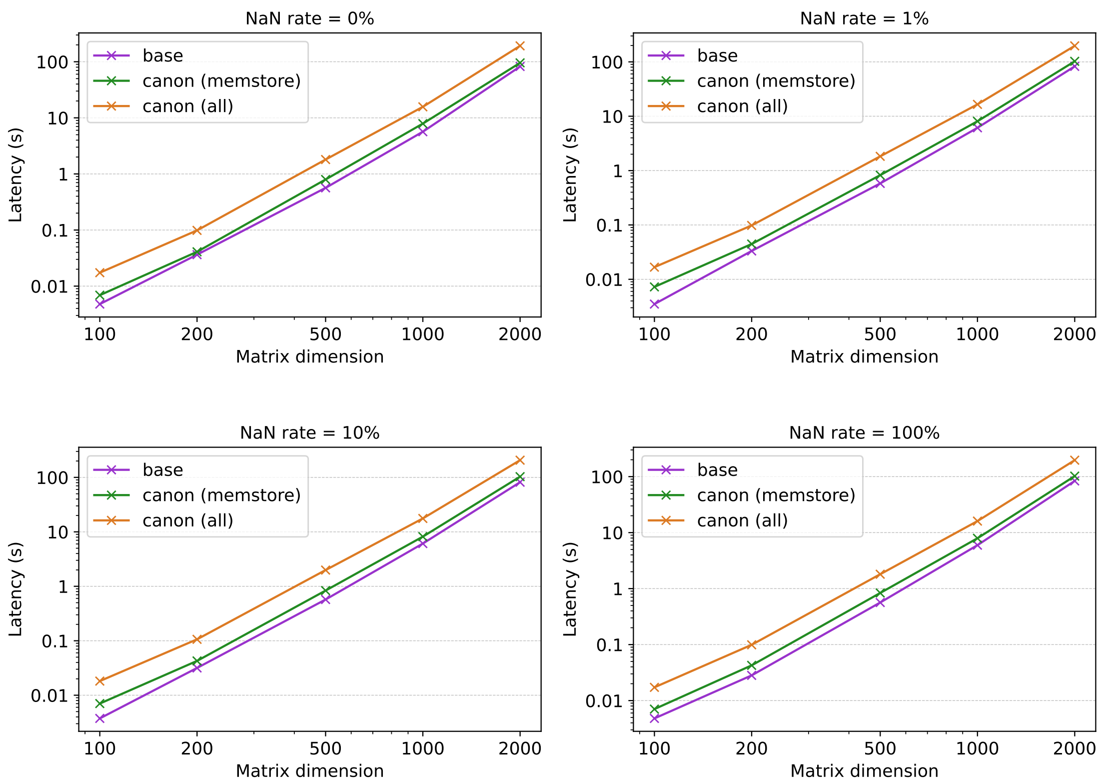


Figure 3.2: Execution times of the C matrix multiplication with and without NaN canonicalization in WASM

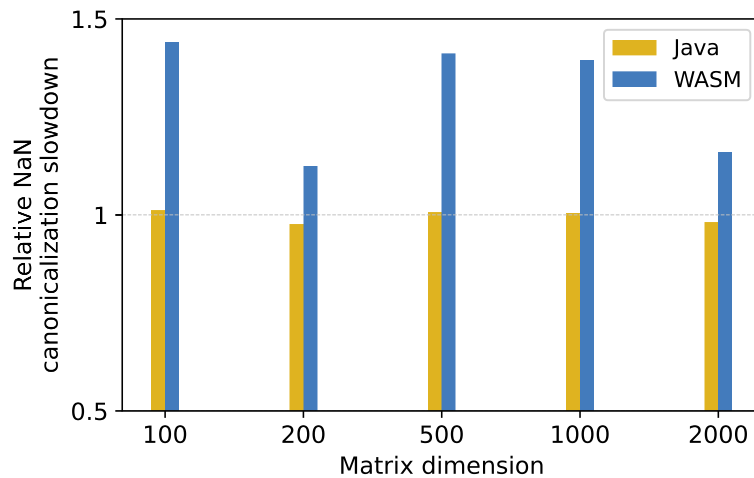


Figure 3.3: Relative slowdown in matrix multiplication (NaN rate = 0%) due to NaN canonicalization in Java and WASM.

methods are marked as intrinsic methods, which means that JVM can replace these methods with hand-crafted assembly code, with hand-crafted compiler IR, or with the combination of

	Transcendental function					
	ln	pow	exp	sin	cos	tan
Math	70	66	72	76	56	62
StrictMath	109	148	90	86	70	80
MPFR	2919	4947	2220	2525	2117	3580
MicroFloat	877	3055	746	484	488	993

Table 3.1: Execution times (ns) of various transcendental functions

both. Therefore, we expect these method calls to execute efficiently. Finally, when creating the input matrices, we put at least one NaN value in each row of the matrix if the NaN percentage value is not zero. As a result, each row-column multiplication produces a NaN, and therefore the output matrix consists of only NaNs. To sum up, our results show that implementing NaN canonicalization in our sandbox brings virtually no overhead to the execution of programs.

Figure 3.2 shows the execution times of the same program that is written in C and compiled to WASM. We report the results for three settings: *base*, which corresponds to no canonicalization, *canon (memstore)*, which corresponds to canonicalizing at memory store operations, and *canon (all)*, which corresponds to canonicalizing at every floating-point operation. Our results show that for large matrix dimensions (*i.e.*, 1000×1000 and 2000×2000), canonicalizing at each memory store operation results in a $1.2-1.4\times$ slowdown. The naive approach of canonicalizing after every floating-point operation adds a significant runtime overhead, as, for large matrix sizes, it incurs a $2.3-3\times$ slowdown. These results verify the performance gains of our sandbox due to its use of a strongly-typed language. For an easier side-by-side comparison of the overhead of NaN canonicalization in Java and WASM, Figure 3.3 represents the slowdown in matrix multiplication due to NaN canonicalization. We only present the slowdown for the scenario where neither matrix contains a NaN value since we expect this to be the common case. For WASM, we use the latency of canonicalizing at memory stores (memstore). This figure further clarifies how Java outperforms WASM in NaN canonicalization.

Math library microbenchmarks

Before we evaluate the performance of the libraries using real applications, we first run a microbenchmark to measure the execution times of common transcendental functions. We report our findings in Table 3.1. We note that `pow` computes x^y where both x and y are double values between 0 and 1, and `exp` computes e^x where x is a double value between 0 and 1. Our results show the price we pay for computing transcendental functions with correct rounding: MPFR is $40-75\times$ slower than Java's `Math` library. We observe that trading off correct rounding for performance by switching to a pure Java library can reduce the performance overhead: MicroFloat is $10-15\times$ slower than `Math` for all operations except for `pow`. We also note that the difference between the two math libraries of Java is marginal.

	Total time (s)
Math	20.4
StrictMath	20.7
MPFR	302
MicroFloat	22.5

Table 3.2: Total time to run user-based collaborative filtering with 5-fold cross-validation using the MovieLens 100K dataset

	Sigmoid		Tanh		ReLU & Softmax	
	bs = 1	bs = 32	bs = 1	bs = 32	bs = 1	bs = 32
Math	25.2	24.9	25.8	24.9	24.8	23.7
StrictMath	26	28.9	26.6	24.8	24.8	23.4
MPFR	47.1	46.2	72	67.4	31.9	34.4
MicroFloat	29.9	28	39.6	36.1	26	24.5

(a) Time to train a model using a single epoch

	Sigmoid	Tanh	ReLU & Softmax
Math	0.79	0.84	0.78
StrictMath	0.79	0.84	0.78
MPFR	3.34	3.69	1.10
MicroFloat	1.39	1.8	0.84

(b) Time to run the test set

Table 3.3: Neural network model training and testing with MNIST

3.6.3 Machine learning algorithms with deterministic transcendental functions

In these experiments, we evaluate the runtime overhead due to supporting deterministic transcendental functions in our sandbox. To this end, we implement two machine learning algorithms in Java: a user-based collaborative filtering algorithm that is used in recommendation systems and a multilayer perceptron, which is a fully-connected multi-layer neural network, that is used in image classification. We compare the performance of MPFR with Java's `Math` and `StrictMath` libraries, and with the MicroFloat library.

User-based collaborative filtering

In this experiment, we implement the user-based collaborative filtering algorithm using the Pearson correlation coefficient function. This algorithm is used in recommendation systems to provide users with recommendations based on their past activities and choices. More specifically, the algorithm finds users who have shown similar interests to a target user in the

	Sigmoid		Tanh		ReLU & Softmax	
	bs = 1	bs = 32	bs = 1	bs = 32	bs = 1	bs = 32
Math	28.9	28.2	26.6	29.1	30.9	28
StrictMath	28.8	28.4	29.4	29.1	33.1	28.1
MPFR	58.1	57.7	91.7	92.6	32.2	32.6
MicroFloat	32.8	34.1	38.3	45	29.8	29.2

(a) Time to train a model using a single epoch

	Sigmoid	Tanh	ReLU & Softmax
Math	1.34	1.33	1.33
StrictMath	1.36	1.37	1.32
MPFR	5.72	6.37	1.52
MicroFloat	2.27	1.86	1.34

(b) Time to run the test set

Table 3.4: Neural network model training and testing with CIFAR-10

past and use the ratings that they had given to an item to predict whether the target user would also like that item or not. The enforced deterministic execution capability of our sandbox can improve the reproducibility of recommendation systems, which in turn can improve the trust of users in the systems, user retention and engagement [72], [138]; and can help with the advancement of recommender systems research [73].

To evaluate our implementation, we use the popular MovieLens 100K dataset that contains 100,000 ratings from 1000 users on 1700 movies. We use the script in the dataset to create five training and test sets, which are created by performing a 80%/20% split on the data, to perform 5-fold cross validation. Table 3.2 shows the total execution times for running user-based collaborative filtering with 5-fold cross-validation. All the libraries except for MPFR take around 20 seconds while it takes MPFR around 300 seconds. The dramatic difference between the results is due to our naive use of the pow function for calculating squares in the Pearson correlation function. Java libraries have special checks to immediately the compute and return the square of a given value, which is not the case with MPFR. Considering that each fold involves around 10 – 11 million square functions, these results are expected. Note that we have confirmed that when we replace the pow functions with multiplications, all libraries perform the same.

Image classification with multilayer perceptrons

Image classification is one of the most popular machine learning applications and represents one of the use cases that can benefit from the capabilities of our sandbox. Being able to deterministically run an image classification task within a smart contract context can improve

the transparency and accountability of the process since any party can verify the trained model. Additionally, combining smart contracts with a popular machine learning application such as image classification, can create decentralized AI marketplaces that can collect large amounts of data from various sources and achieve better models.

In this experiment, we implement a neural network for classifying images. We use two popular datasets for our evaluation: the MNIST database, which is a database of handwritten digits with 60,000 training images and 10,000 test images, and the CIFAR-10 dataset, which consists of 60,000 32×32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 testing images. We use stochastic gradient descent (SGD) as the underlying optimization function. We also implement SGD with mini batching. The difference between these is that the former calculates the gradient to update the weights of the network after every training example, whereas the latter calculates the mean gradient over a batch of training examples and updates the weights once for each batch. Our neural network consists of an input layer, a hidden layer, and an output layer. The number of nodes in each layer is 784, 64, and 10 for the MNIST runs, and 512, 128, and 10 for the CIFAR-10 runs. We use three different activation functions in our neural network: sigmoid, tanh, and ReLU with softmax. In the last case, we use ReLU in the hidden layers and softmax in the output layer.

Tables 3.3 and 3.4 show the training and testing times for the MNIST and CIFAR-10 datasets. We report the latency of running a single epoch when training our model. A pass over the whole dataset is called an epoch. We train a model using both SGD (bs = 1) and SGD with mini batching (bs = 32) where we set the batch size to 32. Our results show that MPFR is roughly $2\times$ slower than the Java libraries when the activation function is sigmoid, and $3-3.5\times$ slower when the activation function is tanh. For ReLU and softmax, the performance of the libraries are similar since it contains fewer transcendental functions due to the use of ReLU in the hidden layer. Our results also show that MicroFloat performs much better than MPFR as it is at most $1.55\times$ slower than the Java libraries.

As we have demonstrated with the two machine learning applications, transcendental functions are one of the main sources of arithmetic operations for some applications. Our results show that using a correctly-rounded library like MPFR can be a significant source of slowdown for these applications. The alternative approach of using a pure Java library can be a better fit for these applications when performance is more important than the accuracy of the transcendental functions.

JVM bytecode instruction counting

In our final experiment, we evaluate the runtime overhead of counting the number of executed JVM bytecode instructions in our sandbox. We implement and evaluate both the per instruction and basic-block based instruction counting schemes.

We first measure the execution time of a simple Java program that recursively adds the numbers

No instrumentation	Instruction	Basic block
39 ms	64 ms	49 ms

Table 3.5: Execution times of a simple Java program with instrumentation for counting the number of executed JVM bytecode instructions

between 0 and n. We set n to 1000 and measure 10,000 runs. Table 3.5 shows the execution times with no instrumentation, per instruction counting, and basic-block based counting. Our results show the performance gain due to the basic-block based approach. We also run a Java program that performs a simple zero-knowledge proof verification of correct encryption as explained in [133]. We measure the time it takes to perform 10,000 proof verifications. Since the program consists mostly of function calls to a third-party library, and since we do not count the number of bytecodes executed in external libraries, the three schemes perform virtually the same, as they all take around 7.65 – 7.66 seconds to finish. To summarize, our experiment results show that instruction counting can be done practically in our sandbox when it is done at the basic block level.

3.7 Related work

Spoto [182] identifies some of the non-deterministic library and operations in Java and proposes a technique that enforces runtime determinism constraints of the Java library. Their work relies on manual analysis to identify deterministic parts of the Java library and uses annotations to specify run-time conditions. However, this work does not discuss non-determinism due to floating-point arithmetic. Like Spoto's work, we use a manual whitelisting approach to make sure DEJAFLOAT does not allow programs to use non-deterministic methods and classes from the Java standard library. However, since this approach requires manual effort, and is therefore time consuming and prone to errors, using static analysis techniques can considerably increase the efficiency and accuracy of the whitelisting process. For example, proving that Java programs written in a functionally pure subset of Java [82] or checking the purity of programs [165] are techniques that fit perfectly with what we are trying to achieve in DEJAFLOAT.

One of the main things that DEJAFLOAT does not support is multithreaded programming due to its non-deterministic nature. We intentionally leave out multithreaded programming support in DEJAFLOAT to simplify its design. That being said, prior work has shown how to enforce deterministic execution of multithreaded programs that are written both in Java [23], [40], [136] and other general-purpose programming languages such as C/C++ [17], [20], [141].

Deterministic instruction counting has been employed in different contexts: CoreDet [17], DMP [56], Determinator [11], and dOS [19] use it for supporting deterministic parallelism; Deterland [218] and TDR [36] use it for protection against timing attacks; and Binder and Hulaas use it as a reliable and reproducible way of profiling applications that run in Java [21],

[22].

Finally, Corda's DJVM [60] is a system that shares with DEJAFLOAT as it is a deterministic JVM sandbox that ensures deterministic execution of Corda smart contracts. Corda only allows explicitly-whitelisted runtime APIs to be called from the programs. Corda does acknowledge non-determinism due to floating-point arithmetic as they rewrite class files to apply the `strictfp` keyword to all methods. However, DJVM does not discuss how (or if) they handle non-determinism due to NaNs and transcendental functions.

3.8 Conclusion

We have presented DEJAFLOAT, an efficient JVM-based deterministic sandbox that can provide smart contracts with deterministic float-point arithmetic support. To enforce deterministic execution of smart contracts, we manually create a deterministically-safe subset of the Java standard library classes and methods, and restrict the programs to use only these whitelisted classes and methods within our sandbox. We eliminate the main sources of non-determinism in floating-point arithmetic, namely NaNs, transcendental functions, and extended precision, with minimum performance overhead by combining techniques that operate at the Java language and JVM level. We evaluated the performance of our DEJAFLOAT prototype using a combination of microbenchmarks and prototyped computationally-intensive machine learning applications. Our evaluation has demonstrated that our sandbox can efficiently support deterministic floating-point arithmetic.

4 Decentralized Private-Data Sharing with On-chain Secrets

4.1 Data confidentiality in blockchains

Blockchain is an emerging technology that enables mutually distrustful parties to securely exchange data and assets without relying on a single trusted party. Blockchains are expected to drive the digital transformation in various application domains, such as healthcare [92], [100], finance [193], [204], [214], identity management [104], [181], and supply chain [115], due to their strong integrity, transparency, and availability guarantees by design. However, despite these desirable properties, blockchains have an important shortcoming: they cannot guarantee confidentiality of data as all blockchain data is public and replicated across the nodes of the blockchain network. Lack of data confidentiality is one of the major roadblocks that prevent blockchains from delivering on their transformative promises.

Confidentiality-preserving blockchains can enable auditable and transparent management of private data in a secure manner without relying on a centralized entity. Today, centralized big-tech companies store and control users' data on the internet. However, the current state of affairs is problematic as these companies have repeatedly compromised the privacy of users due to security breaches and sharing users' data for profit without their consent [32], [58], [63], [93], [152], [208]. The recent data protection laws, such as General Data Protection Regulation (GDPR) [76] and the California Consumer Privacy Act (CCPA) [110], further emphasize the increasing need for data sovereignty and protecting the security and privacy of personal data. Decentralized data-sharing can enable users to have complete control over who can access their personal data and how it can be used, thereby balancing the tension between extracting value from personal data and the security and privacy concerns of the users [111], [119], [127], [188]. Existing solutions fail to achieve full decentralization as they rely on a trusted entity for the availability and security of the shared data [12], [64], [226].

Improving the confidentiality of blockchains can also address the pervasive issue of frontrunning [50], [74], [199] in public blockchains. In a frontrunning attack, an adversary gains unfair financial advantage over other entities by obtaining early access to sensitive information. In the case of decentralized applications that are built on top of blockchains, some examples

of sensitive information can be bid amounts (auctions) or the contents of buy/sell orders (trading). What makes frontrunning attacks possible is that the contents of blockchain transactions are transparent and in plaintext: an attacker can monitor the contents of transactions in the network before they are finalized and create their transactions accordingly to maximize their advantage. Privacy-preserving blockchains can mitigate frontrunning attacks by keeping sensitive on-chain data secret so that an attacker cannot gain advantage by observing the blockchain. Existing solutions for mitigating frontrunning attacks impose latency and traffic overhead as they are based on multi-round commit-reveal schemes [28], [149], rely on a trusted party [16], [183], or are domain-specific [42].

We introduce on-chain secrets, a secure decentralized data-management framework that enables auditable and fair access to confidential data without relying on a single trusted party. Building a framework with the above properties requires on-chain secrets to address two challenges. The first challenge is to ensure that confidential data is only revealed to authorized readers that are specified by the data owner. Data accesses have to be recorded in an immutable log to verify that access control is enforced correctly and to enforce auditability and accountability of data accesses. The second challenge is to ensure that confidential data becomes available to all users at the same time. On-chain secrets addresses these challenges by employing decentralized committees to combine threshold cryptography and the blockchain technology. Threshold cryptography enables a committee of nodes to collectively and securely store confidential data without having a single point of failure or compromise. The blockchain manages accesses to confidential data by enforcing access control policies and maintains an immutable log of data accesses for auditability. Additionally, the blockchain ensures fair access to data by simultaneously releasing it to all authorized parties at the same time based on the rules defined in the access control policy.

We propose two protocols for building on-chain secrets, namely one-time secrets (OTS) and post-quantum one-time secrets (PQ-OTS), that employ different threshold cryptography primitives and have different security guarantees. OTS relies on a publicly-verifiable secret sharing (PVSS) scheme to encrypt data for an ad hoc set of trustees. PQ-OTS replaces PVSS with Shamir's secret sharing (SSS) to support long-term confidentiality of encrypted data against a quantum adversary. We implemented a prototype of on-chain secrets using both protocols to evaluate its performance. Our experiments show that on-chain secrets can scale up to 128 trustees with both protocols to provide a high level of decentralization. Furthermore, we use two real-world applications to evaluate the performance of on-chain secrets under realistic workloads. First, we compare on-chain secrets to a state-of-the-art semi-centralized system using a confidential-document sharing application. On-chain secrets incurs a 18 – 26% slowdown in execution time compared to the semi-centralized system, which is an acceptable overhead for achieving full decentralization. Second, we compare on-chain secrets to a state-of-the-art zero-collateral decentralized lottery. The on-chain secrets lotteries are 15 – 25% faster than the state-of-the-art solution as they require fewer rounds to run the lottery.

On-chain secrets was originally introduced as one of the two components of the CALYPSO

architecture [133]. The other component is a decentralized access-control system called SIAM that enables on-chain secrets to be used with dynamic access-control policies and identities. More specifically, SIAM enables data owners to change the access control policies (*e.g.*, grant or revoke access) even after they share the data using on-chain secrets. Furthermore, it enables users to update their identities (*e.g.*, add or delete public keys) without losing access to data that is already shared with them. In this thesis, we present on-chain secrets a standalone framework without SIAM. Therefore, we assume that the access control policies are static (*e.g.*, list of public keys) and cannot be changed after data is shared with on-chain secrets.

To summarize, our contributions in this chapter are the following:

- The design of on-chain secrets, a secure decentralized data-management framework that enables auditable and fair access to confidential data without relying on a single trusted entity.
- The design and implementation of two on-chain secrets protocols: one-time secrets (OTS) and post-quantum one-time secrets (PQ-OTS). OTS is a PVSS-based protocol that enables users to share their secret data with an ad hoc set of trustees without requiring a setup phase. PQ-OTS enhances OTS to guarantee long-term confidentiality of secret data against quantum adversaries.
- An experimental evaluation of OTS and PQ-OTS using two real-world applications, namely a confidential-document sharing application and a zero-collateral decentralized lottery, under both synthetic and realistic workloads. For both applications, we compare on-chain secrets against the state-of-the-art solutions.

4.2 Motivating applications

Auditable data-sharing: Sharing private user-data among different entities is a desirable capability in various application domains, such as scientific research, business processes, and governance. Sharing medical records among hospitals and researchers is critical for medical research to improve the diagnosis and treatment of diseases [103], [172]. Companies collect and use customer data for making informed business decisions [118], [125], thereby improving the quality of their services and products. Finally, government and intelligence agencies occasionally require access to personal information held by third parties for effective law enforcement [96], [174], [203]. Centralized solutions to private-data sharing are convenient and easy to implement but require trusting a single entity for ensuring data confidentiality and auditability of data accesses.

On-chain secrets provides a secure platform for auditable private data-sharing without relying on a single trusted party. On-chain secrets enables data owners to control who can access their data, to monitor accesses to their data, and to verify that the access control rules are enforced correctly. By improving the security and transparency of the private-data sharing process, on-chain secrets can strengthen users' trust in the process and motivate them to share their sensitive medical data for research purposes [127]. A secure decentralized data-

sharing platform like on-chain secrets can be used for building data markets where users can monetize their data by sharing it with companies. Finally, on-chain secrets can improve the accountability of lawful data-access requests by maintaining a transparent and publicly-verifiable log of data accesses.

Frontrunning prevention: As public blockchains are permissionless and open to everyone, they are susceptible to attacks from malicious users. One of the most common types of attacks is frontrunning where an attacker exploits early access to sensitive information to gain unfair advantage over honest users. These attacks are possible because blockchain transactions are in plaintext and publicly visible. Decentralized applications such as exchanges, crypto collectibles, and gambling games have been affected by frontrunning attacks in the past [75].

On-chain secrets can address the frontrunning problem in these applications by hiding the user inputs until a barrier point (*e.g.*, block height, timestamp) that is specified in the access control policy. After the barrier point is reached, on-chain secrets atomically reveals the inputs to all authorized parties at the same time, thereby preventing a malicious user to obtain premature access to the sensitive data. In this way, on-chain secrets can bring fairness to decentralized applications.

4.3 System overview

4.3.1 Challenges

To better understand the challenges in building a decentralized system for auditable private-data sharing, let us consider an application where Wendy operates a paid service that provides investment advice for the stock market (*i.e.*, buy/sell stocks) and Ron wants to purchase this service. Since Wendy and Ron are mutually distrustful, they might act dishonestly to exploit the system. For example, Wendy might take the money without disclosing the private data (*i.e.*, investment advice) to Ron or Ron might lie about not having received the private data despite making the payment.

A strawman solution can be built using a trusted custodian. Wendy sends the private data and the list of public keys of users who can access this data to the custodian. The custodian generates a proof that it has stored the data and Wendy stores this proof on Bitcoin so that it is publicly and immutably logged. Later, Ron makes the payment to Wendy on Bitcoin and takes the proof of payment to the custodian. Once the custodian verifies the proof of payment and that Ron is authorized to access the data, it releases the data encrypted under Ron's public key and stores a proof on Bitcoin to publicly record that Ron has accessed the data.

The strawman solution provides auditability because both Wendy's and Ron's actions are publicly logged on Bitcoin. Furthermore, the custodian protects the confidentiality of data by enforcing the access-control rules defined by Wendy to prevent unauthorized accesses to

data. However, the strawman solution has a significant drawback as the trusted custodian is a single point of failure and compromise. If the custodian crashes, there is no guarantee that Ron can access the data, even if he has already made the payment. Moreover, if the custodian is malicious or compromised, it can secretly release the data to Ron without a record of payment and without recording the access on Bitcoin. Therefore, we need auditability and data confidentiality in a fully decentralized setting that does not have a single point of failure or compromise. Even if the custodian is honest, this solution can cause frontrunning attacks. Since the custodian releases the data on a first-come-first-serve basis, a user who can get their payment accepted in the Bitcoin network faster can obtain early access to the investment advice. As a result, this user can gain unfair financial advantage in the stock market in our example application. Therefore, we require fair access to data where all users get access at the same time.

4.3.2 System goals

On-chain secrets aims at achieving the following goals:

1. **Confidentiality:** Secret data that is stored on the blockchain is only revealed to authorized users. Unauthorized users cannot learn anything about the secret data.
2. **Auditability:** Data accesses are immutably recorded on the blockchain and are third-party verifiable.
3. **Fair access:** Users are guaranteed to get access to decrypted data if and only if they have recorded a valid access request on the blockchain. If there is a barrier point, all authorized users simultaneously get access to data after the barrier point is reached.
4. **Decentralization:** There is no single point of failure or compromise.

4.3.3 System model

Writers are data owners who share their secret data on the blockchain. They send the encrypted secret data and its access control policy in a *write transaction* to the blockchain. *Readers* are data consumers who access the data that is shared by writers. They send a *read transaction* to the blockchain to obtain access to data. The *access-control blockchain* is responsible for logging write and read transactions, and authorizing accesses to data by enforcing access control policies. The *secret-management committee* is responsible for managing and revealing the secret data. The nodes of the secret-management committee are called *trustees*. We use Wendy and Ron to refer to a generic writer and reader, respectively.

The access-control blockchain can be implemented in different ways as long as it satisfies the following requirements: it has to build on Byzantine-fault-tolerant (BFT) consensus and it has to support program execution on top of the consensus algorithm. For example, a set of trusted nodes that maintains a programmable blockchain using BFT consensus in a permissioned setting or an access-control smart contract that runs on top of a permissionless smart contract

platform (*e.g.*, Etheruem) are two alternative ways of implementing the access-control blockchain. The secret-management committee can be setup in different ways depending on the specific on-chain secrets protocol implementation. In OTS and PQ-OTS, it is set up ad hoc on a per-secret-data basis. It is also possible to set up the committee using a pre-determined set of trustees that persists over a longer period of time [133]. The secret-management committee does not maintain a blockchain and does not require running consensus for data access requests. For architectural clarity, we assume that the access-control blockchain and the secret-management committee are deployed on different sets of nodes. However, in practice, they can also be deployed on the same set of nodes.

4.3.4 Threat model

In OTS (§4.4), we make the standard cryptographic assumptions that secure symmetric-key and public-key algorithms, cryptographic hash functions, and signature schemes exist. We assume that the adversary is computationally bounded and that there is a cyclic group where the decisional Diffie-Hellman assumption holds. In PQ-OTS (§4.5), we assume a stronger adversary that is computationally unbounded and only rely on the existence of secure cryptographic hash functions and symmetric-key algorithms.

We use n_{ACB} and n_{SMC} to denote the number of nodes that run the access-control blockchain and the number of trustees in the secret-management committee, out of which f_{ACB} and f_{SMC} can behave maliciously, respectively. For the access-control blockchain, we assume that the security assumptions of the underlying consensus algorithm holds, *e.g.*, $n_{ACB} = 3f_{ACB} + 1$ for classic BFT consensus or $n_{ACB} = 2f_{ACB} + 1$ for Nakamoto-style consensus. For the secret-management committee, we set the secret-recovery threshold to $t = f_{SMC} + 1$ and require $n_{SMC} \geq 2f_{SMC} + 1$ and $n_{SMC} \geq 3f_{SMC} + 1$ for OTS and PQ-OTS, respectively. It is also possible to choose higher t values to achieve stronger security guarantees. However, this can have a negative impact on availability and performance. Finally, we assume that Wendy and Ron are mutually distrustful as we describe in §4.3.1.

4.3.5 Architecture overview

Figure 4.1 shows the architecture of on-chain secrets and how different entities interact with each other as part of a data-sharing workflow. We assume that Wendy symmetrically encrypts her data under a key k, which is the *secret* that is shared on-chain. Wendy sends the encrypted secret and an access control policy in a write transaction (tx_w) to the access-control blockchain. The access control policy specifies the rules for accessing the secret, such as the identities of authorized readers and a barrier point definition. In this thesis, we assume that both the identities of authorized readers and the access control rules are static (*e.g.*, policy = public key (pk) of Ron). However, on-chain secrets can be extended to support dynamic access-control and identity management [133].

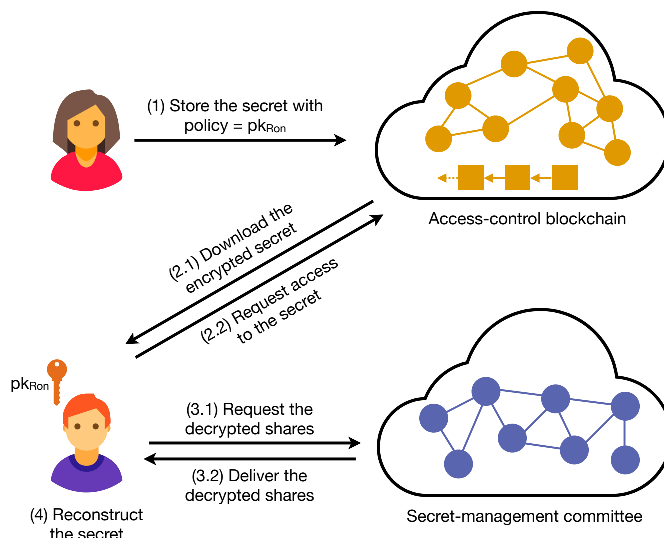


Figure 4.1: The high-level architecture of on-chain secrets

Wendy can store the ciphertext of the symmetrically-encrypted data on the blockchain or on a decentralized file-sharing network (*e.g.*, IPFS) for high availability. Alternatively, Wendy can also hold on to the ciphertext and serve it to Ron herself. In all cases, before requesting access to a secret, Ron should make sure to obtain the relevant ciphertext and to verify that there is already a tx_w for it on the blockchain. Once the access-control blockchain verifies and logs tx_w, Ron can fetch it from the blockchain and send a read transaction (tx_r) to the access-control blockchain. tx_r contains a pointer to tx_w and a cryptographic proof of Ron's identity. The access-control blockchain verifies that tx_w is already logged and Ron's identity is listed as an authorized reader in the access control policy, as specified in tx_w. If Ron is authorized to access the secret, the access-control blockchain logs tx_r.

The logged tx_r serves a publicly-verifiable proof that Ron has requested access to the secret. It also guarantees that Ron is going to receive the decrypted secret. Ron sends the logged tx_w and tx_r in a request (req_{share}) to the secret-management committee to recover the secret. Each secret-management trustee verifies that tx_w and tx_r are logged on the blockchain and the barrier point (if specified) is reached. Then, each trustee decrypts their share of the secret and sends it back ($resp_{share}$) to Ron. Finally, Ron polls the shares, recovers the symmetric key k, and decrypts the data that Wendy shared with him.

4.4 One-time secrets (OTS)

In this section, we introduce *one-time secrets* (OTS), an on-chain secrets implementation that is based on a publicly-verifiable secret sharing (PVSS). As with other secret-sharing schemes, PVSS enables a dealer to share a secret s among n trustees such that any subset of t trustees can combine their shares to reconstruct s, but any subset of $t-1$ trustees cannot learn anything

about s. The additional property of PVSS is that it enables any party (*i.e.*, not only the trustees) to verify the correctness of the distributed shares.

In OTS, Wendy runs PVSS with a distinct secret-management committee on a per-secret basis. Wendy only needs the public keys of the trustees to set up the secret-management committee. Trustees do not need to coordinate with or know about each other. After choosing a set of secret-management trustees, Wendy runs PVSS to create the encrypted shares of a secret for each trustee by using their public keys. She uses the shared secret as the symmetric key k to encrypt the data that she wants to share with Ron. Binding k to the PVSS shares guarantees that recovering k requires a threshold of trustees to reveal their decrypted shares. It also guarantees that those who have not participated in PVSS (*e.g.*, Ron and the access-control blockchain) can also verify that Wendy has distributed the shares correctly. Finally, Wendy sends the encrypted shares, the non-interactive zero-knowledge (NIZK) proofs of consistent encryption and the access control policy in tx_w to the access-control blockchain to share the secret with Ron. Upon logging a valid tx_r on the blockchain, Ron sends the encrypted shares and a proof that his tx_r is logged on-chain to each secret-management trustee to obtain the decrypted shares. Once Ron receives a threshold number of valid decrypted shares, he can reconstruct the symmetric key and open the secret message.

4.4.1 OTS subprotocols

Before diving into the details of the OTS subprotocols (Figure 4.2), we first introduce the notation that we are going to use in this section. Let $\mathbb{G}$ be a multiplicatively written cyclic group of prime order q with generator G where the decisional Diffie-Hellman assumption holds. Each trustee i has a private key x_i and a corresponding public key $y_i = G^{x_i}$. Similarly, we use (y_W, x_W) and (y_R, x_R) to denote Wendy and Ron's public-private key pairs, respectively. We use the angle brackets ($\langle\rangle$) to denote a list of elements. For example, we use $\langle y_i \rangle$ to show the list of the trustee public keys.

Write subprotocol

Wendy prepares a write transaction (tx_w) as follows:

1. Computes $h = \text{H}(\text{policy})$ to map the access control policy to a group element $h \in \mathbb{G}$ to be used as the base point when generating the PVSS polynomial commitments. This prevents replay attacks as we describe in §4.4.3.
2. Chooses a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The secret to be shared is $S = G^{s(0)}$.
3. For each trustee i, computes the encrypted share $\widehat{s_i} = y_i^{s(i)}$ of the shared secret S and creates the corresponding NIZK proof of consistent encryption $\pi_{\widehat{s_i}}$, which proves that the share is encrypted correctly. Creates the polynomial commitments $c_j = h^{a_j}$, for $0 \le j \le t-1$.

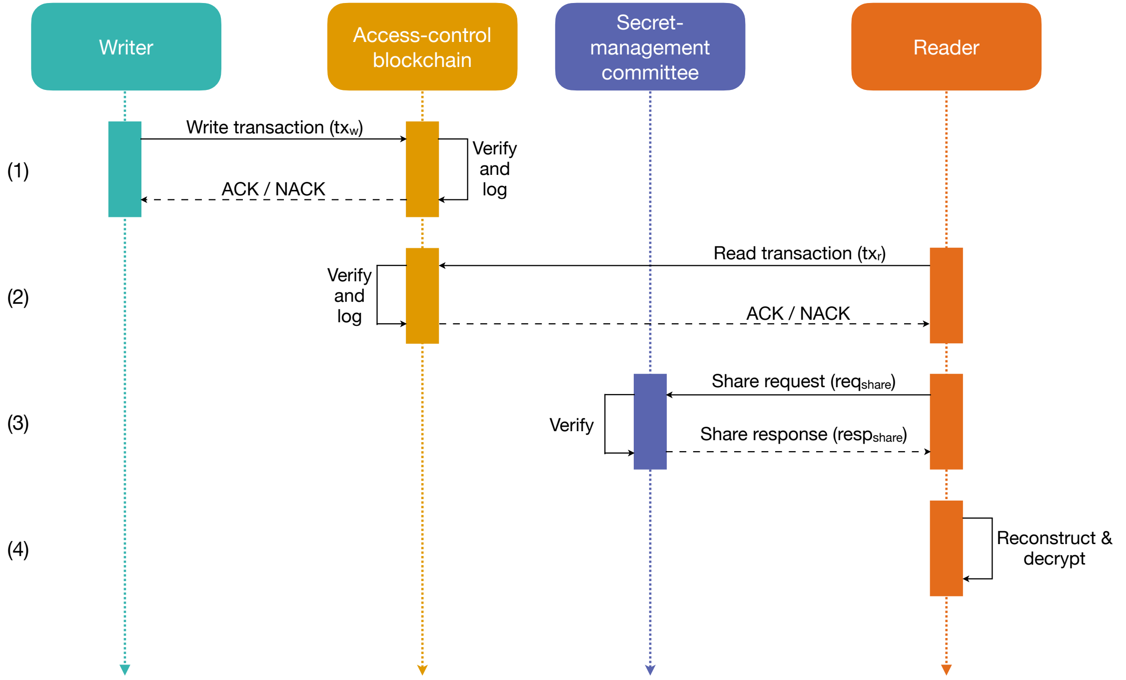


Figure 4.2: One-time secrets subprotocols: (1) Write, (2) Read, (3) Share retrieval, (4) Secret reconstruction

4. Computes $\mathrm{H}(S)$ and uses it as the symmetric key k to encrypt the secret message m as $c = \mathrm{enc}_k(m)$. Computes the hash of the ciphertext $H_c = \mathrm{H}(c)$, which can be used to check the integrity of the ciphertext. Sets policy $= y_R$ to designate Ron as the intended reader of the secret message m.
5. Finally, prepares and signs $\mathrm{tx_w} = [\langle \hat{s}_i \rangle, \langle c_j \rangle, \langle \pi_{\hat{s}_i} \rangle, H_c, \langle y_i \rangle, \mathrm{policy}]_{\mathrm{sig}_W}$ and sends it to the access-control blockchain.

$\pi_{\hat{s}_i}$ proves that the corresponding encrypted share $\hat{s}_i$ is consistent. More specifically, it is a proof of knowledge of the unique $s(i)$ that satisfies:

$$A_i = h^{s(i)}, \hat{s}_i = y_i^{s(i)}$$

where $A_i = \prod_{j=0}^{t-1} c_j{}^{i^j}$. In order to generate $\pi_{\hat{s}_i}$, Wendy picks at random $w_i \in \mathbb{Z}_q$ and computes:

$$a_{1i} = h^{w_i}, a_{2i} = y_i^{w_i},$$
$$C_i = \mathrm{H}(A_i, \hat{s}_i, a_{1i}, a_{2i}), r_i = w_i - s(i)C_i$$

where C_i is the challenge and r_i is the response. Each proof $\pi_{\hat{s}_i}$ consists of C_i and r_i, and it proves that $\log_h A_i = \log_{y_i} \hat{s}_i$.

Before logging $\mathrm{tx_w}$, the access-control blockchain ensures that it contains valid encrypted shares as follows:

1. Derives the PVSS base point $h = \mathrm{H}(\text{policy})$.
2. Computes $A_i = \prod_{j=0}^{t-1} c_j{}^{i^j}$ using the polynomial commitments c_j, $0 \le j < t$.
3. Computes $a'_{1i} = h^{r_i} A_i^{C_i}$ and $a'_{2i} = y_i^{r_i} \widehat{s_i}^{C_i}$
4. Checks that $\mathrm{H}(A_i, \widehat{s_i}, a'_{1i}, a'_{2i})$ matches the challenge C_i.
5. If all shares are valid, logs $\mathrm{tx_w}$ on the blockchain.

Read subprotocol

After $\mathrm{tx_w}$ has been logged, Ron needs to log a read transaction ($\mathrm{tx_r}$) on the access-control blockchain before he can request the secret. To do so, Ron performs the following steps:

1. Retrieves the ciphertext c and block b_w, which stores $\mathrm{tx_w}$, from the access-control blockchain.
2. Checks that $\mathrm{H}(c)$ matches H_c in $\mathrm{tx_w}$ to verify that ciphertext c has not been altered.
3. Computes $H_\mathrm{w} = \mathrm{H}(\mathrm{tx_w})$ to uniquely identify the $\mathrm{tx_w}$ that he wants to read.
4. Prepares and signs $\mathrm{tx_r} = [H_\mathrm{w}]_{\mathrm{sig}_R}$, and sends it to the access-control blockchain.

To log the read transaction on the blockchain, the access-control blockchain performs the following steps:

1. Retrieves $\mathrm{tx_w}$ using H_w. Verifies the signature sig_R on $\mathrm{tx_r}$ against the public key (*i.e.*, y_R) recorded in policy.
2. If the signature is valid, it means that Ron is authorized to access the secret. In that case, logs $\mathrm{tx_r}$ on the blockchain.

Share retrieval subprotocol

After $\mathrm{tx_r}$ has been logged, Ron can run the share retrieval subprotocol with the secret-management committee to obtain the shares of the encryption key k. To do so, Ron initiates the protocol as follows:

1. Creates and signs a share-retrieval request $\mathrm{req_{share}} = [\mathrm{tx_w}, \mathrm{tx_r}, \pi_{\mathrm{tx_r}}]_{\mathrm{sig}_R}$, where $\pi_{\mathrm{tx_r}}$ proves that $\mathrm{tx_r}$ has been logged on the blockchain.
2. Sends $\mathrm{req_{share}}$ to each secret-management trustee to retrieve the decrypted shares.

Upon receiving $\mathrm{req_{share}}$, each secret-management trustee executes the following steps:

1. Uses y_R in $\mathrm{tx_w}$ to verify the signature on $\mathrm{req_{share}}$.
2. Using $\pi_{\mathrm{tx_r}}$ verifies that $\mathrm{tx_r}$ has been logged on the blockchain.
3. Computes the decrypted share $s_i = (\widehat{s_i})^{x_i^{-1}}$ and creates the corresponding NIZK proof of consistent decryption π_{s_i}, which proves that the share is decrypted correctly. The proof shows the knowledge of the unique value that satisfies $\log_G x_i = \log_{s_i} \widehat{s_i}$.
4. Encrypts the decrypted share s_i using y_R to obtain $r_i = \mathrm{enc}_{y_R}(s_i)$. This guarantees that only Ron can retrieve the decrypted share.

5. Creates and signs the share-retrieval response $\text{resp}_{\text{share}} = [r_i, \pi_{s_i}]_{\text{sig}_i}$, and sends it back to Ron or publishes it publicly.

Secret reconstruction subprotocol

To decrypt the secret message m, Ron needs to recover the secret key k by reconstructing S as follows:

1. Decrypts each $s_i = \text{dec}_{y_R}(r_i)$ and verifies it against π_{s_i}.
2. If there are at least t valid shares, uses Lagrange interpolation to reconstruct S.
3. Recovers k by computing $\text{H}(S)$ and uses it to decrypt c to obtain m.

4.4.2 Advantages and shortcomings

OTS has several advantages due to the simplicity of its design. First, it does not require trustees to run a setup phase to create a secret-management committee. In fact, trustees do not even need to coordinate with or be aware of each other. Wendy can create a secret-management committee simply by running PVSS using the public keys of the trustees that she wants to have in the committee. Second, there is no setup phase for the secret-management committee, Wendy can easily set up distinct secret-management committees with different committee sizes and security thresholds on a per-secret basis. Finally, secret-management trustees do not need to store any protocol state except for their private-public key-pairs.

OTS also has certain shortcomings. First, dealing the PVSS shares has a high computational cost: Wendy needs to evaluate the secret sharing polynomial at n points, create n encrypted shares and NIZK proofs, and t polynomial commitments. Therefore, as the number of trustees increase, the client-side overhead of OTS increases as well. Second, the size of tx_w increases linearly with the secret-management committee size. This is due to the trustees not storing any protocol state. As a result, tx_w has to contain the encrypted shares, the NIZK proofs and the polynomial commitments. Finally, since the encrypted shares are bound to the public keys of the trustees, the secret-management committee cannot be changed without creating new shares.

4.4.3 Revisiting the system goals

We explain how OTS achieves the system goals listed in §4.3.2.

Confidentiality: *Secret data that is stored on the blockchain is only revealed to authorized users. Unauthorized users cannot learn anything about the secret data.*

Wendy uses a symmetric key k to encrypt the data that she wants to share with Ron. She runs PVSS among the secret-management trustees and uses the generated shared secret as k. Since the PVSS shares are encrypted under the public keys of the trustees, they do not

leak any information about k. After logging a tx_r on the blockchain, Ron can send a valid req_{share} to the secret-management committee. Since there are at least t honest trustees in the secret-management committee, Ron can collect enough decrypted shares to recover k and decrypt the data. Since there can be at most f_{SMC} malicious trustees and the secret recovery threshold is set to $t = f_{SMC} + 1$, a malicious reader cannot obtain enough decrypt shares to recover the secret without logging a tx_r on the blockchain. Furthermore, a malicious reader cannot access another user's secret data through a replay attack where they copy the encrypted shares from a tx_w to a new tx_w' with a modified access control policy. This replay attack is prevented by binding the access control policy to the base point that is used for the PVSS encryption-consistency proofs.

Auditability: *Data accesses are immutably recorded on the blockchain and are third-party verifiable.*

Assuming that the access-control blockchain builds on top of BFT consensus, all valid tx_w and tx_r are logged on the blockchain. Anyone can use the public blockchain data and inclusion proofs to verify that the transactions are logged on the blockchain.

Fair access: *Users are guaranteed to get access to decrypted secret data if and only if they have recorded a valid access request on the blockchain. If there is a barrier point, all authorized users simultaneously get access to data after the barrier point is reached.*

Shares of the secret data are stored in encrypted format on the blockchain. Since the PVSS recovery threshold is $t = f_{SMC} + 1$, Ron cannot obtain enough decrypted shares from the malicious trustees to recover the secret without the proof that he has already logged a tx_r on the blockchain. Similarly, since there are at least t honest nodes in the secret-management committee, if Ron has the proof that he has logged a tx_r, these nodes are going to follow the protocol and reveal their shares to him.

Decentralization: *There is no single point of failure or compromise.*

OTS does not rely on a single trusted party. The access-control blockchain builds on top of BFT consensus to tolerate a threshold number of malicious nodes. The secret-management committee uses PVSS to tolerate up to $t - 1$ malicious nodes.

4.5 Post-quantum one-time secrets (PQ-OTS)

When describing the threat model for OTS in §4.3.4, we make the assumptions that *the adversary is computationally bounded* and *there is a cyclic group $\mathbb{G}$ where the decisional Diffie-Hellman assumption holds.* These assumptions are essential to the security of one-time secrets since the security of PVSS depends on them. In addition, these are realistic assumptions considering the computational capabilities of today's adversaries: it is computationally infeasible to solve the discrete logarithm problem using traditional computers. Therefore, we can rely on PVSS, and thus OTS, to securely store private data in on-chain secrets.

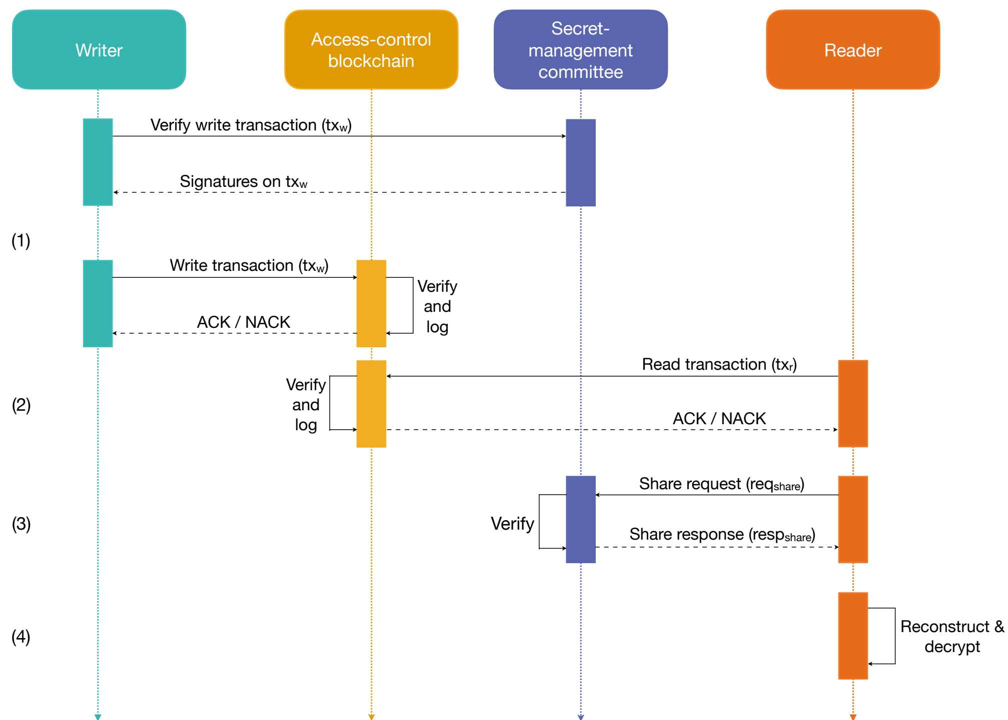


Figure 4.3: Post-quantum one-time secrets subprotocols: (1) Write, (2) Read, (3) Share retrieval, (4) Secret reconstruction

However, there is no guarantee that the cryptographic primitives that we build on today will be secure in the future as well. Given the recent advances in quantum computing, computationally-unbounded adversaries who have access to strong quantum computers can break our currently-secure cryptographic primitives in the future. For example, Shor's algorithm [178] shows that the discrete logarithm problem can be solved efficiently on a quantum computer, thereby breaking the security of many widely-used cryptographic schemes. In the case of PVSS, an attacker can recover the shared secret by using Shor's algorithm to compute the coefficients of the secret-sharing polynomial from the polynomial commitments, thereby breaking the security of PVSS.

A large international community has already acknowledged the security threats posed by quantum computing [92], [126]. Researchers have been supporting the development and prototyping of quantum-safe cryptography protocols [113], [184], [205]. Governments have also been supporting the transition to post-quantum cryptography [79], [116], [117] and identifying the requirements for quantum-resistant security systems [80], [114]. One example of how governments have already started considering the quantum risks when building electronic systems is the e-voting system in Switzerland, which is known to trade off public verifiability of the voting process for protecting the long-term confidentiality of the encrypted ballots.

The above examples show that we need to design our systems to be secure against the threat

of quantum computers. To this end, we present *post-quantum one-time secrets (PQ-OTS)*, a quantum-safe version of OTS that guarantees the *long-term confidentiality* of the private data. The key idea behind PQ-OTS is to replace PVSS with the information-theoretically secure Shamir's Secret Sharing (SSS) algorithm to share the secret data. Even though SSS protects the confidentiality of the shares against quantum attacks, unlike PVSS, it does not prevent Wendy from distributing bad secret shares as SSS does not provide a way of verifying the correctness of the distributed shares.

We address this problem by requiring the secret-management trustees to verify the correctness of the secret shares before Wendy publicly commits to them. To do so, Wendy prepares a commitment for each secret share and requests trustees to verify the consistency of their share against its commitment. Assuming a secret-recovery threshold of $t = f_{SMC} + 1$, we require at least $2f_{SMC} + 1$ trustees to verify the consistency of the shares and require the secret-management committee to consist of at least $n_{SMC} = 3f_{SMC} + 1$ trustees. These thresholds guarantee that at least t honest trustees have verified the shares, which is necessary for reconstructing the secret. Consequently, if Ron fails to decrypt the message using the recovered secret at the end of the protocol, he can hold Wendy accountable for not sharing a valid tx_w.

4.5.1 PQ-OTS subprotocols

Before we describe the subprotocols of PQ-OTS (Figure 4.3), we revisit our security assumptions since we are now operating in a post-quantum world. For PQ-OTS, we assume that the communication channels are quantum safe. We further assume that the quantum-safe versions of basic cryptographic primitives (*e.g.*, digital signatures) are deployed. In terms of notation, we use y_i to denote the public key of trustee i, which is not computed as $y_i = G^{x_i}$ this time.

Write subprotocol

Wendy prepares a write transaction (tx_w) as follows:

1. Chooses a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The secret to be shared is $S = s(0)$.
2. For each trustee i, generates a commitment $com_i = \text{H}(s(i) \parallel r_i)$, where r_i is a random value.
3. Computes $\text{H}(S)$ and uses it as the symmetric key k to encrypt the secret message m as $c = \text{enc}_k(m)$. Computes the hash of the ciphertext $H_c = \text{H}(c)$, which can be used to check the integrity of the ciphertext. Sets policy to designate Ron as the intended reader of the secret message m.
4. Prepares and signs $\text{tx}_\text{w} = [\langle com_i \rangle, H_c, \langle y_i \rangle, \text{policy}]_{\text{sig}_W}$.
5. Sends the share $s(i)$, random value r_i and tx_w to each trustee i.

Each trustee i computes $\text{H}(s(i) \parallel r_i)$ and compares it against com_i, which is stored in tx_w. If

they match, the trustee stores $s(i)$, and sends their signature over tx_w to Wendy. Once Wendy receives the signatures, she sends tx_w and the signatures to the access-control blockchain. Finally, the access-control blockchain logs tx_w if it can verify that at least $2f_{SMC}+1$ trustees have signed tx_w.

Read, share retrieval, and secret reconstruction subprotocol

The remaining subprotocols work the same way as those in OTS except the trustees do not create NIZK proofs of consistent decryption, which means Ron does not have to verify any NIZK proofs. If Ron cannot decrypt m after reconstructing the shared secret and recovering k, it means that Wendy misbehaved. In this case, Ron can publicly reveal the output of each subprotocol to prove that Wendy misbehaved.

4.5.2 Discussion

We finish our discussion on PQ-OTS by briefly discussing its advantages, shortcomings, and success in achieving the system goals. PQ-OTS has the same advantages as OTS in terms of the ad hoc setup of the secret-management committee. One advantage of PQ-OTS over OTS is that it reduces the computational load on Wendy since SSS does not require generating NIZK proofs and polynomial commitments. However, PQ-OTS requires an additional round trip in the write subprotocol since Wendy needs to collect the signatures of trustees on tx_w. Finally, PQ-OTS achieves all of our system goals assuming that the underlying cryptographic primitives and the communication channels are quantum safe. Otherwise, PQ-OTS cannot guarantee the confidentiality of secrets.

4.6 Implementation

We implemented a prototype of on-chain secrets with OTS and PQ-OTS as the underlying protocols in ~1100 and ~1350 LoC, respectively, in Go. Additionally, we implemented two real-world applications, namely a clearance-enforcing document sharing application and a zero-collateral decentralized lottery, on top of on-chain secrets to evaluate its performance. We build the access-control blockchain on top of the ByzCoin blockchain [30]. We use Kyber [55] and the Go standard library for all the cryptographic primitives and operations. More specifically, we use Kyber's implementation of the Edwards25519 curve for elliptic-curve cryptography operations, and implementation of the PVSS and SSS protocols. We use the SHA-256 and AES-256 implementations in the Go crypto package. Finally, we use Onet [196] for creating an overlay network between the trustees and deploying our protocols.

In the document-sharing application, user A shares a confidential document with user B. To do so, A uses the shared secret from PVSS/SSS to generate a symmetric key, encrypts the document with the symmetric key, and sends a tx_w, which specifies B as an authorized reader,

to the access-control blockchain. B receives the encrypted document from A, sends a tx_r to the access-control blockchain, retrieves the decrypted shares from the secret-management committee, and recovers the symmetric key to decrypt the document. For comparison, we implement the same application on top of a state-of-the-art semi-centralized system [226] that uses a blockchain to enforce access control policies and a trusted server to entrust the data.

In the zero-collateral decentralized lottery, we implement a commitment-based approach where participants' inputs are used to generate an unpredictable random value that determines the winner of the lottery. We assume that there is a smart contract that collects the bids and inputs from the users and decides on the winner. Each participant encrypts their random input, creates a tx_w, and logs it on the blockchain by following the same steps as in the document-sharing application. After a predefined barrier point is reached, the lottery closes for new inputs. At this point, any authorized user can send tx_rs to the blockchain and retrieve the decrypted shares and their proofs. Finally, the smart contract can recover the original random inputs of the participants and XOR them to generate a random value, which is used for selecting the winner. For comparison, we implement a state-of-the-art zero-collateral lottery by Miller and Bentov [149] on top of the ByzCoin blockchain.

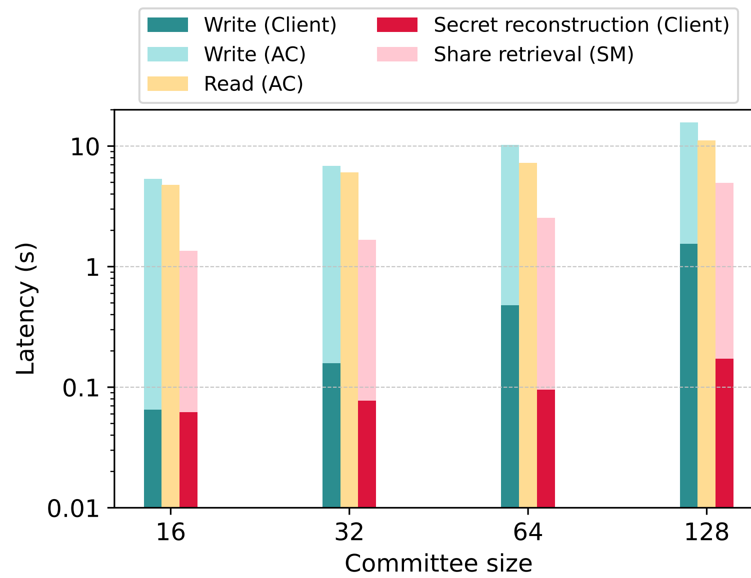
4.7 Evaluation

In our experiments, we evaluate the performance of on-chain secrets with OTS and PQ-OTS protocols to answer the following questions:

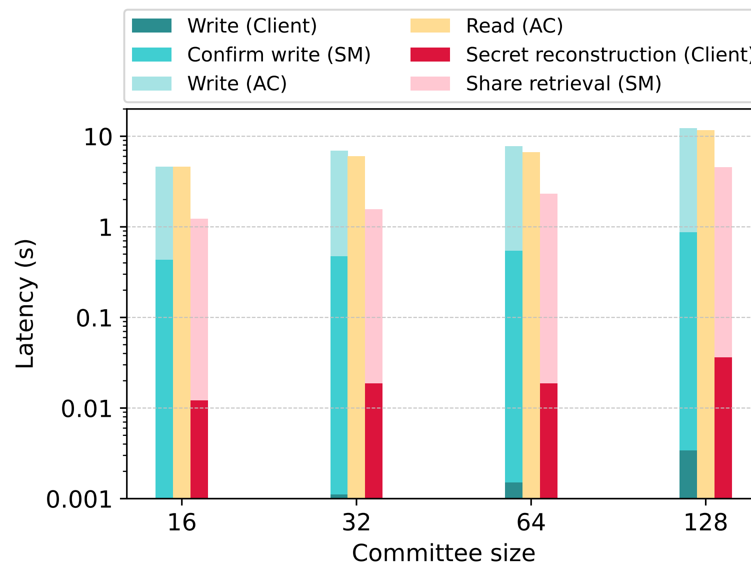
1. Can on-chain secrets scale to large numbers of trustees to achieve high levels of decentralization without sacrificing performance?
2. Can on-chain secrets achieve comparable performance to a semi-centralized system that trades off decentralization for performance?
3. Can on-chain secrets improve on the state-of-the-art approaches that implement popular decentralized applications such as lotteries?

4.7.1 Experimental setup

We run our experiments on a Ubuntu 20.04 VM instance that is equipped with 16 vCPU (Intel Xeon Silver 4216 CPU @ 2.10 GHz) and 64 GB RAM. We use Mininet to simulate a realistic network where the link delay is 100 ms and the link bandwidth is 100 Mbps. We assume that the same set of nodes run both the access-control blockchain and the secret-management committee. Unless stated otherwise, we use a committee size of 16.



(a) OTS



(b) PQ-OTS

Figure 4.4: Latency of on-chain secrets protocols for varying committee sizes

4.7.2 Level of decentralization vs. performance

In this experiment, we measure the total latency of OTS and PQ-OTS protocols for varying numbers of trustees to evaluate the impact of committee sizes on the performance. We report the latencies of the four subprotocols, namely write, read, share retrieval and secret reconstruction, separately. We further break down the total latency of a subprotocol into its client-side and committee-side components when possible. We assume that all trustees

belong to both the access-control blockchain and the secret-management committee.

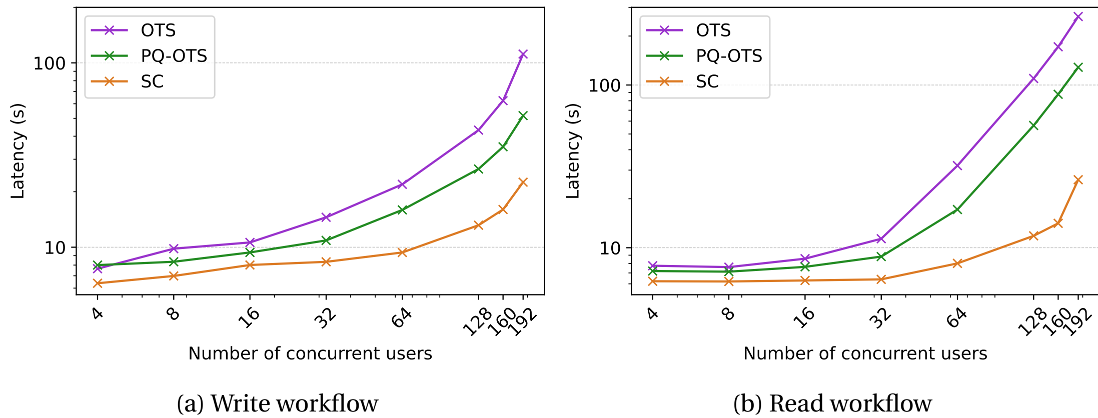
Figure 4.4a shows the results for OTS. We observe that the client-side latency of preparing a tx_w increases with the number of trustees and reaches to 1.5 s for a committee size of 128. These results are due to the computationally-heavy PVSS operations, such as creating the encrypted shares, the NIZK proofs, and the polynomial commitments, as we mention in §4.4.2. The client-side latency of secret reconstruction is at most ~0.2 s and is almost an order of magnitude less than the latency of preparing a tx_w for a committee size of 128. The difference is due to the fewer elliptic-curve cryptography operations required at secret reconstruction. Finally, we observe that the access-control blockchain performance scales well with the committee size, as it takes 2.7× and 2.4× more time to log a tx_w and tx_r, respectively, when the committee size increases by 8×. Moreover, it takes the access-control blockchain $1.1 - 1.35\times$ longer to log a tx_w than a tx_r. This is because the access-control blockchain has to verify the encrypted PVSS shares against the NIZK proofs before logging the tx_w.

Figure 4.4b shows the results for PQ-OTS. We observe that the client-side latency of preparing a tx_w is significantly smaller ($\sim 85 - 450\times$) than that of OTS because PQ-OTS uses SSS and hash-based commitments, which have a lower setup cost than PVSS. We also observe that the additional step in PQ-OTS (Confirm write) where the writer collects signatures from the secret-management committee trustees before posting the tx_w takes between 0.5 and 1 s. Additionally, logging a tx_w is up to 26% faster compared to OTS. Therefore, we observe that the total write latency of PQ-OTS is smaller than that of OTS. The secret reconstruction and share retrieval subprotocols are also faster in PQ-OTS. Share retrieval is faster in PQ-OTS because the secret-management committee trustees do not generate NIZK proofs of consistent decryption. Secret construction is faster because the client does not verify NIZK proofs of decryption. As a result, the decryption phase in PQ-OTS is up to ~10% faster than OTS.

Our experiment results demonstrate the latency overhead of OTS and PQ-OTS due to their use of threshold cryptography. We argue that these overheads are acceptable since the total latency is dominated by the block time, which is in the order of seconds or minutes in the existing blockchain systems. Moreover, our results also show that both protocols can scale with the number of trustees to achieve high degrees of decentralization.

4.7.3 On-chain secrets vs. a semi-centralized system

We use the clearance-enforcing document sharing application to compare on-chain secrets to a state-of-the-art semi-centralized system. We define two workflows for this application: write and read. The write workflow consists of the write transaction protocol. The read workflow consists of the read transaction, share retrieval, and secret reconstruction protocols. As in on-chain secrets, the blockchain is used for logging the access requests and enforcing the access control policies. The difference in the semi-centralized system is that confidential data is encrypted under the public key of a single trusted server instead of a collective key managed by a committee of nodes.



(a) Write workflow

(b) Read workflow

Figure 4.5: Average write and read workflow latencies under different loads in clearance-enforcing document sharing

In our first experiment, we use a synthetic workload where users concurrently and independently execute the write and read workflows as fast as they can. We vary the number of concurrent users in the workload from 4 to 192 and report the average of user-observed latencies computed over five experiment runs. Since we cannot get healthy measurements when we set the number of concurrent users to 256, we reduce the step size after 128 to collect more data points.

Figure 4.5a shows the average latencies for the write workflow. OTS and PQ-OTS perform comparably to the semi-centralized (SC) system when there are up to 16 concurrent users. The performance gap between the on-chain secrets protocols and the semi-centralized widens as we increase the total load in the system. OTS is 74% slower than the semi-centralized system for 32 users. It takes PQ-OTS a higher load to reach the same level of slowdown as it is 70% slower than the semi-centralized system for 64 users. For the maximum load of 192 users, OTS takes 112 s and it is 2.2× and 5× slower than PQ-OTS and the semi-centralized system, respectively.

Figure 4.5b shows the average latencies for the read workflow. Once again, both on-chain secrets protocols have comparable performance to the semi-centralized system up to 16 users. However, the performance gap between the on-chain secrets protocols and the semi-centralized system is larger for read workflows. OTS and PQ-OTS are 77% and 114% slower than the semi-centralized system for 32 and 64 users, respectively. For the maximum load of 192 users, OTS takes 263 s and it is 2.1× and 10.1× slower than PQ-OTS and the semi-centralized system, respectively. The performance gap between the on-chain secrets protocols and the semi-centralized system is larger for the read workflow due to the share retrieval protocol run by the secret-management committee, which is replaced by a trusted server in the semi-centralized system.

Although the above results do not look good for the on-chain secrets protocols under heavy loads, we claim that our experimental setup has a significantly negative impact on the per-

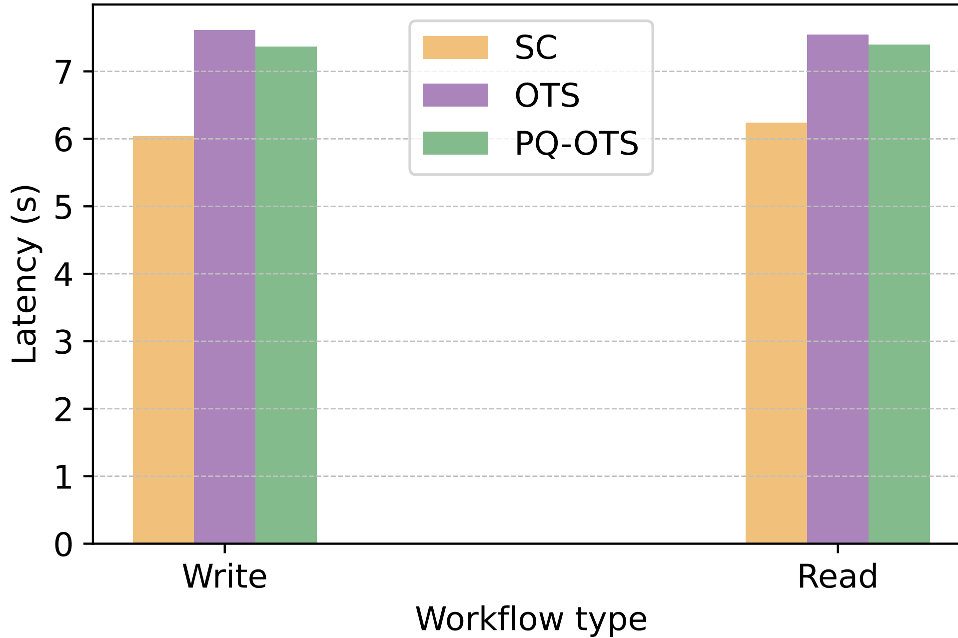


Figure 4.6: Average write and read workflow latencies for the replayed real-world data trace in clearance-enforcing document sharing

formance of on-chain secrets. We use a single physical machine to both deploy the trustees and simulate the network communication via Mininet. Therefore, as we increase the number of users in the system, the CPU becomes a bottleneck as clients, trustees and Mininet competes for CPU resources. Compared to on-chain secrets, the semi-centralized system requires less system resources since it employs fewer trustees and requires less network communication. Therefore, the semi-centralized system is impacted by the experimental setup less than on-chain secrets does.

In our second experiment, we evaluate the performance of the clearance-enforcing document sharing application using a data trace from a real-world deployment. We obtained the data trace from a contractor of the Ministry of Defense of a European country using on-chain secrets with a permissioned BFT blockchain. The data trace is collected from the company's testbed over a period of 15 days and includes 1821 tx_w and 1470 tx_r. The average, minimum, and maximum number of transactions per block are 2.62, 1, and 7, respectively. We replay the trace on on-chain secrets and the semi-centralized system by treating each block as a discrete event. We concurrently execute all the transactions in a block and wait for all of them to finish before we move on to the next block. We execute the write and read workflows for tx_w and tx_r, respectively.

Figure 4.6 shows the average latencies of tx_w and tx_r. We observe that both OTS and PQ-OTS have comparable performance to the semi-centralized system. PQ-OTS takes 22% and 18% longer than the semi-centralized system to execute the write and read workflows, respectively.

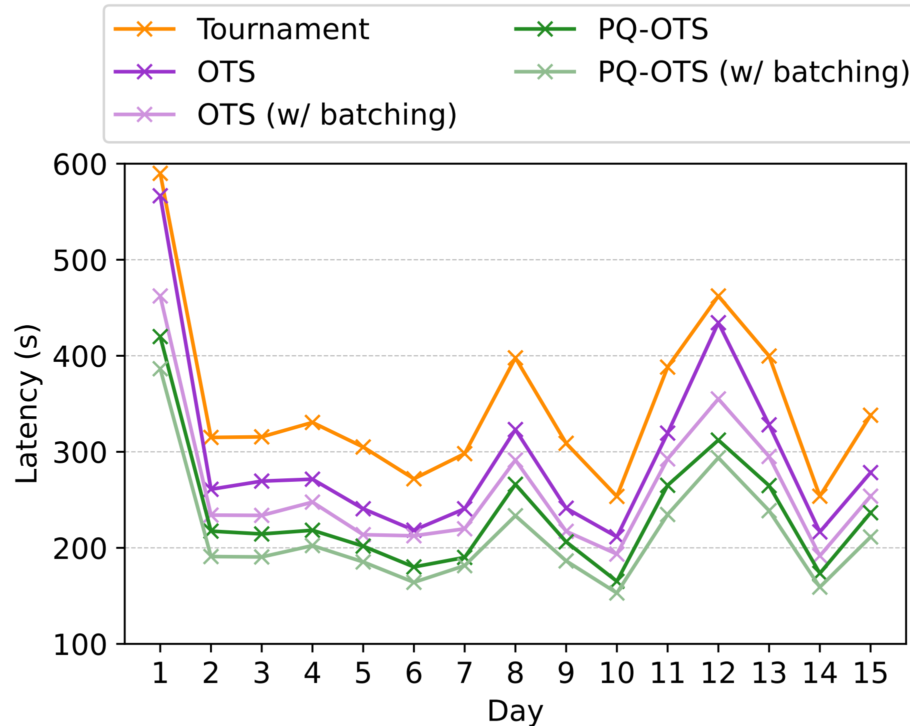


Figure 4.7: Lottery execution times using the Fire Lotto workload

OTS performs slightly worse than PQ-OTS as it takes 26% and 21% longer than the semi-centralized system for the write and read workflows, respectively. We highlight that these results are consistent with the results of the previous experiment at low system loads, which is expected considering the characteristics of the replayed data trace.

4.7.4 A decentralized lottery application with on-chain secrets

In our final experiment, we demonstrate how on-chain secrets can improve the performance of decentralized lotteries, which is a popular class of decentralized applications. To this end, we compare our lottery implementations based on on-chain secrets to a state-of-the-art lottery by Miller and Bentov [149] that does not require any collateral from the participants.

Miller and Bentov's protocol is based on a binary-tree tournament structure and requires $O(\log N)$ rounds, where N is the number of participants. At each round, participants compete against each other in pairs by running a commit-reveal protocol. The winner of each pair advances to the next round where they run the commit-reveal protocol again, in pairs. The process continues until there is only one participant left, who becomes the winner of the lottery. In comparison, our on-chain secrets lotteries use a commit-reveal protocol that runs in one round. The round begins with all participants sending their encrypted commitments to on-chain secrets and ends with all commitments getting decrypted and revealed at the same

time.

We evaluate both a simple and an optimized implementation of the on-chain secrets lotteries. In the simple implementation, we send a separate decryption request to the secret-management committee for each commitment one after the other. In the optimized implementation, we send a single batch request that returns all decrypted commitments. We use a real-world trace based on Fire Lotto [194], which is an Ethereum-based decentralized lottery platform, to evaluate the performance of the lottery implementations. We collect transactions sent to the Fire Lotto smart contract on Ethereum over a period of 15 days, where each day is a separate run of the lottery. The average, minimum, and maximum number of transactions per day are 83.4, 61, and 142, respectively.

Figure 4.7 shows the total execution time of the daily lottery runs for each day in our trace. We only report the latency of the reveal phase, where the commitments are opened and the winner is picked, since the commitment phase lasts for hours and participants can send in their commitments whenever they want during that period. The simple OTS- and PQ-OTS-based lotteries are 15% and 25% faster than the tournament lottery on average, respectively. On-chain secrets lotteries have an additional overhead due to decrypting the commitments, but the tournament lottery is slower due to having $O(\log N)$ times more interaction with the blockchain, which has a higher overhead. Batching the decryption requests further improves the performance as the OTS- and PQ-OTS-based lotteries are 32% and 38% faster than the tournament lottery on average, respectively.

4.8 Related work

Previous work has proposed approaches for confidential sharing of sensitive user-data using blockchains. Zyskind et al. [226] proposes a system that enables users to selectively share their personal data with third-party services. They use the blockchain as an access-control manager and store encrypted data off-chain. Their system has a stronger security assumption as it assumes an honest-but-curious reader, unlike on-chain secrets where readers can act maliciously. Enigma [227] improves on [226] by using secure multi-party computation to split sensitive data among a group of nodes. Other systems [12], [64] use blockchains for auditable and transparent sharing of electronic medical records. Both systems fail to achieve full decentralization as they use a centralized off-chain component to store the sensitive data, even though they rely on the blockchain for managing data accesses and permissions. Finally, Droplet [177] is a decentralized access control system that supports sharing and management of streaming time-series data. Droplet uses an encryption-based access control mechanism where a data stream is divided into chunks based on time segments and each chunk is encrypted with a distinct key that is generated for that time segment. Droplet's authorization service builds on a blockchain to maintain the access control state and check the access permissions without a trusted entity.

One of the earliest works in the area of decentralized life-cycle management is Vanish [88],

which addresses the problem of irreversibly and automatically revoking access to all copies of a shared data without relying on a trusted party. Vanish encrypts data using a random encryption key; uses Shamir secret sharing to create shares of the key; and distributes the shares of the key to random nodes in a distributed hash table (DHT). Vanish relies on the DHT's property of periodically discarding old data to ensure that the distributed key shares will disappear from the DHT over time and the data will vanish as there will not be enough shares to recover the encryption key. Shortly after Vanish was published, researchers presented feasible attacks against Vanish where an adversary can recover encryption keys long after their supposed removal from the DHT [216]. We can build a Vanish-like system where users can revoke access to their shared data by combining on-chain secrets with a decentralized access-control system as described in CALYPSO [133].

Finally, researchers have proposed solutions that encrypt transactions to mitigate frontrunning attacks. FairBlock [150] employs identity-based encryption (IBE) to create an encryption key for each block height. Users encrypt their transactions to a future block using the key corresponding to the height of the future block. FairBlock replaces the trusted third party with a committee of nodes to avoid a single point of failure or compromise. Ferveo [14] uses threshold cryptography to make sure that the transaction contents remain encrypted until the transaction is finalized in a block. Other solutions [61], [131] use time-lock puzzles [169] to guarantee that transactions can only be decrypted in the future after a certain amount of time has passed.

4.9 Conclusion

We have presented on-chain secrets, a secure decentralized data-management framework that enables auditable and fair access to confidential data without relying on a trusted entity. on-chain secrets uses threshold cryptography to store confidential data using a committee of trustees, and the blockchain technology to manage accesses to confidential data by enforcing access control policies in a transparent and auditable manner. We design and implement two on-chain secrets protocols, OTS and PQ-OTS, to guarantee the confidentiality of secret data against today's adversaries and the quantum adversaries of the future, respectively. We evaluate our prototype of on-chain secrets using real-world applications under both synthetic and realistic workloads. Our results demonstrate that on-chain secrets can improve the security and fairness of decentralized applications while achieving acceptable performance without sacrificing decentralization.

5 Conclusion

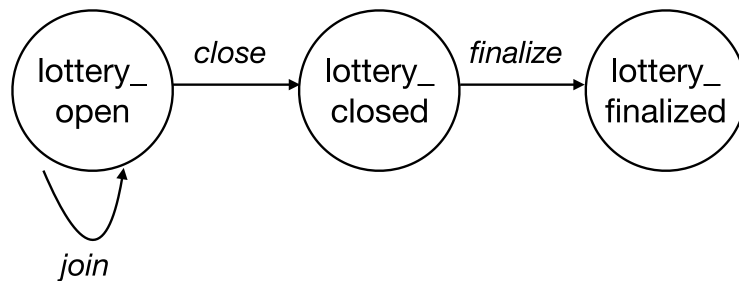
In this dissertation, we have presented systems to improve the functional richness of smart contracts. To this end, we have proposed both a novel grand-vision architecture idea in PROTEAN, and standalone systems in DEJAFLOAT and on-chain secrets to address specific technical challenges in the current smart contract systems. We have built prototypes of the proposed systems and evaluated their performance using real-world applications. Our experimental evaluation has shown that our systems are feasibly practical and can address the functional limitations of the existing smart contract systems. Moreover, we believe that the architectural vision of PROTEAN has the potential to accelerate innovation in the decentralized computing domain.

That being said, we think that there is still a lot of room for interesting future work. One of the important limitations of PROTEAN is its handling of concurrent transactions. It would be interesting to see how we can combine concurrency control techniques from the database systems (*e.g.*, multi-version concurrency control, deterministic databases) with the security requirements of smart contracts to build high-performance systems. Another important future work for PROTEAN is to create a domain-specific language for defining workflows in an expressive and programmer-friendly way. Another venue for future research is building on our experience with building a deterministic JVM sandbox to create a language-neutral deterministic sandbox for multiple general-purpose programming languages. Since this would require creating deterministic language profiles for several languages, it would be interesting to investigate how techniques from programming languages can be utilized to improve the efficiency and correctness of the deterministic sandbox.

A Workflows and FSM models of the PROTEAN applications

A.1 randlotto

A.1.1 FSM model



A.1.2 Workflows


```
{
  "join":{
    "txns":{
      "join":{
        "opcodes":[
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"join_randlotto"
              },
              "readset":{
```

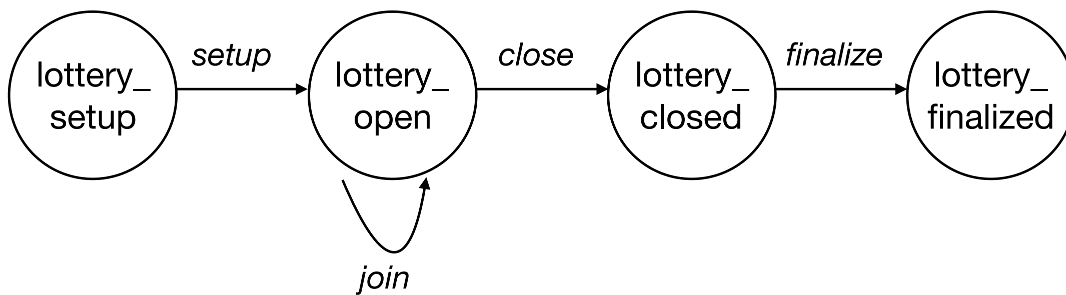
```
                "src":"KEYVALUE",
                "value":"tickets,curr_stage"
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":0
              }
            }
          }
        ]
      }
    }
  },
  "close":{
    "txns":{
      "close":{
        "opcodes":[
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"close_randlotto"
              },
              "barrier":{
                "src":"CONST",
                "value":100
              },
              "readset":{
                "src":"KEYVALUE",
                "value":"curr_stage"
              }
            }
          },
```

```
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":0
              }
            }
          }
        ]
      }
    }
  },
  "finalize":{
    "txns":{
      "finalize":{
        "opcodes":[
          {
            "name":"get_randomness",
            "dfu_id":"rand_dfu",
            "inputs":{
              "round":{
                "src":"CONST",
                "value":10
              }
            }
          },
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"finalize_randlotto"
              },
              "round":{
                "src":"CONST",
                "value":10
              },
              "readset":{
```

```
              "src":"KEYVALUE",
              "value":"tickets,curr_stage"
            },
            "randomness":{
              "src":"OPCODE",
              "src_name":"randomness",
              "idx":0
            }
          }
        },
        {
          "name":"update_state",
          "dfu_id":"state_dfu",
          "inputs":{
            "ws":{
              "src":"OPCODE",
              "src_name":"writeset",
              "idx":1
            }
          }
        }
      ]
    }
  }
}
}
```

A.2 secretlotto

A.2.1 FSM model



A.2.2 Workflows


```
{
  "setup":{
    "txns":{
      "setup":{
        "opcodes":[
          {
            "name":"run_dkg",
            "dfu_id":"threshold_dfu"
          },
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"setup_secretlotto"
              },
              "pk":{
                "src":"OPCODE",
                "src_name":"pk",
                "idx":0
              },
              "readset":{
                "src":"KEYVALUE",
                "value":"curr_stage"
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":1
              }
            }
          }
        ]
```

```
      }
    }
  },
  "join":{
    "txns":{
      "join":{
        "opcodes":[
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"join_secretlotto"
              },
              "readset":{
                "src":"KEYVALUE",
                "value":"enc_tickets,curr_stage"
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":0
              }
            }
          }
        ]
      }
    }
  },
  "close":{
    "txns":{
      "close":{
        "opcodes":[
          {
            "name":"exec",
```

```
          "dfu_id":"exec_dfu",
          "inputs":{
            "fnname":{
              "src":"CONST",
              "value":"close_secretlotto"
            },
            "barrier":{
              "src":"CONST",
              "value":100
            },
            "readset":{
              "src":"KEYVALUE",
              "value":"curr_stage"
            }
          }
        },
        {
          "name":"update_state",
          "dfu_id":"state_dfu",
          "inputs":{
            "ws":{
              "src":"OPCODE",
              "src_name":"writeset",
              "idx":0
            }
          }
        }
      ]
    }
  }
},
"finalize":{
  "txns":{
    "finalize":{
      "opcodes":[
        {
          "name":"exec",
          "dfu_id":"exec_dfu",
          "inputs":{
            "fnname":{
              "src":"CONST",
              "value":"prepare_decrypt"
```

```
        },
        "readset":{
          "src":"KEYVALUE",
          "value":"enc_tickets"
        }
      }
    },
    {
      "name":"decrypt",
      "dfu_id":"threshold_dfu",
      "inputs":{
        "ciphertexts":{
          "src":"OPCODE",
          "src_name":"ciphertexts",
          "idx":0
        }
      }
    },
    {
      "name":"exec",
      "dfu_id":"exec_dfu",
      "inputs":{
        "fnname":{
          "src":"CONST",
          "value":"finalize_secretlotto"
        },
        "readset":{
          "src":"KEYVALUE",
          "value":"curr_stage"
        },
        "plaintexts":{
          "src":"OPCODE",
          "src_name":"plaintexts",
          "idx":1
        }
      }
    },
    {
      "name":"update_state",
      "dfu_id":"state_dfu",
      "inputs":{
        "ws":{
```

```
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":2
              }
            }
          }
        ]
      }
    }
  }
}
```


A.3 e-voting

A.3.1 FSM model

election_ setup → *setup* → election_ open → *lock* → election_ closed → *shuffle* → election_ shuffled → *tally* → election_ finalized

election_ open → *vote* → election_ open

A.3.2 Workflows


```
{
  "setup":{
    "txns":{
      "setup":{
        "opcodes":[
          {
            "name":"run_dkg",
            "dfu_id":"threshold_dfu"
          },
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
```

```
                "value":"setup_election"
              },
              "pk":{
                "src":"OPCODE",
                "src_name":"pk",
                "idx":0
              },
              "readset":{
                "src":"KEYVALUE",
                "value":"curr_stage"
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":1
              }
            }
          }
        ]
      }
    }
  },
  "vote":{
    "txns":{
      "vote":{
        "opcodes":[
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"vote"
              },
              "readset":{
                "src":"KEYVALUE",
```

```
                "value":"enc_ballots,curr_stage"
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":0
              }
            }
          }
        ]
      }
    }
  },
  "finalize":{
    "txns":{
      "lock":{
        "opcodes":[
          {
            "name":"exec",
            "dfu_id":"exec_dfu",
            "inputs":{
              "fnname":{
                "src":"CONST",
                "value":"lock"
              },
              "barrier":{
                "src":"CONST",
                "value":100
              },
              "readset":{
                "src":"KEYVALUE",
                "value":"lock,curr_stage"
              },
              "h":{
                "src":"PRECOMMIT",
                "value":"h"
```

```
        }
      }
    },
    {
      "name":"update_state",
      "dfu_id":"state_dfu",
      "inputs":{
        "ws":{
          "src":"OPCODE",
          "src_name":"writeset",
          "idx":0
        }
      }
    }
  ]
},
"shuffle":{
  "opcodes":[
    {
      "name":"exec",
      "dfu_id":"exec_dfu",
      "inputs":{
        "fnname":{
          "src":"CONST",
          "value":"prepare_shuffle"
        },
        "readset":{
          "src":"KEYVALUE",
          "value":"enc_ballots,h"
        }
      }
    },
    {
      "name":"shuffle",
      "dfu_id":"mixnet_dfu",
      "inputs":{
        "pairs":{
          "src":"OPCODE",
          "src_name":"pairs",
          "idx":0
        },
        "h":{
```

```
          "src":"OPCODE",
          "src_name":"h",
          "idx":0
        }
      }
    },
    {
      "name":"exec",
      "dfu_id":"exec_dfu",
      "inputs":{
        "fnname":{
          "src":"CONST",
          "value":"prepare_proofs"
        },
        "readset":{
          "src":"KEYVALUE",
          "value":"curr_stage"
        },
        "proofs":{
          "src":"OPCODE",
          "src_name":"proofs",
          "idx":1
        }
      }
    },
    {
      "name":"update_state",
      "dfu_id":"state_dfu",
      "inputs":{
        "ws":{
          "src":"OPCODE",
          "src_name":"writeset",
          "idx":2
        }
      }
    }
  ]
},
"tally":{
  "opcodes":[
    {
      "name":"exec",
```

```
    "dfu_id":"exec_dfu",
    "inputs":{
      "fnname":{
        "src":"CONST",
        "value":"prepare_decrypt_votes"
      },
      "readset":{
        "src":"KEYVALUE",
        "value":"proofs"
      }
    }
  },
  {
    "name":"decrypt",
    "dfu_id":"threshold_dfu",
    "inputs":{
      "ciphertexts":{
        "src":"OPCODE",
        "src_name":"ciphertexts",
        "idx":0
      }
    }
  },
  {
    "name":"exec",
    "dfu_id":"exec_dfu",
    "inputs":{
      "fnname":{
        "src":"CONST",
        "value":"tally_votes"
      },
      "readset":{
        "src":"KEYVALUE",
        "value":"curr_stage"
      },
      "plaintexts":{
        "src":"OPCODE",
        "src_name":"plaintexts",
        "idx":1
      },
      "candidate_count":{
        "src":"CONST",
```

```
                "value":10
              }
            }
          },
          {
            "name":"update_state",
            "dfu_id":"state_dfu",
            "inputs":{
              "ws":{
                "src":"OPCODE",
                "src_name":"writeset",
                "idx":2
              }
            }
          }
        ]
      }
    }
  }
}
```

Bibliography

[1] M. Abadi, P. Barham, J. Chen, *et al.*, TensorFlow: A System for Large-Scale Machine Learning, in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[2] E. C. Alp, C. Băsescu, P. N. Tennage, N. Kocher, G. Bosson, and B. A. Ford, Efficient Deterministic Execution of Smart Contracts, Tech. Rep., 2022.

[3] E. C. Alp, E. Kokoris-Kogias, G. Fragkouli, and B. Ford, Rethinking General-Purpose Decentralized Computing, in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019, pp. 105–112.

[4] 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*, 2018, 30:1–30:15.

[5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek, Secure Multiparty Computations on Bitcoin, *Communications of the ACM*, vol. 59, no. 4, pp. 76–84, 2016.

[6] A. Arteaga, O. Fuhrer, and T. Hoefler, Designing Bit-Reproducible Portable High-Performance Applications, in *28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1235–1244.

[7] ASM, accessed on 2023-07-18.

[8] 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*, 2017, pp. 164–186.

[9] A. Aviram and B. Ford, Deterministic OpenMP for Race-Free Parallelism, in *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011.

[10] A. Aviram, S. Hu, B. Ford, and R. Gummadi, Determinating Timing Channels in Compute Clouds, in *ACM Cloud Computing Security Workshop (CCSW)*, 2010.

[11] A. Aviram, S. Weng, S. Hu, and B. Ford, Efficient System-Enforced Deterministic Parallelism, in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2010, pp. 193–206.

[12] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, MedRec: Using Blockchain for Medical Data Access and Permission Management, in *2nd International Conference on Open and Big Data (OBD)*, 2016, pp. 25–30.

[13] D. H. Bailey, R. Barrio, and J. M. Borwein, High-Precision Computation: Mathematical Physics and Dynamics, *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, 2012.

[14] J. Bebel and D. Ojha, Ferveo: Threshold Decryption for Mempool Privacy in BFT networks, *Cryptology ePrint Archive, Paper 2022/898*, 2022.

[15] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, Zerocash: Decentralized Anonymous Payments from Bitcoin, in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 459–474.

[16] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware, in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1521–1538.

[17] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution, in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 53–64.

[18] T. Bergan, J. Devietti, N. Hunt, and L. Ceze, The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?, in *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET)*, 2011.

[19] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, Deterministic Process Groups in dOS, in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[20] E. D. Berger, T. Yang, T. Liu, and G. Novark, Grace: Safe Multithreaded Programming for C/C++, in *24th ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2009.

[21] W. Binder and J. Hulaas, Exact and Portable Profiling for the JVM Using Bytecode Instruction Counting, *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 3, pp. 45–64, 2006.

[22] W. Binder and J. Hulaas, Using Bytecode Instruction Counting as Portable CPU Consumption Metric, *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 57–77, 2006.

[23] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, A Type and Effect System for Deterministic Parallel Java, in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[24] D. Boneh, X. Boyen, and S. Halevi, Chosen Ciphertext Secure Public Key Threshold Encryption Without Random Oracles, in *The Cryptographers' Track at the RSA Conference (CT-RSA)*, ser. Lecture Notes in Computer Science, vol. 3860, 2006, pp. 226–243.

[25] D. Boneh, M. Drijvers, and G. Neven, Compact Multi-signatures for Smaller Blockchains, in *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security*, 2018, pp. 435–464.

[26] D. Boneh, B. Lynn, and H. Shacham, Short signatures from the Weil pairing, in *International Conference on the Theory and Application of Cryptology and Information Security*, 2001, pp. 514–532.

[27] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, ZEXE: Enabling Decentralized Private Computation, in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 947–964.

[28] L. Breidenbach, T. Kell, S. Gosselin, and S. Eskandari, LibSubmarine: Defeat Front-Running on Ethereum, 2018, accessed on 2023-06-22.

[29] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, Zether: Towards Privacy in a Smart Contract World, in *Financial Cryptography and Data Security (FC)*, 2020, pp. 423–443.

[30] ByzCoin, accessed on 2023-06-24.

[31] 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)*, 2000.

[32] C. Cadwalladr and E. Graham-Harrison, Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach, Mar. 17, 2018.

[33] M. Castro and B. Liskov, Practical Byzantine Fault Tolerance, in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[34] M. Castro and B. Liskov, Practical Byzantine Fault Tolerance and Proactive Recovery, *ACM Transactions on Computer Systems (TOCS)*, 2002.

[35] D. Chaum, Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms, *Communications of the ACM*, vol. 24, 2 1981.

[36] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, Detecting Covert Timing Channels with Time-Deterministic Replay, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 541–554.

[37] B. Chen, M. Wen, Y. Shi, D. Lin, G. K. Rajbahadur, Z. Ming, *et al.*, Towards Training Reproducible Deep Learning Models, *arXiv preprint arXiv:2202.02326*, 2022.

[38] 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 Contracts, in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2019, pp. 185–200.

[39] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, and G. L. Lee, Determinism and Reproducibility in Large-Scale HPC Systems, in *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2013.

[40] J.-D. Choi and H. Srinivasan, Deterministic replay of Java multithreaded applications, in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT*, 1998, pp. 48–59.

[41] J. Chow, T. Garfinkel, and P. M. Chen, Decoupling dynamic program analysis from execution in virtual environments, in *USENIX Annual Technical Conference (ATC)*, 2008.

[42] M. Ciampi, M. Ishaq, M. Magdon-Ismail, R. Ostrovsky, and V. Zikas, FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker, in *Cyber Security, Cryptology, and Machine Learning - 6th International Symposium (CSCML)*, 2022, pp. 428–446.

[43] D. Clausen, MicroFloat, accessed on 2023-07-18.

[44] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults, in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[45] Common Patterns — Solidity 0.8.20 documentation, accessed on 2023-06-22.

[46] M. J. Corden and D. Kreitzer, Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?, *Intel Corp., Software Solutions Group, Santa Clara, CA, USA, Tech. Rep*, 2009.

[47] A. U. P. P. Council, Statement on Algorithmic Transparency and Accountability, *Commun. ACM*, 2017.

[48] H. Cui, J. Wu, and J. Yang, Stable Deterministic Multithreading Through Schedule Memoization, in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

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

[50] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges, *arXiv preprint arXiv:1904.05234*,

[51] J. Darcy, JEP 306: Restore Always-Strict Floating-Point Semantics, accessed on 2023-07-15.

[52] B. Dawson, Intermediate Floating-Point Precision, accessed on 2023-07-19.

[53] F. De Dinechin, C. Lauter, and J.-M. Muller, Fast and Correctly Rounded Logarithms in Double-Precision, *RAIRO - Theoretical Informatics and Applications (RAIRO: ITA)*, vol. 41, no. 1, pp. 85–102, 2007.

[54] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[55] DEDIS Advanced Crypto Library for Go, accessed on June 24, 2023.

[56] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, DMP: Deterministic Shared Memory Multiprocessing, in *14th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[57] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, RCDC: A Relaxed Consistency Deterministic Computer, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 67–78, 2011.

[58] R. Dillet, Google and Mastercard reportedly partner to track offline purchases, Aug. 31, 2018.

[59] R. Dingledine, N. Mathewson, and P. Syverson, Tor: The Second-Generation Onion Router, in *12th USENIX Security Symposium*, 2004.

[60] djvm: Deterministic Sandbox for the JVM.

[61] Y. Doweck and I. Eyal, Multi-Party Timed Commitments, *arXiv preprint arXiv:2005.04883*, 2020.

[62] Drand - Distributed Randomness Beacon, accessed on 2023-07-07.

[63] S. Dredge, Yes, those free health apps are sharing your data with other companies, Sep. 3, 2013.

[64] A. Dubovitskaya, Z. Xu, S. Ryu, M. Schumacher, and F. Wang, Secure and Trustable Electronic Medical Records Sharing using Blockchain, in *American Medical Informatics Association Annual Symposium (AMIA)*, 2017.

[65] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay, in *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[66] Eclipse Java development tools (JDT), accessed on 2023-07-18.

[67] B. Einarsson, B. Wichmann, K. W. Dritz, C. C. Douglas, H. P. Langtangen, V. Snyder, R. F. Boisvert, and R. Pozo, Chapter 8. General Methods for Implementing Reliable and Correct Software, in *Accuracy and Reliability in Scientific Computing*, 2005, pp. 125–171.

[68] EIP-1829: Precompile for Elliptic Curve Linear Combinations , accessed on 2023-07-19, 2019.

[69] EIP-5656: MCOPY - Memory copying instruction , accessed on 2023-07-19, 2021.

[70] EIP-616: SIMD Operations for the EVM , accessed on 2023-07-19, 2017.

[71] EIP-665: Add precompiled contract for Ed25519 signature verification, accessed on 2023-07-19, 2018.

[72] M. D. Ekstrand, D. Kluver, F. M. Harper, and J. A. Konstan, Letting Users Choose Recommender Algorithms: An Experimental Study, in *Proceedings of the 9th ACM Conference on Recommender Systems (RecSys)*, 2015, pp. 11–18.

[73] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. Riedl, Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit, in *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2011, pp. 133–140.

[74] S. Eskandari, S. Moosavi, and J. Clark, SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain, in *Financial Cryptography and Data Security (FC) Workshops, VOTING and WTSC*, 2019, pp. 170–189.

[75] S. Eskandari, S. Moosavi, and J. Clark, SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain, in *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC*, 2019.

[76] European Parliament and Council of the European Union, Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016.

[77] Ethereum flavored WebAssembly (ewasm).

[78] Fdlibm, accessed on 2023-07-18.

[79] Federal Office for Information Security, Migration to Post Quantum Cryptography, Aug. 24, 2020.

[80] Federal Office for Information Security, Quantum-safe cryptography – fundamentals, current developments and recommendations, May 18, 2022.

[81] I. Ferguson, What is DeFi and How is AI Creating New Opportunities In the Sector?, 2018.

[82] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, Verifiable Functional Purity in Java, in *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008, pp. 161–174.

[83] A. Fog, NaN payload propagation - unresolved issues, 2018.

[84] B. Ford, Plugging Side-Channel Leaks with Timing Information Flow Control, in *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[85] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding, *ACM Trans. Math. Softw.*, vol. 33, no. 2, p. 13, 2007.

[86] T. E. Gamal, A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, in *Advances in Cryptology, Proceedings of CRYPTO '84*, ser. Lecture Notes in Computer Science, vol. 196, 1984, pp. 10–18.

[87] H. Ge, S. Y. Chau, V. E. Gonsalves, H. Li, T. Wang, X. Zou, and N. Li, Koinonia: Verifiable e-Voting with Long-Term Privacy, in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 270–285.

[88] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, Vanish: Increasing Data Privacy with Self-Destructing Data, in *18th USENIX Security Symposium*, 2009, pp. 299–316.

[89] D. Geels, G. Altekar, S. Shenker, and I. Stoica, Replay Debugging for Distributed Applications, in *USENIX Annual Technical Conference (ATC)*, 2006.

[90] A. E. Gencer, R. van Renesse, and E. G. Sirer, Short Paper: Service-Oriented Sharding for Blockchains, in *International Conference on Financial Cryptography and Data Security*, 2017, pp. 393–401.

[91] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, Secure Distributed Key Generation for Discrete-Log Based Cryptosystems, in *Advances in Cryptology - EUROCRYPT, International Conference on the Theory and Application of Cryptographic Techniques*, 1999, pp. 295–310.

[92] P. Goel, Is blockchain the solution for failing global healthcare?, Sep. 16, 2022.

[93] V. Goel and N. Perlroth, Yahoo Says 1 Billion User Accounts Were Hacked, Dec. 14, 2016.

[94] The Go Programming Language, Feb. 2018.

[95] D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[96] S. Goldwasser and S. Park, Public Accountability vs. Secret Laws: Can They Coexist?: A Cryptographic Proposal, in *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, 2017, pp. 99–110.

[97] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, The Java Language Specification - Java SE 17 Edition, 2021.

[98] J. Gosling and H. McGilton, The Java Language Environment - A White Paper, 1995.

[99] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, Bringing the Web up to Speed with WebAssembly, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, 2017, pp. 185–200.

[100] Y. Han, Y. Zhang, and S. H. Vermund, Blockchain Technology for Electronic Health Records, *International Journal of Environmental Research and Public Health*, vol. 19, no. 23, 2022.

[101] J. D. Harris, Analysis of Models for Decentralized and Collaborative AI on Blockchain, in *Blockchain - ICBC 2020 - Third International Conference, Held as Part of the Services Conference Federation, SCF 2020*, ser. Lecture Notes in Computer Science, vol. 12404, 2020, pp. 142–153.

[102] J. D. Harris and B. Waggoner, Decentralized and Collaborative AI on Blockchain, in *IEEE International Conference on Blockchain*, 2019, pp. 368–375.

[103] K. F. Hollis, To Share or Not to Share: Ethical Acquisition and Use of Medical Data, in *Summit on Clinical Research Informatics, (CRI)*, 2016.

[104] Blockchain in Digital Identity, accessed on 2023-06-21.

[105] IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[106] EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine.

[107] The Solidity Contract-Oriented Programming Language, 2021.

[108] Wrong signs on division producing NaN #55131, 2018.

[109] Document guarantees (or lack thereof) regarding sign, quietness, and payload of NaNs #73328, 2020.

[110] The California Consumer Privacy Act of 2018, 2018.

[111] Ocean Protocol: Tools for the Web3 Data Economy, Sep. 1, 2022.

[112] SingularityDAO: AI-powered DeFi Crypto Portfolios.

[113] NIST Announces First Four Quantum-Resistant Cryptographic Algorithms, Jul. 5, 2022.

[114] NSA Releases Future Quantum-Resistant (QR) Algorithm Requirements for National Security Systems, Sep. 7, 2022.

[115] Data protection within new technologies: blockchain, 2019.

[116] National Quantum Initiative, accessed on 2023-06-21.

[117] FACT SHEET: President Biden Announces Two Presidential Directives Advancing Quantum Technologies, May 4, 2022.

[118] Building Value with Blockchain Technology: How to Evaluate Blockchain's Benefits, Jul. 16, 2019.

[119] N. Hynes, D. Dao, D. Yan, R. Cheng, and D. Song, A Demonstration of Sterling: A Privacy-Preserving Data Marketplace, *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2086–2089, 2018.

[120] Introduction to Chainlink VRF, accessed on 2023-07-15.

[121] Introduction to Web3, accessed on 2023-08-30, 2023.

[122] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007, pp. 59–72.

[123] Java Platform, Standard Edition & Java Development Kit Version 17 API Specification: Class Double, accessed on 2023-07-15.

[124] Java Platform, Standard Edition & Java Development Kit Version 17 API Specification: Class Float, accessed on 2023-07-15.

[125] Z. Jiang, The Data-Sharing Advantage: A Strategy For Unrestricted Innovation, Nov. 1, 2021.

[126] A. E. Kaafarani, Who Are The Early Adopters Of New Quantum-Secure Cryptography?, Jan. 6, 2023.

[127] B. Kacsmar, K. Tilbury, M. Mazmudar, and F. Kerschbaum, Caring about Sharing: User Perceptions of Multiparty Data Sharing, in *31st USENIX Security Symposium*, 2022, pp. 899–916.

[128] W. Kahan, A logarithm too clever by half, 2004.

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

[130] A. Kate and I. Goldberg, Distributed Key Generation for the Internet, in *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2009, pp. 119–128.

[131] R. Khalil, A. Gervais, and G. Felley, TEX - A Securely Scalable Trustless Exchange, *Cryptology ePrint Archive, Paper 2019/265*, 2019.

[132] C. Killer, B. Rodrigues, E. J. Scheid, M. Franco, M. Eck, N. Zaugg, A. Scheitlin, and B. Stiller, Provotum: A Blockchain-based and End-to-end Verifiable Remote Electronic Voting System, in *IEEE 45th Conference on Local Computer Networks (LCN)*, 2020, pp. 172–183.

[133] E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, CALYPSO: Private Data Management for Decentralized Ledgers, *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 586–599, 2020.

[134] 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.

[135] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding, in *39th IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 19–34.

[136] R. B. Konuru, H. Srinivasan, and J. Choi, Deterministic Replay of Distributed Java Applications, in *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS)*, 2000, pp. 219–228.

[137] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts, in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 839–858.

[138] L. M. Krebs, O. L. A. Rodriguez, P. Dewitte, J. Ausloos, D. Geerts, L. Naudts, and K. Verbert, Tell Me What You Know: GDPR Implications on Designing Transparency and Accountability for News Recommender Systems, in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[139] P. Li, D. Gao, and M. K. Reiter, Stopwatch: A Cloud Architecture for Timing Channel Mitigation, *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 2, pp. 1–28, 2014.

[140] T. Lindholm, F. Yellin, G. Bracha, and D. S. Alex Buckley, The Java Virtual Machine Specification - Java SE 17 Edition, 2021.

[141] T. Liu, C. Curtsinger, and E. Berger, DTHREADS: Efficient Deterministic Multithreading, in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 327–336.

[142] LLVM Language Reference Manual, accessed on 2023-09-24.

[143] D. Lucchetti, S. K. Reinhardt, and P. M. Chen, ExtraVirt: Detecting and Recovering from Transient Processor Faults, in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 1–8.

[144] A. Lucic, M. J. R. Bleeker, S. Jullien, S. Bhargav, and M. de Rijke, Reproducibility as a Mechanism for Teaching Fairness, Accountability, Confidentiality, and Transparency in Artificial Intelligence, 11, vol. 36, 2022, pp. 12 792–12 800.

[145] W. Lueks, Security and Privacy via Cryptography Having your cake and eating it too, Ph.D. dissertation, 2017.

[146] A. Mavridou and A. Laszka, Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach, in *Financial Cryptography and Data Security (FC)*, 2018, pp. 523–540.

[147] 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*, 2017, pp. 357–375.

[148] R. C. Merkle, A Digital Signature Based on a Conventional Encryption Function, in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques*, ser. Lecture Notes in Computer Science, vol. 293, pp. 369–378.

[149] A. Miller and I. Bentov, Zero-Collateral Lotteries in Bitcoin and Ethereum, in *IEEE European Symposium on Security and Privacy Workshops (EuroS&P Workshops)*, 2017, pp. 4–13.

[150] P. Momeni, S. Gorbunov, and B. Zhang, FairBlock: Preventing Blockchain Front-running with Minimal Overheads, *Cryptology ePrint Archive, Paper 2022/1066*, 2022.

[151] D. Monniaux, The Pitfalls of Verifying Floating-Point Computations, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, pp. 1–41, 2008.

[152] R. Mortier, J. Zhao, J. Crowcroft, *et al.*, Personal Data Management with the Databox: What's Inside the Box?, in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking (CAN@CoNEXT)*, 2016, pp. 49–54.

[153] MPFR Java Bindings, accessed on 2023-07-18.

[154] J.-M. Muller, On the definition of ulp (x), Ph.D. dissertation, INRIA, LIP, 2005.

[155] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, *et al.*, *Handbook of Floating-Point Arithmetic*. Springer, 2018.

[156] I. Müller, A. Arteaga, T. Hoefler, and G. Alonso, Reproducible Floating-Point Aggregation in RDBMSs, in *34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1049–1060.

[157] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, CIEL: A Universal Execution Engine for Distributed Data-Flow Computing, in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[158] P. Nagarajan, G. Warnell, and P. Stone, The Impact of Nondeterminism on Reproducibility in Deep Reinforcement Learning, 2018.

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

[160] National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS) 180-4, Secure Hash Standard (SHS), accessed on 2023-09-01, 2015.

[161] 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 (CCS)*, 2001, pp. 116–125.

[162] 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*, 2017, pp. 1271–1287.

[163] Numerics — WebAssembly 2.0, accessed on 2023-09-01, 2023.

[164] On-chain programs: Float Support.

[165] D. J. Pearce, JPure: A Modular Purity System for Java, in *20th International Conference on Compiler Construction*, 2011, pp. 104–123.

[166] T. P. Pedersen, A Threshold Cryptosystem without a Trusted Party, in *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques*, 1991, pp. 522–526.

[167] B. A. Plale, T. Malik, and L. C. Pouchard, Reproducibility Practice in High-Performance Computing: Community Survey Results, *Computing in Science & Engineering*, vol. 23, no. 5, pp. 55–60, 2021.

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

[169] R. L. Rivest, A. Shamir, and D. A. Wagner, Time-Lock Puzzles and Timed-Release Crypto, Tech. Rep., 1996.

[170] J. Rodriguez, When DeFi Becomes Intelligent, 2018.

[171] 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)*, 2001.

[172] J. Scheibner, J. L. Raisaro, J. R. Troncoso-Pastoriza, M. Ienca, J. Fellay, E. Vayena, and J. Hubaux, Revolutionizing Medical Data Sharing Using Advanced Privacy Enhancing Technologies: Technical, Legal and Ethical Synthesis, *arXiv preprint arXiv:2010.14445*, 2020.

[173] C. Schnorr, Efficient Identification and Signatures for Smart Cards, in *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science, vol. 435, 1989, pp. 239–252.

[174] A. Segal, J. Feigenbaum, and B. Ford, Open, privacy-preserving protocols for lawful surveillance, *arXiv preprint arXiv:1607.03659*, 2016.

[175] M. Seifelnasr, H. S. Galal, and A. M. Youssef, Scalable Open-Vote Network on Ethereum, in *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC*, 2020, pp. 436–450.

[176] Semantic Versioning 2.0.0, accessed on 2023-07-12.

[177] H. Shafagh, L. Burkhalter, S. Ratnasamy, and A. Hithnawi, Droplet: Decentralized Authorization and Access Control for Encrypted Data Streams, in *29th USENIX Security Symposium*, 2020, pp. 2469–2486.

[178] P. W. Shor, Algorithms for Quantum Computation: Discrete Logarithms and Factoring, in *35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 124–134.

[179] V. Shoup and R. Gennaro, Securing Threshold Cryptosystems against Chosen Ciphertext Attack, in *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques*, 1998, pp. 1–16.

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

[181] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen, Decentralized Identifiers (DIDs) v1.0, Jul. 19, 2022.

[182] F. Spoto, Enforcing Determinism of Java Smart Contracts, in *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC*, 2020, pp. 568–583.

[183] C. Stathakopoulou, S. Rüsch, M. Brandenburger, and M. Vukolic, Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs, in *40th International Symposium on Reliable Distributed Systems, (SRDS)*, 2021, pp. 34–45.

[184] D. Stebila and M. Mosca, Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project, in *International Conference on Selected Areas in Cryptography*, 2016, pp. 14–37.

[185] S. Steffen, B. Bichsel, R. Baumgartner, and M. T. Vechev, ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs, in *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 179–197.

[186] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. T. Vechev, zkay: Specifying and Enforcing Data Privacy in Smart Contracts, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1759–1776.

[187] 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*, 2001, pp. 149–160.

[188] H. Subramanian, Decentralized Blockchain-Based Electronic Marketplaces, *Commun. ACM*, vol. 61, no. 1, pp. 78–84, 2018.

[189] Swizerland E-Voting, accessed on 2023-07-17.

[190] 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 (S&P)*, 2017.

[191] 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 (S&P)*, May 2016.

[192] N. Szabo, The Idea of Smart Contracts, 1997.

[193] A. Tapscott and D. Tapscott, How Blockchain is Changing Finance, Mar. 1, 2017.

[194] F. Team, Fire Lotto blockchain lottery, 2018.

[195] The Algorand Virtual Machine (AVM) and TEAL, accessed on 2023-09-01.

[196] The Cothority Overlay Network Library - Onet, accessed on 2023-06-24.

[197] The GNU MPFR Library, accessed on 2023-07-18.

[198] The Go Programming Language Specification, accessed on 2023-09-01, 2023.

[199] C. F. Torres, R. Camino, and R. State, Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain, in *30th USENIX Security Symposium*, 2021, pp. 1343–1359.

[200] C. F. Torres, J. Schütte, and R. State, Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts, in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 664–676.

[201] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, in-toto: Providing farm-to-table guarantees for bits and bytes, in *28th USENIX Security Symposium*, 2019, pp. 1393–1410.

[202] A. H. Trechsel, V. Kucherenko, and F. Silva, Potential and Challenges of E-voting in the European Union, Tech. Rep., 2016.

[203] A. Treiber, D. Müllmann, T. Schneider, and I. S. genannt Döhmann, Data Protection Law and Multi-Party Computation: Applications to Information Exchange between Law Enforcement Agencies, in *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, 2022, pp. 69–82.

[204] P. C. Treleaven, R. G. Brown, and D. Yang, Blockchain Technology in Finance, *Computer*, vol. 50, pp. 14–17, 2017.

[205] M. O. and Vadim Lyubashevsky, IBM scientists help develop NIST's quantum-safe standards, Jul. 6, 2022.

[206] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, DoublePlay: Parallelizing Sequential Logging and Replay, in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 15–26.

[207] Vyper: Pythonic Smart Contract Language for the EVM, accessed on 2023-09-24.

[208] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds, in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 611–626.

[209] wasmerio/wasmer: The leading WebAssembly Runtime supporting WASIX, WASI and Emscripten, accessed on 2023-07-18.

[210] WebAssembly, accessed on 2023-09-01.

[211] WebAssembly Design Rationale: NaN bit-pattern nondeterminism, accessed on 2023-09-01.

[212] WebAssembly/binaryen: Optimizer and compiler/toolchain library for WebAssembly, accessed on 2023-07-18.

[213] WebAssembly/wasi-sdk: WASI-enabled WebAssembly C/C++ toolchain, accessed on 2023-07-18.

[214] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, SoK: Decentralized Finance (DeFi), *arXiv preprint arXiv:2101.08778*, 2021.

[215] Why Web3 Matters, 2021, accessed on 2023-08-30.

[216] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel, Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs, in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

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

[218] W. Wu and B. Ford, Deterministically Deterring Timing Attacks in Deterland, in *Conference on Timely Results in Operating Systems (TRIOS)*, 2015.

[219] x87note - GCC Wiki, accessed on 2023-07-12.

[220] A. L. Xiong, B. Chen, Z. Zhang, B. Bünz, B. Fisch, F. Krell, and P. Camacho, VeriZexe: Decentralized Private Computation with Universal Setup, in *32nd USENIX Security Symposium*, 2023.

[221] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, Town Crier: An Authenticated Data Feed for Smart Contracts, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 270–282.

[222] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, DECO: Liberating Web Data Using Decentralized Oracles for TLS, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 1919–1938.

[223] 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*, vol. 22, no. 1, pp. 41–53, 2004.

[224] Z. Zheng, P. Xie, X. Zhang, S. Chen, Y. Chen, X. Guo, G. Sun, G. Sun, and L. Zhou, Agatha: Smart Contract for DNN Computation, *arXiv preprint arXiv:2105.04919*, 2021.

[225] A. R. Zihao Ding and A. Joshi, Reproducibility, 2020.

[226] G. Zyskind, O. Nathan, and A. Pentland, Decentralizing Privacy: Using Blockchain to Protect Personal Data, in *IEEE Symposium on Security and Privacy Workshops (SPW)*, 2015, pp. 180–184.

[227] G. Zyskind, O. Nathan, and A. Pentland, Enigma: Decentralized Computation Platform with Guaranteed Privacy, *arXiv preprint arXiv:1506.03471*, 2015.

Enis Ceyhun Alp

Contact Information

EPFL IC IINFCOM DEDIS
BC 263 (Bâtiment BC)
Station 14
CH-1015 Lausanne

E-mail: enis.alp@epfl.ch

Education

École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland — Sep 2017 – Aug 2023
PhD, Computer and Communication Sciences
Thesis: "Towards General-Purpose Decentralized Computing with Permissionless Extensibility
Advisor: Bryan Ford

University of Wisconsin-Madison, Madison, WI, USA — May 2017
M.S., Computer Science
GPA: 3.94

Bogazici University, Istanbul, Turkey — Jan 2015
B.S., Computer Science and Engineering
Ranked 3rd out of 75 students

University of Washington, Seattle, WA, USA — Spring 2014
Exchange student

Work Experience

Google — Zurich, Switzerland
Software Engineering Intern (PhD) — Sep-Nov 2020

- Tech Infastructure - Security & Privacy Team
- Developed fuzzers for GNOME and opendesktop.org projects. Integrated fuzzers with Google's open-source fuzzing infrastructure OSS-Fuzz to provide continuous fuzzing for the libraries.
- Fuzzers have already started finding bugs in the libraries. Two upstreams have already patched their libraries to fix the reported bugs.

comScore, Inc. — Madison, WI
Data Analyst Intern — Jun-Aug 2016

- Contributed to Hadoop-based big-data projects that ingest billions of Internet measurements on a daily basis.
- Developed new capabilities for the Internet device graphing project. My contributions were deployed to comScore's daily data processing pipeline.

DAI-Labor at Technical University of Berlin — Berlin, Germany
Engineering Intern — Jun-Sep 2013

Publications

Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, Bryan Ford. *CALYPSO: Private Data Management for Decentralized Ledgers*. *47th International Conference on Very Large Data Bases (VLDB '21)*, August 2021.

Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, Bryan Ford. *Rethinking General-Purpose Decentralized Computing*. *17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, May 2019.

Keith Funkhouser, Matthew Malloy, Enis Ceyhun Alp, Phillip Poon, Paul Barford. *Device Graphing by Example*. *24th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '18)*, August 2018.

Matthew Malloy, Paul Barford, Enis Ceyhun Alp, Jonathan Koller, Adria Jewell. *Internet Device Graphs. 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*, August 2017.

Patents

Matthew Malloy, Enis Ceyhun Alp and Paul Barford. *Systems and Methods for Generating and Transmitting Content Based on Access from a Common Device*. Patent pending, 2017.

Skills

Languages (in decreasing order of familiarity): Go, Java, Python, C, C++

Big-data systems: HDFS, MapReduce, Pig, Spark

Operating Systems: Linux, xv6

Tools: GDB, Wireshark, Git, Meson, Ninja, libFuzzer

Awards and Honors

Fellowship for Doctoral Studies, *School of Computer and Communication Sciences, EPFL* — 2017

Graduate Assistantship, *Department of Computer Sciences, UW-Madison* — 2015

- Guaranteed financial support for the first 4 academic years through teaching assistantship
- Financial support includes stipend and tuition remission

Dean's High Honor List, *Bogazici University* — 2015

- Requires having a GPA of at least 3.50/4.00 in all semesters attended

Erasmus Student Mobility for Placements (SMP) Grant — 2013

Turkish Presidential Scholarship — 2009

- Awarded to the students who ranked in the top 100 in the nationwide University Entrance Exam or Foreign Language Examination in English

Teaching Assistantships

Technologies of Societal Self-Organization (CS-234) — Fall 2019–2022, EPFL

Decentralized System Engineering (CS-438) — Fall 2018, EPFL

Information Security and Privacy (COM-402) — Spring 2018, EPFL

Introduction to Computer Systems (CS 354) — Spring 2017, UW-Madison

Introduction to Computer Networks (CS 640) — Fall 2016, UW-Madison

Advanced Operating Systems (CS 736) — Spring 2016, UW-Madison

Introduction to Operating Systems (CS 537) — Fall 2015, UW-Madison

Service

- Sub-reviewer, NSDI 2019
- Sub-reviewer, FC 2018

Student Advising

- Mahdi Bakhshi (co-advised with Philipp Jovanovic): "Investigating Transport Layer Protocols for Threshold Logical Clocks", *Summer internship*, Summer 2019
- Manuel Vidigueira (co-advised with Philipp Jovanovic): "Threshold Logical Clocks", *Master's semester project*, Spring 2019