AN ANALYSIS OF SECURITY OF BLOCKCHAIN PROTOCOLS AND THE ROLE OF FORMAL METHODS

GRZEGORZ FABIAŃSKI

A thesis submitted for the degree of
Doctor of Philosophy
Supervisor: Prof. Stefan Dziembowski

Faculty of Mathematics, Informatics, and Mechanics
University of Warsaw

April 2026

Grzegorz Fabiański: *An analysis of security of blockchain protocols and the role of formal methods* © April 2026

ABSTRACT

Cryptographic protocols are complex. This is particularly evident in blockchain systems, where the requirements of security, efficiency, and decentralization are in tension with one another. For this reason, an in-depth analysis of whether protocols actually satisfy their claimed properties is crucial.

In this dissertation, we focus on a detailed analysis of existing protocols. We obtain both negative results – in the form of identifying fundamental limitations of certain classes of protocols – and positive results – in the form of correctness proofs obtained using formal methods. In particular, we analyze in detail protocols from the pegged-ledgers family (such as Plasma), the Lightning Network protocol, and circuit garbling.

For pegged-ledgers, we establish a lower bound showing that protocols in this family must allow situations that require high communication. Our result explains why existing protocols fall into two families – Plasma Cash (operating on coins) and Plasma MVP (operating on arbitrarily divisible amounts).

We then turn to the analysis of the correctness of the Lightning Network and circuit garbling protocols. To this end, we apply formal analysis – we present correctness proofs that are verified not by a human, but by a computer. In each of the considered cases, an important challenge is to make the formal proof concise and understandable. To address this, we develop specific tools.

In the work on the Lightning Network, we propose a method for analyzing properties of given funds, which operates even when the funds under consideration are managed by an off-chain system. We also present a new, abstract specification of the channel used in the Lightning Network.

For the analysis of circuit garbling, we use a method known as symbolic cryptography. This is an approach to proving the correctness of cryptographic protocols based on symbolic expressions. We show that this method effectively helps in constructing a formal correctness proof for circuit garbling. In particular, our result includes a formalization of the symbolic approach itself.

ABSTRAKT

Protokoły kryptograficzne są złożone. Szczególnie wyraźnie widać to w systemach blockchain, gdzie wymagania bezpieczeństwa, efektywności i decentralizacji pozostają ze sobą w napięciu. Dlatego kluczowa jest pogłębiona analiza, czy protokoły rzeczywiście spełniają deklarowane własności.

W niniejszej rozprawie skupiamy się na pogłębionej analizie istniejących protokołów. Otrzymujemy zarówno wyniki negatywne –

w postaci identyfikacji fundamentalnych ograniczeń pewnych klas protokołów – jak i pozytywne – w postaci dowodów poprawności uzyskanych przy użyciu metod formalnych. W szczególności poddajemy szczegółowej analizie protokoły z rodziny pegged-ledgers (takie jak Plasma), protokół Lightning Network oraz circuit garbling.

Dla pegged-ledgers udaje nam się wykazać dolne ograniczenie, które pokazuje, że protokoły z tej rodziny muszą dopuszczać sytuacje wymagające dużej komunikacji. Nasz wynik wyjaśnia, dlaczego istniejące protokoły dzielą się na dwie rodziny – Plasma Cash (operujące na monetach) oraz Plasma MVP (operujące na dowolnie podzielnych sumach).

Następnie skupiamy się na analizie poprawności protokołów Lightning Network oraz circuit garbling. Stosujemy w tym celu analizę formalną – prezentujemy dowody poprawności, które są weryfikowane nie przez człowieka, lecz przez komputer. W każdym z omawianych przypadków istotną trudnością jest uczynienie formalnego dowodu zwięzłym i zrozumiałym. W tym celu wypracowujemy specyficzne narzędzia.

W pracy nad Lightning Network proponujemy metodę analizy własności danych funduszy, która działa nawet wtedy, gdy analizowane środki są zarządzane przez system off-chain. Przedstawiamy również nową, abstrakcyjną specyfikację kanału wykorzystywanego w Lightning Network.

W celu analizy circuit garbling wykorzystujemy metodę zwaną symbolic cryptography. Jest to sposób dowodzenia poprawności protokołów kryptograficznych, oparty na symbolicznych wyrażeniach. Pokazujemy, że metoda ta skutecznie pomaga w zbudowaniu formalnego dowodu poprawności circuit garbling. W szczególności nasz wynik zawiera formalizację samego podejścia symbolicznego.

PUBLICATIONS

This thesis consists of the following publications:

[1] Stefan Dziembowski, Grzegorz Fabianski, Sebastian Faust, and Siavash Riahi. "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma." In: *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, Virtual Conference, January 6-8, 2021*. Ed. by James R. Lee. Vol. 185. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 72:1–72:20. DOI: 10.4230/LIPICS.ITCS.2021.72. URL: https://doi.org/10.4230/LIPIcs.ITCS.2021.72.

[2] Stefan Dziembowski, Grzegorz Fabiański, Daniele Micciancio, and Rafal Stefański. "Computationally-Sound Symbolic Cryptography in Lean." In: accepted at CSF 2026, in press.

[3] Grzegorz Fabianski, Rafal Stefanski, and Orfeas Stefanos Thyfronitis Litos. "A Formally Verified Lightning Network." In: *Financial Cryptography and Data Security - 29th International Conference, FC 2025, Miyakojima, Japan, April 14-18, 2025, Revised Selected Papers, Part I*. Ed. by Christina Garman and Pedro Moreno-Sanchez. Vol. 15751. Lecture Notes in Computer Science. Springer, 2025, pp. 3–20. DOI: 10.1007/978-3-032-07024-1_1. URL: https://doi.org/10.1007/978-3-032-07024-1_1.

With thanks to everyone who has made me who I am: my family, friends, community, coworkers, closer and more distant acquaintances, as well as those whom I am unable to mention by name.

CONTENTS

1 INTRODUCTION 1
- 1.1 Blockchain systems 1
- 1.2 Scaling Blockchains Using Layer 2 Solutions 5

2 SUMMARY OF RESULTS 9
- 2.1 Limitations of Plasma 9
- 2.2 Verified Lightning Network 12
- 2.3 Circuit Garbling Security and Symbolic Soundness 14
  - 2.3.1 Symbolic Cryptography 16
  - 2.3.2 My contributions 17

3 LOWER BOUNDS FOR OFF-CHAIN PROTOCOLS: EXPLORING THE LIMITS OF PLASMA 19
- 3.1 Abstract 19
- 3.2 Introduction 20
  - 3.2.1 Introduction to Plasma 22
  - 3.2.2 Our contribution and organization of the paper 28
- 3.3 Plasma Payment Systems 30
  - 3.3.1 Protocol operation 31
  - 3.3.2 Security properties 33
- 3.4 Our main result 34
- 3.5 Proof of Thm. 1 35
  - 3.5.1 Isolation scenario 35
  - 3.5.2 Proof overview 36

4 A FORMALLY VERIFIED LIGHTNING NETWORK 41
- 4.1 Abstract 41
- 4.2 Introduction 41
  - 4.2.1 Overview of LN 42
  - 4.2.2 Why3 43
  - 4.2.3 Preliminaries on Cryptography 43
  - 4.2.4 Our Contributions 44
  - 4.2.5 Protocol Flow in Our Implementation 44
  - 4.2.6 Security Goals 45
  - 4.2.7 Adversarial Model 45
  - 4.2.8 Architecture Overview 46
  - 4.2.9 Related Work 47
- 4.3 Security Model 48
- 4.4 Modeling Bitcoin 51
- 4.5 Proof Strategy 52
  - 4.5.1 Digital Signatures Ideal Functionality 53
  - 4.5.2 Encapsulation of Implementation-aware Properties 53
  - 4.5.3 `goodSimpleParty` 54
  - 4.5.4 Funds Ownership 54
  - 4.5.5 Evaluator Solvency Overview 56
- 4.6 Conclusion & Future Work 56

5 COMPUTATIONALLY SOUND SYMBOLIC CRYPTOGRAPHY IN LEAN 59
5.1 Abstract 59
5.2 Introduction 59
5.2.1 Our Contribution 61
5.2.2 Related Work 62
5.2.3 Limitations 64
5.2.4 Lean Code 65
5.3 Symbolic Cryptography 65
5.3.1 Cryptographic Expressions 66
5.3.2 Computational Semantics 67
5.3.3 Symbolic Indistinguishability 68
5.4 Extended Example — Circuit Garbling 76
5.4.1 Multi-Party Computation and Circuit Garbling 76
5.4.2 Security of Circuit Garbling 78
5.4.3 Proof Technique for Working with Fixpoints 79
5.5 Computational Indistinguishability 80
5.5.1 Indistinguishability-Based Security of Encryption Schemes 81
5.5.2 Computational Soundness Theorem 85
5.6 Conclusion and Future Work 89

BIBLIOGRAPHY 91

1

INTRODUCTION

1.1 BLOCKCHAIN SYSTEMS

Blockchains are distributed systems that enable digital financial operations without relying on a single trusted provider. The cornerstone capability offered by nearly all blockchain systems is payment functionality, allowing users to transfer money to one another securely. Historically, the first blockchain was Bitcoin [79], which still enjoys the highest market capitalization today. Another prominent example is Ethereum [99] – the first system to focus on enabling users to transfer funds to custom programs, known as contracts, which execute in a trustless manner.

Blockchains can be studied from various perspectives, such as economics or social phenomena. In this work, we adopt a computer science perspective: we analyze the technical foundations of these systems, the functionalities they provide, and their performance requirements.

BLOCKCHAINS OF STATE MACHINE REPLCIATION SYSTEMS. What constitutes a blockchain system? One way to describe a blockchain system is by specifying the format of transactions and explaining how their execution affects the system's state. However, every blockchain system also provides a synchronization mechanism to ensure that all participants have a common view of the system. This aspect can be separated into two main components: the *execution layer*, which focuses on processing transactions and updating the state, and the *consensus layer*, which ensures consistency among participants' views.

From a distributed computing perspective, blockchains enable all participants to obtain a shared, consistent view of the financial system's state. Ideally, all participants would maintain exactly the same view of the system at all times. In practice, however, such a strong consistency requirement conflicts with efficiency. For example, the proof-of-work protocol used in the Bitcoin network does not guarantee a perfectly synchronized state for all users. As a result, a relaxed consistency model is adopted: some users may see slightly outdated states compared to others.

The functionality that models such a system is called *atomic broadcast*. During its operation, each user submits entries to be included in a shared log. Each user runs a program, called a replica, which participates in the underlying protocol and outputs the log (that is, the replica produces a sequence of entries during its execution). These logs are then read by users. Let us denote the log read by user A as log_A. A key requirement of atomic broadcast is *consistency*: for any two honest users, their logs must be consistent in the sense that either log_A is a prefix of log_B, or vice versa. The second crucial property, called

liveness, requires that any entry submitted by a user will eventually appear in the log of every honest user. The *consensus layer* consists of a protocol that implements the atomic broadcast functionality. Different blockchain systems not only use different protocols for consensus, but also rely on different underlying assumptions. For example, Bitcoin uses a proof-of-work protocol [33], while Ethereum employs a custom proof-of-stake protocol [89].

It is important to note that while the notion of byzantine (atomic) broadcast captures a key technical component present in all blockchain solutions, it does not directly describe the user-facing functionality these systems provide. Users are primarily interested in making payments or executing other financial operations, not just sequencing abstract log entries. This raises an important question: how does log replication give rise to financial applications, such as payment functionality? In blockchain terminology, the log replication system allows all participants to agree on a shared ordering of executed transactions. Financial capabilities are established by defining the system's state (such as the balance of each participant) and specifying a deterministic procedure for how each transaction modifies this state. Any party can then process the transactions from the log in sequence, starting from a predetermined initial state. Most blockchains can be described in this way, though they differ in the details of their state spaces and transaction processing. For example, the state of the Bitcoin system is represented by a list of unspent transaction outputs (UTXOs), which function like single-use accounts. In contrast, the state of the Ethereum system consists of a set of (multiple-use) accounts.

The process described so far can be viewed in the following way: the replicated log is interpreted as a sequence of transactions. These transactions are executed sequentially, with each one modifying the state of the system. This process is formalized by the notion of a state machine. A state machine consists of a state space (describing the possible states of the system), a set of legal transactions, an initial state, and a transition function. The transition function takes the current system state and a transaction, and produces a new system state. Log replication protocols can therefore be seen as state machine replication protocols. This is achieved by having each user interpret the log as a sequence of transactions to execute using the state machine. The specific state machine used in a blockchain system defines its *execution layer*.

SMART CONTRACTS. Up to this point, we have considered only the most basic and fundamental capability of blockchains: transferring coins between users. Many (if not most) deployed blockchain systems offer additional features. In particular, they often allow users to extend the system's capabilities by submitting custom code that is executed on-chain. Examples of such extensions include the creation of new tokens, cross-chain bridges (which enable transfers between different blockchains), options trading, prediction markets, and others.

The core technology enabling this application is the *smart contract*. The concept was introduced by Szabo in the late 1990s [91], where

he envisioned binding contracts with automatic enforcement. On blockchain, smart contrat are autonomous object, that could hold coins. Each smart-contract could be viewed as state machine – it is defined by state space, initial state and method to process transactions send by users. An user can send transaction to smart contract, providing transferred amount and some additional data. Smart contracts processes such transactions by reading transaction data and amount, modifying its own state in process and calculating how much money transfer back to user.

Let us consider a simple example that demonstrates how to introduce a new type of coin, called Lit, into a blockchain system. A smart contract can keep track of a list of user balances for this new coin. Each user can send a transaction to the smart contract with additional data of the form "send 7 Lits to B," while transferring zero native coins to the contract. The smart contract processes such transactions by updating its list of balances accordingly. In fact, this is the same kind of state machine we described earlier when discussing how blockchain systems themselves are constructed. Smart contracts can be thought of as "worlds within worlds," allowing the contract deployer to define and enforce their own set of rules.

BLOCKCHAINS AS TRUSTED PARTY. In cryptography, many classical problems arise from the lack of trust among participants. Typically, a small group of parties wants to achieve a common goal by executing a protocol, while recognizing that some parties may behave dishonestly and violate the rules. However, many of these problems become trivial if at least one participant is fully trusted by all others. In such cases, all parties can simply send their data (including secrets) to the trusted party, who computes the correct result and distributes it to the relevant participants. On the other hand, many real-world scenarios lack parties who are completely trusted by all participants.

Are protocols relying on such a trusted party still useful? Yes: they can be implemented using a smart contract on a blockchain system that supports this functionality. Therefore, a smart contract can be seen as a special, though limited, form of trusted party. In traditional financial applications, trust is usually placed in real-world intermediaries like banks. Blockchain systems, by ensuring the correct execution of the state machine that processes transfers, remove the need for such trusted intermediaries.

However, blockchains do not trivialize all cryptographic problems. This is because they have a crucial limitation: the state of the blockchain is public to all participants. As a result, blockchain systems cannot hold any secrets. In summary, a blockchain can be viewed as a trusted party, but one whose state is fully transparent and accessible to everyone. An additional issue is the high cost of using blockchains, which we discuss in more detail in the next section.

PERFORMANCE OF BLOCKCHAINS. In practice, blockchain systems face several performance limitations. The most basic metric is transaction throughput, which is defined as the number of simple transfer

transactions a blockchain can process per second. For example, Bitcoin can handle approximately 7 transactions per second (tx/s) [37]. These rates are much lower than those of global payment networks; for instance, Visa can process around 2000 tx/s. Another important metric for transaction processing capacity is the fee that users must pay to have their transactions included in the blockchain. At the time of writing, a transaction on the Bitcoin network costs around $0.4USD$[1]. These two measures are closely related: the limited capacity for processing transactions drives up the price users must pay for inclusion.

Another important metric is the time between initiating a money transfer and the recipient actually receiving the funds. This metric is crucial for any party that wants to act based on the confirmation of payment—for example, a seller who ships goods only after receiving payment. This concept is formalized by the notion of transaction finality: a transaction is considered final when all honest parties eventually agree that it has occurred and cannot be reversed.

To report bounds on finality time in actual systems, we need to specify a security level. For example, we might ask: what is the upper bound on Bitcoin's transaction finality time if we require that the probability of a finalized transaction being invalidated is less than 10^{-6}? (This corresponds to the probability required by the definition of a secure log-replicated system.) Since we require this guarantee under any adversarial conditions, the finality time for a given security level cannot be measured experimentally; instead, it must be calculated theoretically, as the security level of any particular finalization rule cannot be empirically observed. For example, [33] estimates Bitcoin's latency to be 6 hours for error probabilty of 10^{-6}. In comparison, centralized systems such as payment card networks (e.g., Visa) typically provide finality within several seconds to a few dozen seconds.

Finally, note that both performance indicators discussed so far – transaction throughput and finality delay – are determined by the capabilities of the consensus layer of a given blockchain. Transaction throughput corresponds to the throughput of the generated log, that is, how many bits can be added to the blockchain per second. Finality delay refers to the time between when a transaction is generated by a user and when it is actually included in the log—specifically, the delay required to satisfy the liveness property of a log replication system.

On the other hand, the execution layer has its own limitations. One important constraint is the size of the state of the replicated machine – in practice, this means the total size needed to describe the state of all accounts, including balances and the states of smart contracts. Clients[2] of a blockchain system need to be able to store this state in their memory. Therefore, it is essential to keep the size of the state reasonably small, as an excessively large state would prevent many potential clients from participating. Another important metric is the running time of the transition function that the state machine uses to execute each transaction. In most systems, the cost of a transaction is

1 checked on https://bitinfocharts.com/comparison/bitcoin-transactionfees.html

2 This includes both full clients and validators.

the sum of three parts: its size (which relates to throughput limits), its execution time (determined by the runtime of the state machine's transition function), and its storage impact (how much it increases the total size of the system state).

Fortunately, costs related to storage and execution time can be greatly reduced using technologies based on succinct proofs. Such proofs allow one to verify the effect a transaction has on the (hash of) the system state without re-executing the transaction or manually updating the full state. Importantly, the size of these proofs and the time needed to verify them are both smaller than the time required for direct execution and state updates. These techniques form the foundation of a technology called *rollups* [92].

However, rollups do not improve transaction throughput, since all transaction data must still be posted directly on-chain. As a result, the maximum transaction throughput remains limited by the size of the transaction data. Likewise, long finality times are not affected by rollups. These issues are addressed by so-called Layer-2 solutions, which will be discussed in the next section.

1.2 SCALING BLOCKCHAINS USING LAYER 2 SOLUTIONS

Layer 2 solutions are a family of protocols designed to scale blockchain systems. They allow transactions to be made off-chain, meaning that parties can transact without interacting directly with the blockchain for each individual transfer. Multiple types of Layer 2 solutions have been proposed, each based on different assumptions and offering various trade-offs. While the exact functionality depends on the specific protocol, at a minimum, Layer 2 solutions typically provide payment functionality—that is, they allow participants to transfer coins between themselves. Although this basic functionality is also provided directly by the blockchain, using Layer 2 solutions reduces the cost of transactions and, in some systems, improves finality times. A comprehensive overview of Layer 2 solutions can be found in [54].

A Layer 2 solution can be understood as a protocol involving a set of participants and a smart contract operating on the blockchain[3]. At the start, each participant deposits a certain amount of money into the smart contract. After this initial step, the Layer 2 protocol begins operating. Participants can then make transfers to each other within the Layer 2 system—these are called *off-chain* transactions. Importantly, the total communication with the blockchain during this phase is much less than if each transaction were performed directly on-chain.

Participants can exit the Layer 2 system at any time, withdrawing their current balance to their on-chain account (an account registered directly on the underlying blockchain). In many Layer 2 designs, off-chain transactions are made without any interaction with the blockchain, and the amount of required on-chain communication does not depend on the number of off-chain transactions performed.

3 Some solutions, such as the Lightning Network, are based on a weaker form of programmability than full smart contracts.

EXAMPLE PROTOCOL. To illustrate how Layer 2 solutions work, consider a simple example. This scenario demonstrates the basic operation of a class of solutions known as payment channels.

Suppose there are three parties: Alice, Bob, and Nate. Alice and Bob each have a large number of identical rare coins that they want to transfer between themselves repeatedly. Physical transfers are impractical, so they both deposit their coins with Nate, a notary whom they both trust. When Alice or Bob wants to transfer coins, they visit Nate and sign documents changing ownership of a portion of the deposit. However, going to Nate and paying a notarial fee for every transfer quickly becomes inconvenient.

To improve this, Alice and Bob agree on the following procedure. If either Alice or Bob shows Nate a note properly signed by both parties describing the division of their joint deposit, Nate will contact the other party to request their version of the note. Nate will only accept notes signed by both Alice and Bob. If both parties provide such notes, Nate compares the dates on the two notes and distributes the coins according to the most recent note.

With this agreement in place—formalized by signing the necessary legal documents—Alice and Bob can transfer coins simply by exchanging signed notes. Whenever they want to transfer coins, they agree on a new split of the deposit, write the date on a new note, and both sign it. Each keeps a copy of the most recent note. If Nate ever contacts either of them, they present their latest note. This process ensures that even if, for example, Alice tries to use an outdated note (perhaps describing a more favorable split for herself), the latest note from Bob will determine the actual payout—thus protecting both parties from the other's misbehavior.

How does this story relate to blockchains? In the blockchain context, the role of Nate is played by a smart contract. Alice and Bob deposit some coins into a smart contract to initiate a channel. The "notes" are now digital messages signed by both Alice and Bob using digital signatures. If a dispute occurs or one party wants to settle, these signed messages are sent to the smart contract, which enforces the same logic as Nate. The only difference is that while Nate can directly contact a party (for example, by phone), a smart contract must rely on parties to monitor its state and respond within a specified time.

The key property of this protocol is that off-chain transfers between Alice and Bob are made entirely by exchanging messages between themselves. No transaction fees are required for these transfers, since they do not involve any on-chain operations. Only opening, closing, or settling the channel requires interaction with the blockchain.

CHARACTERISTICS OF LAYER 2 SOLUTIONS. The security of funds deposited in a Layer 2 solution is of paramount importance. Most Layer 2 solutions aim for unconditional security: a user's funds remain safe as long as the user follows the protocol correctly. However, achieving this security typically requires users to actively monitor the blockchain and respond in a timely manner to any potentially malicious behavior by other participants. Security here means that if

a user responds as needed, they will be able to withdraw all of their funds, regardless of others' actions.

Another fundamental property is liveness: each user should be able to transfer their funds as intended. Many systems offer liveness only in the optimistic case—for example, when at least some distinguished parties behave honestly. However, any reasonable solution should guarantee liveness at least when all participants are honest.

A crucial requirement for any blockchain scaling solution is cost reduction. Generally, Layer 2 solutions break the direct link between the number of transactions and blockchain costs. This means that many off-chain transfers can be made for a fixed, limited cost paid to the blockchain, keeping the cost per off-chain transaction very low. This property is key to achieving real scalability.

Another important metric is finality delay (latency). Some Layer 2 solutions enable transactions to be finalized more quickly than the underlying blockchain. For example, in our earlier toy example, even if contacting Nate (the notary) takes some time, Alice and Bob can consider a payment finalized as soon as both have signed an updated note. By exchanging messages directly, Alice and Bob can ensure that the new state will become binding as soon as either decides to exit the system. While actual resolution through Nate (or the smart contract) might involve some delay, the result is already determined, so the participants can immediately consider the transfer final.

We say that a Layer 2 solution provides *immediate finality* if, in the optimistic case where all participants behave honestly, the finality of a transfer can be achieved without any further interaction with the blockchain. In other words, the only delay is due to network communication between participants. Although there may still be a slight delay, direct communication between users is typically so fast as to be practically instantaneous for humans.

NOTABLE LAYER 2 PROTOCOLS In this section, we briefly describe the main classes of Layer 2 solutions.

Channels netorks [86] are a family of protocols that enable immediate finality. Single channel operate among a small group of participants in a manner similar to the toy example described earlier. Crucially, any change to the state requires valid signatures from all participants. This approach, however, comes with a significant limitation: any participant can halt the channel's operation by simply refusing to sign new messages, effectively stopping all transfers. As a result, liveness for payment channels is only guaranteed when all participants are honest. One unique advantage of channels is their compatibility with blockchains that do not support full smart-contract functionality. In fact, channel-based solutions are dominant on Bitcoin, which provides only limited scripting support.

Channel networks are built by connecting many channels (usually each with only two participants) to form a network. This design allows payments to be routed across the network via intermediaries. Channel networks benefit from immediate finality and minimal interaction with the blockchain. However, they also come with additional

assumptions and complexities not present in other solutions. Firstly, channel networks can function only if some users are willing to lock more funds in the system than they plan to transfer, acting as liquidity providers. This fund-locking imposes additional costs, as liquidity providers usually require compensation. Furthermore, a payment succeeds only if a suitable path of channels exists between sender and receiver, requiring intermediaries to be online and responsive. If any intermediary is dishonest or unavailable, the transfer fails or may lock funds for some time. Thus, channel networks work best when the vast majority of participants are honest. In addition, defining channel network functionality is complicated, since it often requires the user to be aware of intermediaries and the network structure. To summarize, while channel networks provide better resilience against individual misbehavior compared to single channels, they rely on fund-locking and intermediaries, and expose more complexity to users.

Plasmas (also called *commit chains* or Nonust) [61, 76, 85, 94] rely on a distinguished party called the *operator*. Plasma design guarantee fund security even if the operator is dishonest, but transaction processing speed and liveness depend on the operator's honesty. Notably, liveness in Plasmas does not depend on the honesty of regular users, which makes Plasmas suitable for mass deployments with large numbers of anonymous users. In Plasmas, users never interact directly with each other, but only with the operator and the blockchain. This keeps communication costs linear in the number of users. However, Plasmas face three main limitations. First, finality is relatively slow—often even slower than the base blockchain—since off-chain transfers are only finalized once a so-called *checkpoint* (a short message from the operator summarizing transactions) is posted and confirmed on-chain. Second, while these checkpoints allow scaling (because their size does not depend on the number of included transactions), they must be posted regularly, incurring operational costs that are linear with time. This makes small-scale Plasmas impractical, as a critical mass of users is needed to amortize the cost of checkpoints. Finally, Plasmas designs like Plasma suffer from the *mass exit problem*: under certain conditions, many users may need to simultaneously submit exit transactions on-chain, which can cause congestion.

2

SUMMARY OF RESULTS

In this section, we describe the contributions of each paper included in this thesis.

2.1 LIMITATIONS OF PLASMA

As previously mentioned, Plasmas solutions often suffer from the mass-exit problem. A mass exit occurs when a significant number of participants (often a fixed percentage of all users) need to send messages to the underlying blockchain within a limited period of time. Such events are problematic because they generate a spike in communication with the blockchain, which undermines the goals of efficient scaling.

An additional complication is that mass-exit events can lead to congestion on the blockchain. If the number of messages exceeds the processing capacity of the blockchain, some messages will be delayed. This is particularly dangerous because users may lose funds in the pegged-ledger if their responses do not reach the blockchain in time. One way to partially address this issue is by using dynamic waiting periods. For example, if the smart contract managing the pegged-ledger enters a state that requires party A to respond, instead of using a fixed timeout to determine whether A has failed to respond, the contract can adapt the waiting period in response to observed congestion. In this way, even if messages from A are delayed due to congestion, they do not immediately time out.

Several protocols have been proposed for pegged-ledgers, frequently referred to as "Plasma flavors." Below, we describe the most prominent among them.

MINIMAL VIABLE PLASMA (PLASMA MVP) In this section, we describe how Plasma operates. We present its fundamental model of operation using the Minimal Viable Plasma protocol (Plasma MVP), proposed in [27], as a representative example. In Plasma MVP, the operator periodically submits hashes of the system state—called checkpoints—to a smart contract on the blockchain. In addition, the operator communicates directly with each user, providing them with access to the full system state. This process is informally referred to as the operator "explaining" the checkpoint, allowing users to verify the correctness of the checkpoint data.

If a user detects that the operator has published an incorrect checkpoint, they must alert the blockchain, effectively withdrawing all their funds based on the balance state from the previous checkpoint. This process is known as initiating the exit procedure.

A particularly problematic scenario occurs when the operator stops communicating with most users and submits a checkpoint corre-

sponding to an invalid state—for example, one where all funds have been assigned to a user secretly controlled by the operator. If this invalid state were accepted, users would lose their funds. However, the protocol prevents this by allowing affected users to initiate the exit procedure, thereby exiting the system before the incorrect state impacts their balances. Unfortunately, initiating exits requires significant communication with the blockchain and can lead to congestion. Thus, even though a dishonest operator cannot directly steal funds, they can still cause major disruption.

One might consider mitigating dishonest operator behavior by requiring the operator to provide collateral—funds that could be confiscated in case of misconduct. However, this approach is fundamentally flawed. Consider the following, especially concerning, type of attack: the operator follows the protocol correctly, except that they stop providing users with access to the full system state. In this scenario, users must initiate the exit procedure, as they can no longer verify the correctness of published checkpoints. However, users are unable to prove to the blockchain that the operator acted dishonestly. This problem is known as non-unique attribution: the fault (failing to send information) cannot be uniquely tied to the operator, as from an external perspective, it is impossible to distinguish between a message not being sent and one not being received. Due to this non-attributable fault, the protocol cannot justifiably confiscate the operator's collateral, making collateral requirements ineffective as a deterrent.

PLASMA CASH An alternative approach to building pegged-ledgers is demonstrated by the Plasma Cash protocol [28]. In Plasma Cash, users transfer funds using coins—each coin has a fixed denomination and can be individually transferred between users. A coin holder can withdraw a coin, which destroys the coin and grants the user the corresponding funds on-chain. Crucially, the exit process is handled on a per-coin basis. This means that if a user wants to withdraw 100 coins, they must identify and process each of the 100 coins separately, generating a significant amount of communication with the blockchain. Protocols that operate on the coin level are called *non-fungible*.

This coin-based approach imposes certain limitations on the payment system. Coin denominations cannot be too small, or else the fees associated with withdrawing will be too high relative to the value of the coin. Since each payment must be a whole number multiple of coins, users cannot transfer arbitrary amounts. Additionally, fees in Plasma Cash are proportional to the number of coins involved in a transaction (i.e., the system's payment granularity), making it difficult to compare with direct blockchain usage, where fees are usually paid per transaction. In practice, Plasma Cash requires coins of non-negligible value (coarse granularity) to achieve significant cost savings.

Not all coins in Plasma Cash need to have the same denomination; the system is similar to how people use physical cash with both coins and banknotes of various denominations. Typically, people keep only a small number of coins and will exchange smaller denominations

for higher ones if needed. There was hope that a similar mechanism could be implemented in Plasma Cash to automatically minimize the number of coins any user holds. However, no such scheme has been found that can reduce communication in the worst-case scenarios.

How does Plasma Cash compare to MVP Plasma? Unlike Plasma MVP, Plasma Cash does not require a forced exit mechanism, thereby avoiding the mass-exit problem. However, large withdrawals in Plasma Cash still generate substantial communication with the blockchain, which ultimately limits scalability.

FUNDAMENTAL LIMITATION The existence of multiple proposed protocols for implementing pegged-ledgers—each with its own set of drawbacks—raises a natural question: is there a pegged-ledger protocol that avoids these issues? More specifically, is it possible to design a fungible pegged-ledger protocol without the mass-exit problem?

In the paper "Lower Bounds for Off-Chain Protocols: Explor- ing the Limits of Plasma", we have resolved this question in the negative. In our analysis, we identified a fundamental limitation shared by all pegged-ledger protocols. More precisely, we proved that every pegged-ledger protocol must fall into one of two categories. In protocols of the first type, there exists a scenario where the communication required during the exit procedure is proportional to both the number of users and the amount of value withdrawn. These protocols therefore share the weakness of Plasma Cash—namely, the requirement for large amounts of communication during exits. In protocols of the second type, there exists an execution in which a constant fraction of users must communicate with the blockchain, even if none of them originally intended to exit the system. This is the core weakness of Minimal Viable Plasma—namely, the possibility of a forced, mass communication event (mass exit).

Furthermore, our separation holds even if the operator is required to post collateral. This follows from the fact that in protocols of the second type, the mass communication event can be triggered without any uniquely attributable operator fault.

Our result explains the observed practical separation of Plasma protocols into two groups and shows that no protocol can avoid the characteristic weaknesses of both at the same time. Although our result is negative, understanding such limitations is an important aspect of protocol design. In particular, in the context of consensus protocols, investigation into lower and upper bounds has often driven the development of new constructions. To the best of our knowledge, ours is the first lower bound specifically addressing the construction of off-chain protocols. Results of this nature help guide protocol designers by clearly identifying which goals are unattainable. They also highlight the importance of explicit modeling assumptions: any search for protocols with low communication complexity must rely on either enhanced capabilities for participants or new restrictions on adversaries.

2.2 VERIFIED LIGHTNING NETWORK

The security of protocols used in financial applications is of utmost importance, as any attack can have devastating financial consequences. In the financial sector, adversaries are highly motivated, since successful attacks often result in direct monetary gain by stealing funds. As a result, financial systems—including blockchain-based systems—face constant threats from determined attackers.

Most off-chain protocols can be proven secure within relevant security models. These security definitions are cryptographic in nature, typically using simulation-based or game-based techniques to establish the security of a protocol. Unfortunately, such security proofs are often highly technical and complex. Verifying their correctness requires expert knowledge and considerable effort. The challenge is further compounded by the fact that pen-and-paper proofs frequently omit details considered trivial, under the assumption that an expert reader can fill in the gaps. While this is sometimes justified, in other cases, such omissions can conceal subtle errors.

Formal verification is regarded as the highest standard of mathematical proof. Its key idea is to create proofs that are *machine-checkable*—that is, a computer program can rigorously validate their correctness. Unlike informal proofs, machine-checked proofs must be fully explicit and leave no gaps. In effect, the verifier acts as a highly meticulous, skeptical reader, checking each logical step in detail.

Given the need for high security assurance in Layer 2 solutions, a natural question arises: is it feasible to use formal methods to back the security of Layer 2 protocols? In our work, we address this question by focusing on the Lightning Network—a channel-based solution for off-chain payments on the Bitcoin Network. In the project "Formally Verified Lightning Network", we provide the first formally verified implementation of the core component of Lightning: the two-party direct payment channel. Our verification effort focused on fund security—arguably the most important property for any user—while liveness was left for future work.

MODELLING Formalizing the security proof of a complex multi-party protocol is a substantial undertaking. The Lightning Network protocol executed by clients involves dozens of different states that a party can occupy, as well as many possible transitions between them. Furthermore, the protocol requires each party to keep track of an ever-growing set of revocations (invalidated prior states). This makes standard model checking techniques less effective, as they usually cannot handle systems with unbounded memory requirements. In addition, the protocol relies on cryptographic signatures and their unforgeability, which is typically defined using advanced cryptographic concepts such as polynomial-time adversaries. Reasoning about probabilistic processes also adds another layer of difficulty. In order to make formal verification feasible, it is crucial to address these challenges effectively.

To this end, we developed the following modeling strategy. We represent signatures using an ideal functionality: every use of a signature is modeled as an interaction with a trusted party that faithfully implements the desired cryptographic functionality. This abstraction is justified by the Universal Composability (UC) framework [32], which shows that security proofs using an ideal signature functionality are equivalent to proofs using any concrete secure signature scheme (against polynomial-time adversaries). Importantly, the ideal signature functionality is deterministic, which eliminates a major source of randomness. For our work, we chose to define and prove security in a model based on the ideal signature functionality.

Building on this, we further strengthen the security statement: rather than proving security only against polynomial-time adversaries, we prove it against arbitrary sequences of adversarial actions—even those that may be uncomputable. While this does not increase the practical strength of the final result (since the abstraction is only sound for polynomial-time adversaries), it greatly simplifies the modeling. Specifically, we do not have to define or reason about polynomial-time computability, and all randomness is removed, making our protocol model fully deterministic. This deterministic setting significantly simplifies the formal reasoning required for verification.

The approach presented above enable us to use Why3 [23] – a framework for formal verification of high-level code. Usually, verification of cryptographic protocols requires specialized tools, able to efficiently reason about probabilities programs and capable of expressing hardness assumptions. Our approach obviates such needs, enabling as to use much simpler logic and more efficient tools. In particular, proof engineering in Why3 benefits from the help of SMT solvers [15, 35] – a powerful tools that automates proves for simple statements. That enables efficient handling of thousands verification conditions.

ADVERSARIAL MODEL In our work, we focus on *funds security* – a property that ensures any honest user of the network can always withdraw their rightful balance on-chain. Specifically, this property guarantees that when a user closes a channel, the amount withdrawn is at least equal to the initial deposit, increased by the total value of all announced incoming payments, and decreased by the value of all outgoing payments authorized by the user.

We assume the protocol is executed in the presence of an adversary. The adversary controls the behavior of the counterparty and can send arbitrary messages on its behalf. They also have (limited) control over message delivery; for example, they can delay Bitcoin transactions sent by the honest party, but only up to a fixed maximum delay. Additionally, the adversary determines the actions taken by the honest party: they can choose when the channel is opened and closed, as well as the sequence of outgoing transfers initiated. In a real deployment, these actions would be decided by the user. However, since the adversary might influence user behavior, we follow the standard approach in cryptography and prove security even in the extreme case where the adversary has complete control over the client's actions.

USED TECHNIQUES During our proof effort, we developed several key concepts that helped manage the complexity of the Lightning Network. The first concept is related to modularity. The Lightning Network protocol involves multiple interconnected parties. For example, while the most prominent part of an honest client concerns its interaction with a counterparty, the client must also monitor the blockchain and respond accordingly. Each type of interaction requires its own set of complex invariants to track correctness. Additionally, the signature functionality should be treated as an independent interactive entity, since it is used by both clients and the blockchain, further increasing the number of potential interactions and invariants.

To untangle this web of dependencies, we identified the most complex component and isolated it from the rest of the system. In our model, the client's behavior is the most intricate part, described by a dozen states and even more transitions. We developed a specification (called 'party iterface') that abstracts away implementation details of the client's behavior. More precisely, our specification defines the values a client must expose and a set of invariants that describe legal client behaviors. This allowed us to split the proof into two main parts. First, we proved that our model of client behavior meets all the obligations specified in the abstraction. Second, we showed that any implementation adhering to this specification is secure—that is, no adversary can violate the security of user funds. An additional benefit of this approach is that our specification can be used to justify the security of other Lightning Network implementations. This is significant, as our specification is much lower-level and more manageable to verify than the overall fund security property itself.

Another insight from the project was that modeling blockchain payments (even without any off-chain protocol) is itself a nontrivial task. The very intuitive concept of "ownership" of money turns out to be subtle. Consider the following scenario: party A signs and publishes a transaction sending some coins to B. Immediately after the transaction is signed, one would assume that A no longer owns the coins. However, this ownership change occurs independently of any observable action on the blockchain. At the same time, B does not immediately gain ownership, since A might have signed a double-spending transaction that redirects the coins back to themselves (only one of the transactions will be confirmed). Although people can usually determine which party truly "owns" the funds (i.e., has complete control over them), the formal definition of this notion is intricate. In our work, we proposed a game-based approach to defining ownership, capturing these nuanced scenarios in a precise way.

2.3 CIRCUIT GARBLING SECURITY AND SYMBOLIC SOUNDNESS

Blockchain systems provide mechanisms for finality of public computations. However, in many applications the data being processed is private or even secret. A classical example is the following. Alice holds input A and Bob holds input B. They want to jointly compute

some function f on both of their inputs. At the same time, they do not want to reveal their inputs to each other. Ideally, the only information reveale aboat A and B would be the value of $f(A, B)$. This problem is known as *multi-party computation* (MPC) and is a fundamental tool in cryptography [49, 102] (see [43]). Typically, one assumes that function f, that Alice and Bob want to jointly execute, is represented by a circuit C, where some of the input wires belong to Alice and the others to Bob.

A classical MPC protocol is *circuit garbling*. In this approach, Alice computes a garbled version of the circuit. Intuitively, this consists of repeatedly encrypting the information about the gates in the circuit. She also computes a garbled encoding of each possible input value (two labels for each wire: one corresponding to truth and one to falsehood). The construction is arranged so that, after learning the garbled labels for Alice's and his own input wires, Bob can evaluate the garbled circuit and obtain the output. It is crucial that Bob can perform this evaluation using only the provided input labels; otherwise, he could cheat by evaluating the circuit multiple times and thereby gain additional information about Alice's input. To distribute the labels for Bob's wires from Alice (who generated them) to Bob (who uses them), a functionality called *oblivious transfer* is invoked [44, 87]. For each of Bob's wires, it guarantees that Bob learns exactly one of the two labels, while Alice does not learn which label Bob requested. The garbled-circuit technique has applications beyond secure computation; for example, it has been used to construct zero-knowledge protocols. Protocols in the garbled-circuit family have been extensively optimized, using techniques such as *free XOR* [63] and *row reduction* [84].

Given the fundamental role that circuit garbling plays in cryptography, it is natural to apply formal methods to justify its correctness. This is particularly important because even the security of the basic protocol is nontrivial: while the main ideas are very clear, carrying out a rigorous proof on paper requires great attention to detail. The difficulty of this process is illustrated, for example, by the fact that the originally proposed correctness proof for 3-XOR not only turned out to be flawed, but also relied on assumptions that were later shown to be too weak [34].

Modern cryptography formulates its statements using a formalism similar to that found in other branches of mathematics. In particular, it is possible to use general-purpose proof assistants to formalize theorems in cryptography. At the same time, the specific nature of cryptographic proofs motivates the development and use of specialized tools, such as EasyCrypt [16, 17] or CryptoVerif [21]. For example, many cryptographic results are based on indistinguishability of distributions and are proved via reductions; the above tools facilitate carrying out such proofs. On the other hand, many cryptographic arguments rely on nontrivial mathematical facts. Using general-purpose proof assistants makes it possible to carry out a complete proof within a single system. Moreover, these assistants are often much more developed, due to their significantly larger user base. This raises the question: is it feasible to use such general tools for cryptographic

proofs? Must this inevitably lead to substantial complications, due to the lack of specialized support?

In the paper "Computationally-Sound Symbolic Cryptography in Lean" we present a formal proof of correctness for circuit garbling in Lean, a popular general-purpose proof assistant. The security of this protocol had previously been formalized using EasyCrypt. Our work gives a proof of comparable length, with a transparent structure, thereby demonstrating that it is possible to carry out such a proof efficiently in Lean. We achieve this by using *symbolic cryptography* [67, 73]. We describe this approach in detail below.

2.3.1 *Symbolic Cryptography*

In cryptography, indistinguishability is a particularly important class of properties. Informally, it states that two procedures produce outputs that cannot be told apart. The notion of indistinguishability is used to define encryption security, zero-knowledge, and many other concepts. The symbolic approach aims to develop a method that allows one to decide, in a simple way, whether a given cryptographic property holds. Naturally, any such attempt must restrict the class of properties under consideration, since without such restrictions the relevant decision problems are often undecidable.

The key ingredient of the symbolic approach is the definition of *symbolic indistinguishability* [67, 73] of expressions. This is a syntactic procedure, specified as a sequence of operations on strings. The appeal of the symbolic method lies in two facts. First, the procedure itself is easy to carry out and reduces to a sequence of algorithmic operations on symbolic expressions. Second, it is correct: if two expressions are symbolically indistinguishable, then the actual programs implementing them, using any secure encryption scheme, produce computationally indistinguishable outputs. We refer to this property as the *soundness theorem*.

How does the symbolic procedure for checking indistinguishability work? First, the procedures under consideration are written as symbolic expressions. Next, both expressions are reduced to a normal form. Then, an iterative procedure analyses which values cannot be learned by the adversary from a given expression; all such unreadable values are eliminated. Finally, the resulting expressions are compared with each other.

The main advantage of the symbolic approach is that such proofs do not work directly with reduction-based arguments. This is particularly important for circuit garbling, where the nested, layered structure of the garbled circuit makes reduction proofs highly technical and complicated. Thanks to the use of the symbolic approach, our correctness proof for circuit garbling naturally splits into two parts. The analysis of circuit garbling itself is carried out entirely at the symbolic level: we show that the generator of garbled circuits always produces outputs that are symbolically indistinguishable from those of a simulator. In this formulation, the problem becomes a problem of equivalence

of two programs, whose proof does not rely on any cryptographic assumptions or reductions and is therefore significantly simpler.

At the same time, we also formalize the proof of correctness of the symbolic approach itself (the soundness theorem). As a result, the final theorem we obtain is a standard (computational) security result for circuit garbling—the symbolic method is used merely as a proof technique and does not weaken the security guarantee. Paradoxically, by using cryptographic reductions only to prove a much more general statement (the correctness of the symbolic framework rather than a specific protocol), our overall proof becomes more transparent and easier to understand.

2.3.2 *My contributions*

In the work "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma" my main contribution is the lower-bound proof, which establishes the minimum communication required for any secure Plasma protocol and formally separates Plasma protocols into two distinct categories.

In the work "A Formally Verified Lightning Network" I proposed the main ideas behind the project (to model the Lightning Network in Why3). I identified the overall strategy and provided the initial proof of concept. I led the project and contributed substantially to both the codebase and the proof development (about 33%). I identified the possibility of introducing an abstraction layer that encapsulates implementation-specific parts of the client ("party interface") and isolates the required properties.

In the work "Computationally Sound Symbolic Cryptography in Lean" I contributed significantly to the formalization effort, authoring roughly half of the resulting codebase. I developed the pseudo-fixpoint concept, which is used to characterize the outputs of computations involving fixpoints.

3

LOWER BOUNDS FOR OFF-CHAIN PROTOCOLS: EXPLORING THE LIMITS OF PLASMA

AUTHORS

Stefan Dziembowski, Grzegorz Fabianski, Sebastian Faust, Siavash Riahi

3.1 ABSTRACT

Blockchain is a disruptive new technology introduced around a decade ago. It can be viewed as a method for recording timestamped transactions in a public database. Most of blockchain protocols do not scale well, i.e., they cannot process quickly large amounts of transactions. A natural idea to deal with this problem is to use the blockchain only as a timestamping service, i.e., to hash several transactions $tx_1, \ldots, tx_m$ into one short string, and just put this string on the blockchain, while at the same time posting the hashed transactions $tx_1, \ldots, tx_m$ to some public place on the Internet ("off-chain"). In this way the transactions tx_i remain timestamped, but the amount of data put on the blockchain is greatly reduced. This idea was introduced in 2017 under the name *Plasma* by Poon and Buterin. Shortly after this proposal, several variants of Plasma have been proposed. They are typically built on top of the Ethereum blockchain, as they strongly rely on so-called *smart contracts* (in order to resolve disputes between the users if some of them start cheating). Plasmas are an example of so-called *off-chain protocols*.

In this work we initiate the study of the inherent limitations of Plasma protocols. More concretely, we show that in every Plasma system the adversary can either (a) force the honest parties to communicate a lot with the blockchain, even though they did not intend to (this is traditionally called *mass exit*); or (b) an honest party that wants to leave the system needs to quickly communicate large amounts of data to the blockchain. What makes these attacks particularly hard to handle in real life is that these attacks do not have so-called *uniquely attributable faults*, i.e. the smart contract cannot determine which party is malicious, and hence cannot force it to pay the fees for the blockchain interaction. An important implication of our result is that the benefits of two of the most prominent Plasma types, called *Plasma Cash* and *Fungible Plasma*, cannot be achieved simultaneously.

Besides of the direct implications on real-life cryptocurrency research, we believe that this work may open up a new line of theoretical research, as, up to our knowledge, this is the first work that provides an impossibility result in the area of off-chain protocols.

3.2 INTRODUCTION

What does it mean to *timestamp* a digital document? Haber and Stornetta in their seminal paper [56] define timestamping as a method to certify when a given document was created. In many settings the timestamped document T remains secret after it was timestamped, until its creator decides to make it public. This is often because of efficiency reasons – for example in the scheme of [56] what is really timestamped is the cryptographic hash[1] $H(T)$ (not T), which leads to savings in communication between T's creator and the timestamping service. Sometimes the secrecy of T is actually a desired feature. According to [97] several researchers in the past (including Galileo Galilei and Isaac Newton) have used ad-hoc methods to timestamp their research ideas before publishing them, in order to later claim priority.

Recently, in the context of blockchain, timestamping has been used in a slightly different way. Namely, in the paper that introduced this technology [79], the "timestamping" mechanism is such that T's creator does not only get a proof that T has been created at a given time, but also that it has been *made public* at this time. Let us call this kind of scheme *public timestamping*, and let us refer to the former type of timestamping as *secret* timestamping. The public timestamping feature of blockchain has been one of the main reasons why this technology attracted so much attention. In fact one of the first projects that use blockchain for purposes other than purely financial was *Namecoin* that used the timestamping to create a decentralized domain name system (see, e.g., [59] for more on this project).

In many cases timestamping is expensive, and its costs grow linearly with the length of the timestamped document. This is especially true for blockchain-based solutions, where all parties in the network need to reach *consensus* about what document was published and when. For example, Bitcoin (the blockchain system introduced in [79]) can process at most around 1MB of data per 10 minutes. In Ethereum, which is another very popular blockchain system [99] "timestamping" a word of 32 bytes cost currently around USD 0.80. A similar problem appears in several other blockchain protocols. We give a short introduction to blockchain in Sec. 3.2.1. For a moment let us just say that typical blockchains come with their own virtual currencies (also called *cryptocurrencies*). In the most standard case the "timestamped" messages define financial transfers between the network participants, and are hence called *transactions*. We will refer to timestamping a transaction as *posting it on the blockchain*.

To summarize, from the efficiency point of view the secret timestamping is better than the public one (as only hashes need to be timestamped). From the security perspective these two types of timestamping are incomparable as the fact that the timestamped document T needs to be published can be considered to be both an advantage

1 In this paper we assume reader's familiarity with basic cryptographic notions such as hash functions, negligible functions, etc. For an introduction to this topic see, e.g., [60].

and a disadvantage, depending on the particular application. For example, one could argue that considering timestamped hashes of academic papers as being sufficient evidence for claiming priority would slow down scientific progress, as it would disincentive making the papers public (until, say, the author writes down all followup papers that build upon the timestamped paper).

A natural question therefore is as follows. Suppose we have access to a timestamping service that is expensive to use (so we would rather timestamp only very short messages there). Can we use it to "emulate" public timestamping that would be cheap for long documents T? Of course this "emulation" cannot work in general, since some scenarios may simply require a proof that the whole T was available publicly at a given time. So the best we can hope for is to find *some* applications which permit such emulation. An idea for such emulation emerged recently within the cryptocurrency community under the names *Plasma* or *commit chains*[2] [54, 61, 76, 85, 94]. Very informally speaking, Plasma allows to "compress" a number of blockchain transactions $tx_1, \ldots, tx_m$ into one very short string $h = H(T)$, where $T := (tx_1, \ldots, tx_m)$. The transactions can from from different parties called *users*. The only document that gets posted on the blockchain is h. The compression and posting h is done by one designated party called the *operator* (denoted *Op*). Besides of posting h, the operator publishes all the transactions T on some public network (say: on her web-page). Since this publication is *not* done on the blockchain we also say that it is performed *off-chain*.

Publishing T off-chain is important since only then each user can verify that her transaction was indeed included into the hashed value. Moreover, typical Plasma designs use as H the so-called *Merkle-tree hashing* with $tx_1, \ldots, tx_m$ being the labels of the leaves (see the full version of this paper [40] for more on this technique). Thanks to this, every user can prove that his transaction was included in $H(T)$ with a proof of length $O(\log n)$. If the operator does not include some tx_i in T then the user U that produced tx_i can always post tx_i directly to the blockchain. In this case using Plasma does not bring any benefits to U compared to just using the blockchain directly from the beginning. However, in reality it is expected that this is not going to happen often, especially since the operators are envisioned to be commercial entities that will charge some fee for their services.

What is much more problematic is the case when the malicious operator does *not* publish T off-chain. This situation is typically referred to as *data unavailability*. Note that data unavailability is subjective, i.e., the parties can have different views on whether it happened or not. This is because, unlike the situation on the blockchain, there may be no consensus on what was published off-chain. Moreover, there is no way to produce a proof that data unavailability happens: even if some parties complain on the blockchain that they did not receive T there is no way to determine (just by looking at the blockchain) whether they are right, or if they are just falsely accusing the operator. The situation when a disagreement between parties happens but

2 In this paper we mostly use the name "Plasma" due to its brevity.

there is no "blockchain-only" way to verify which party is corrupt, is commonly referred to as a *non-uniquely attributable fault*. Finally, some Plasma protocols require all the honest parties to immediately act on blockchain after a data unavailability attack happened. This is called "mass exit", although (for the reasons explained in Sec. 3.2.1 in this paper we call it "large forced on-chain action")

Plasma comes in many variants and has been discussed in countless articles (see Sec. 3.2.1). One of the most fundamental distinction is between the two types of Plasma systems: *Plasma Cash* and *Fungible Plasma*. As we explain in more detail in Sec. 3.2.1 they both serve for the "emulation" that we outlined above, but have different incomparable features. From one point of view Fungible Plasma is better than Plasma Cash since it is "fungible", which means that the money can be arbitrarily divided and merged. On the other hand: Fungible Plasma suffers from some problems that Plasma Cash does not have, namely the adversary can cause a "non-uniquely attributable large forced mass actions" (we explain these notions Sec. 3.2.1). The cryptocurrency community has been unsuccessfully looking for a Plasma solution that would have the benefits of both Plasma Cash and Fungible Plasma. The main result of this paper is that such Plasma cannot be achieved simultaneously. In other words, we show inherent limitations of the compression technique, at least for compressing blockchain financial transactions. Our paper can also be viewed as initiating the theoretical analysis of lower bounds for the smart contract protocols. We write more about our contribution in Sec. 3.2.2. First, however, let us provide some more introduction to blockchain and to Plasmas.

3.2.1 *Introduction to Plasma*

Let us start with providing some more background on blockchain and smart contracts. Blockchain can be viewed as a public *ledger* containing timestamped transactions that have to satisfy some correctness constraints. Moreover, several blockchains permit to execute the so-called *smart contracts* [91] (or simply: "contracts"), which informally speaking, are "self-executable" agreements described in form of computer programs. Examples of such blockchain platforms include *Ethereum*, *Hyperledger Fabric*, or *Cardano*. Typically, it is assumed that contracts are deterministic and have a public state. Moreover, they can own some coins. Executing a contract is done by posting transactions on the blockchain and it costs fees that depend on the computational complexity of the given operation, and on the amount of data that needs to be transmitted to the contract.

Let us now explain the basic idea of Plasma, and introduce some standard terminology. Since it is an informal presentation, we mix the definition of the protocol with its construction. In the formal sections of the paper these two parts are separated (the definition appears in Sec. 3.3, and the constructions in the full version of this paper [40]). As highlighted above Plasma address the scalability problem of blockchain by keeping the massive bulk of transactions outside

of the blockchain ("off-chain"). The parties that are involved in the protocol rely on a smart contract that is deployed on the ledger of the underlying cryptocurrency, but they try to minimize interacting with it. Typically, this interaction happens only when the parties join and leave the protocol, or when they disagree. Since all parties know that in case of disagreement, disputes can always be settled on the ledger, there is no incentive for the users to disagree, and honest behavior is enforced.

In the optimistic case, when the parties involved in the protocol play honestly, and the off-chain transactions never hit the ledger, these protocols significantly reduce transaction fees and allow for instantaneous executions. Off-chain protocols also resemble an idea explored in cryptography around two decades ago under the name "optimistic protocols" [8, 30]. In this model the parties are given access to a trusted server that is "expensive to use", and hence they do not want to contact it, unless it is absolutely necessary

Plasma's operator *Op* provides a "simulated ledger", in which other parties can deposit their coins, and then perform operations between each other. The key requirement is that its users do not need to trust the operator, and in particular if they discover that she is cheating, then they can safely withdraw their funds. The latter is called an *exit* from the simulated ledger, and requires communication with the underlying ledger.

Plasma protocols come in different variants (see Sec. 3.2.1), however, they are all based on a single framework proposed in [85]. The parties that execute Plasma are: the *users* $U_1, \ldots, U_n$, and the *operator* Op. Moreover, the parties have access to a contract on the blockchain. In our formal modeling this contract will be represented as a trusted interactive machine Γ with public state, owning some amount of coins. Each user U_i has some number of coins initially deposited in his Plasma account which is maintained by Γ. This number is called a *balance* and is denoted with $b_i \in \mathbb{Z}_{\geq 0}$. Users' balances are changing dynamically during the execution of the protocol. The total number of coins owned by the contract Γ is equal to the sum of all balances of its users. A vector $\overrightarrow{b} := (b_1, \ldots, b_n)$ is called a *Plasma chain*. When referring to the underlying blockchain (i.e. the one on which Γ is deployed) we use the term *main chain*. Note that the operator Op has no account and only facilitates transfers of the users. In some variants of Plasma (see Sec. 3.2.1) the operator blocks some amount of coins (called operator's *collateral*) that can be used to compensate the users their losses in case she misbehaves.

Let us briefly describe the different operations that parties of the Plasma protocol can execution during the lifetime of the system. We divide time into *epochs* (e.g. 1 epoch takes 1 hour). In the ith epoch the operator sends some information C_i to Γ. We can think of C_i as "compressed" information about the vector $(b_1, \ldots, b_n)$ containing the users' balances. By "compressed" we mean that $|C_i|$ is much shorter than the description of $(b_1, \ldots, b_n)$, and usually its length is constant in every epoch. We will refer to C_i as a "commitment" to $(b_1, \ldots, b_n)$. The length of C_i is called the *commit size*.

In each epoch every user U_i can request to *exit*, by which we mean that all b_i coins from her Plasma account get converted to the "real" coins on the main chain, and she is no longer a part of this Plasma chain (which, in our formal modeling will be indicated by setting $b_i := \perp$). Plasma's security properties guarantee that every user can exit with all the coins that she currently has in the given Plasma chain. It is often required that exiting can be done cheaply, and in particular that the total length of the messages sent by the exiting user to Γ is small. The amount of data that a user needs to send to Γ in order to exit the Plasma chain is called the *exit size*.

Finally, any two users of the same Plasma chain can make *transfers* between each other. Suppose U_k wants to transfer v coins from her account to U_j. This transfer operation involves only communicating with U_j, and with the operator Op, while no interaction with the contract Γ is needed. Under normal circumstances (i.e. when the operator is honest) the next Plasma block that is committed to the main chain will simply have v coins deduced from U_k's account and v coins added to U_j's account.[3]

3.2.1.1 *Challenges in designing Plasma systems.*

The main challenge when designing a Plasma system is to guarantee that every user can exit with her money. This is usually achieved as follows: each C_i is a commitment to $(b_1, \ldots, b_n)$, computed using a Merkle tree. An honest operator Op is obliged to obey the following rule:

EXPLAINING COMMITMENTS — each time Op sends C_i to Γ, she sends the corresponding $\overrightarrow{b} := (b_1, \ldots, b_n)$ to all the users.

Technically, sending $\overrightarrow{b}$ to the users can be realized, e.g., by publishing it on the operator's web-page (i.e.: "off-chain"). Every user U_j can now check if she has the correct amount on her account and if C_i was computed correctly. Moreover, thanks to the properties of Merkle trees, U_j has a short proof of size $O(\log(n))$ that b_j has been "committed" into C_i.

The above description assume that the operator is honest. If she is corrupt things get more complicated. Note that C_i sent by the operator to Γ is publicly known (due to the properties of the underlying blockchain). Hence, we can assume that all the users agree on whether C_i was published and what is its exact value. The situation is different when it comes to the vector $\overrightarrow{b}$ that should be published off-chain. In particular, if $\overrightarrow{b}$ has *not* been published, then the users have no way to prove this to Γ. This is because whether some data has been published off-chain or not does not have a digital evidence that can be interpreted by Γ. This leads us to the following attack that can be carried out by a malicious operator, and is intensively studied by the cryptocurrency community (see, e.g., [26]).

3 To keep things simple, in this paper we do not discuss things like "transfer receipts", i.e., confirmations for the sender that the coins have been transferred.

DATA UNAVAILABILITY ATTACK — in this attack the corrupt operator publishes C_i but does not publish $\overrightarrow{b}$.

Note that this attack has no extra cost for the operator because from the point of view of the smart contract Γ, the operator behaves honestly, and hence the users cannot complain to Γ and request, e.g., that Op sends $\overrightarrow{b}$ to Γ. Furthermore determining if this attack happened is "subjective", i.e., every user U_j has to detect it herself. Moreover, in case of data unavailability it is impossible for Γ to determine whether U_j or Op is dishonest, since a sheer declaration of U_j that Op did not send him the data obviously cannot serve as a proof that it indeed happened. This leads to the following definition.

NON-UNIQUELY ATTRIBUTABLE FAULTS — this refers to the situation when the contract has to intervene in the execution of the protocol (because the protocol is under attack), but it unable to determine which party misbehaved (see, e.g., [25]).

Non-uniquely attributable faults appear typically in situations when a party claims that it has not received a message from another party. In contrast if a contract *is* able to determine which party is corrupt then we have a *uniquely*-attributable fault. A typical example of such a fault is when a party signs two contradictory messages. Unfortunately, non-uniquely attributable faults are hard to handle in real life, since it is not clear which party should pay the fees for executing the smart contract, or which party should be punished for misbehaving. In particular, what is unavoidable in such a case is that a malicious party P can force another participant P' to lose money on fees (potentially also loosing money herself). This phenomenon is known as *griefing* [25].

When a user realizes that the operator is dishonest, then she often needs to start quickly interacting with Γ in order to protect her coins. This action has to be done quickly, and has to be performed by each honest user. This leads to the following definition.

FORCED ON-CHAIN ACTION OF SIZE α — this term refers to the situation that honest parties who did not intend to perform an exit are forced by the adversary to quickly interact with Γ, and the total length of the messages sent by them to Γ is α. Informally, when α is large (e.g. $\alpha = \Omega(n)$) we say this is a *mass* forced on-chain action.

Note that this definition talks about all honest "parties", and hence it includes also the case when the operator is honest, but it is forced to act because of the behavior of the corrupt users. Typically $\alpha = \Omega(n)$ and by "quickly" we mean "1 epoch". In most Plasma proposals [27, 64, 85] "interacting with the smart contract" means simply exiting the Plasma chain with all the coins. Hence, a more common term for this situation is "mass exit". Since in our work we are dealing with the lower bounds, we need to be ready to cover also other, non-standard, ways of protecting honest users' coins. For example, it could be the case that a user U_j does *not* exit immediately, but, instead, keeps her

coins in a special account "within Γ" and withdraws them much later. Of course, this requires interacting with Γ immediately, but, technically speaking does not require "exiting". To capture such situations, we use the term "forced on-chain *action*", instead of "mass *exit*".

After a party announces an exit, we need to ensure that she is exiting with the right amount of coins. The main problem comes from the fact that we cannot require that users exit from the last C_i by sending the explanation for her balance b_i to Γ. This is because it could be the case that a given user does not know the explanation $\overrightarrow{b}$ of C_i (due to the data unavailability attack). For a description of how this can be done in practice see [27, 85], or the full version of this paper [40].

Mass exits (or large forced on-chain actions) caused by data unavailability are considered a major problem for Plasma constructions. They are mentioned multiple times in the original Plasma paper [85] (together with some ad-hoc mechanism for mitigating them). They are also routinely discussed on "Ethereum's Research Forum"[4], with even conferences organized on this topic[5]. One of the main reasons why the mass exits are so problematic is that they may results in blockchain congestion (i.e., situations when too many users want to send transactions to the underlying blockchain). Moreover, the adversary can choose to attack Plasma precisely in the moments when the blockchain is already close to being congested (see, e.g., [98] for a description of real-life incident of the Ethereum blockchain congestion). She can also attack different Plasma chains (established over the same main chain) so their users simultaneously send large amounts of data to the blockchain. In order to be prepared for such events in real-life Plasma proposals it is sometimes suggested that the time T for reacting to data unavailability should be very large (e.g. $T = 2$ weeks). This, unfortunately, has an important downside, namely that also an honest coin withdrawal requires time T.

Consider a non-fungible Plasma that supports coin identifiers from some set $\mathcal{C}$. In a non-fungible Plasma the Plasma chain is of a different form than before: instead of a vector of balances $\overrightarrow{b}$, it is a function $f : \mathcal{C} \to \{U_1, \ldots, U_n, \perp\}$ that assigns to every coin $c \in \mathcal{C}$ its current owner $f(c)$ (or $\perp$ if the coin has been withdrawn). Similar to before the commitment to the value of f will be done using Merkle trees. Whenever a coin is withdrawn its identifier c is sent to the smart contract Γ, and hence it becomes public. This is important for the mechanism that prevents parties from stealing coins. To this end, each user U monitors Γ, and sends a complaint whenever some (corrupt) user U' tries to withdraw one of U's coins. For the contract Γ to decide if c belongs to U or U' can require some additional interaction, but the system is designed in such a way that the honest user is guaranteed that finally she will win such dispute. Hence, every malicious attempt to withdraw someone else's coin will be stopped.

The main difference between Plasma Cash and Fungible Plasma is that in Plasma Cash every user has to "protect" only her own

4 Available at: ethresear.ch.

5 See: ethresear.ch/t/data-unavailability-unconference-devcon4.

coins. Thanks to this, even in case of the data unavailability attack, each honest user U does *not* need to immediately take any action. Instead, she can just monitor Γ, and has to act only if someone tries to withdraw one of U's coins. Of course, the corrupt user can still force all the honest ones to quickly act on the blockchain. However, this requires much more effort from them than in Fungible Plasma, namely: they need to withdraw many coins of the honest users at once, hence forcing the honest users to react. This is "fairer" both honest and malicious users have to make similar effort. Most importantly, however, this attack has *uniquely-attributable faults*.

This advantage of Plasma Cash comes at a price, namely the "exit size" is not constant anymore, as it depends on the number of coins that a user has (since each coin has to be withdrawn "independently"). The Ethereum research community has been making some efforts to deal with this problem. One promising approach is to "compress" the information about withdrawn coins. For example one could assume that the identifiers in $\mathcal{C}$ are natural numbers. Then a user U who owns coins from some interval $[a, \ldots, b]$ (with $a, b \in \mathbb{N}$) could simply withdraw them by posting a message "User U withdraws all coins from the interval $[a, \ldots, b]$ (instead of withdrawing each $i \in [a, \ldots, b]$ independently). This, of course, works only if the coins that users own can be divided into such intervals. Some authors (in particular V. Buterin) have been suggesting "defragmentation" techniques for achieving such a distribution of coins. This is based on the assumption that the parties periodically *cooperate* to "clean up" the system. Hence, it does not work in a fully malicious settings[6] (if the goal of the adversary is to prevent the cleaning procedure).

3.2.1.2 *The landscape of Plasmas.*

Soon after the original groundbreaking work on Plasma [85], some concrete variants have been proposed. Some of them we already described in Sec. 3.2.1. Since this paper focuses on the *impossibility* results, we do not provide a complete overview of the many different variants that exist and what features they achieve. Plasma projects that are frequently mentioned in the media are Loom, Bankex, NOCUST and OmiseGO [76, 94]. This area is mostly developed by a very vibrant on-line community that typically communicates results in form of so-called "white-papers", blog articles, or post on discussion forums (such as the "Ethereum Research Forum", see footnote 4 on page 26). See also the diagram called "Plasma World Map" illustrating the different flavors of Plasma in the full version of this paper [40]. A notable exception are NOCUST and NOCUST-ZKP described in [61]. This work, up to our knowledge, is the first academic paper on this topic. It provides a formal protocol description (with several interesting innovations such as " Merkle interval trees") and a security argument. Moreover, the authors of [61] describe a version of NOCUST that puts a collateral on the operator (this is done in order to achieve instant

6 See ethresear.ch/t/plasma-cash-defragmentation and subsequent posts by Buterin on the Ethereum Research Forum.

transaction confirmation). The authors of [61] (see also [54]) introduce the term "commit chains". Yet, unfortunately, they do not define a full formal security model that we could re-use in our work.

Let us also mention some of the so-called "distributed exchanges" that look very similar to Plasma. One example is StarkDEX (informally described in [48]), which is also based on the idea of a central operator batching transactions using Merkle trees, and a procedure for the users to "escape" from the system if something goes wrong. This protocol uses non-interactive zero-knowledge protocols to ensure correctness of the operator's actions (similar approach has been informally sketched in the original Plasma paper [85], and has been also used in NOCUST-ZKP [61]). While zero knowledge can be used to demonstrate that some data was computed correctly, it *cannot* be used to prove that the off-chain data *was published at all*. Consequently, the authors of this system also encountered the challenge of handling the data unavailability attack. Currently, in StarkDEX this problem is solved by introducing an external committee that certifies if data is available.[7] StarkDEX plans to eventually replace the committee-based solution with an approach that is only based on trusting the underlying blockchain. Our result however shows that in general this will be impossible, as long as fungibility and short exits are required (unless the operator puts a huge collateral).

3.2.2 *Our contribution and organization of the paper*

We initiate the study of lower bounds (or: "impossibility results") in the area of off-chain protocols. Our results can also be viewed as a part of a general research program of "bringing order to Plasma". We believe that the scientific cryptographic community can provide significant help in the efforts to systematize this area, and to determine the formal security guarantees of the protocols (in a way that is similar to the work on "Bitcoin backbone" [47], or more recently on "Mimblewimble" [46], state channels [38], or the Lightning Network [62]). Investigating the limits of what Plasma can achieve is part of this process. We focus on proving lower bounds that concern the necessity of mass forced on-chain actions, especially caused by the attack that have no uniquely attributable faults (as a result of data unavailability). This is motivated by the fact that such attacks are particularly important for the off-chain protocols: since the main goal of such protocols is to move the transactions *off*-chain, the necessity of quickly acting *on*-chain can be viewed as a big disadvantage.

We start with a formal definition of Plasma (this is done in Sec. 3.3). Since in this work we are interested only in the impossibility results, our definition is very restrictive for practical systems. By "restrictive" we mean that we make several assumptions about how the protocol operates. For example we have very strict synchronicity rules, and in particular we only allow the users to start the Plasma operations in certain moments (see "payment orders" and "exit orders" phases in

7 See their FAQs at https://www.starkdex.io.

Sec. 3.3.1). Obviously, such restrictions make our lower bounds only stronger, since they also apply to a more realistic model (without such restrictions). We believe that fully formalizing real-life Plasma (e.g. in the style of [39, 41, 62]) is an important future research project, but it is beyond the scope of this work.

Our main result is presented in Thm. 1, stated formally in Sec. 3.4. It states that in certain cases the adversary can force mass on-chain actions of the honest users of *any* Plasma system. One subtle point that we want to emphasize is that whenever we talk about "forcing actions" on the honest users, we mean a situation when the users that did *not* want to exit (in a given epoch) are forced to act on-chain. This is important, as otherwise our theorem would hold trivially (one can always imagine a scenario when lots of users decide to exit Plasma because of some other, external, reasons). The notion of "wanting to exit" is formalized by an *environment machine* $\mathcal{Z}$ (borrowed from the *Universal Composability* framework [31]) that "orders" the parties to behave in certain way.

More formally, Thm. 1 states that in Plasma either there exists an attack that provokes a mass action, or there is an attack that requires a party that exits to post long messages on the blockchain (i.e. this Plasma has large exits). Moreover, both attacks have *no* uniquely attributable faults. Note that, strictly speaking, this theorem also covers Plasma systems where the commit size is large (even $\Omega(n)$)[8], but in this case it holds trivially since the honest operator needs to send the large commitments to Γ even if everybody is honest (hence: there is an "unprovoked" mass action in every epoch).

The most interesting practical implication of this theorem is that it confirms the need for "two different" Plasma flavors, as long as the operator is not required to put aside a collateral of size comparable to the total amount of coins in the system[9]. One way to look at it is: either we want to have a Plasma system that does not have large exits, in which case we need to have (non-uniquely attributable) mass actions (this is Fungible Plasma/Plasma MVP); or alternatively we insist on having Plasma without such mass actions, but then we have to live with large exits (as in Plasma Cash). Our theorem implies that there is no Plasma that would combine the benefits of both Fungible Plasma and Plasma Cash, and hence can serve as a justification why both approaches are complementary. Before our result one could hope that the opposite is true and that, e.g., the only reason why Plasma Cash is popular is its relative simplicity (compared to Fungible Plasma). Besides of this reason, and the general scientific interest, we believe that our lower bound has some other important practical applications. In particular, lower bounds often serve as a guideline for constructing new systems or tweaking the definitions. We hope it will contribute to consolidate the countless research efforts in constructing new Plasma

8 In practice, Plasma systems with unbounded commit size are not interesting since they do not bring any advantages to the users. Moreover, they can be trivially constructed just by putting every transaction on the main chain.

9 This is clearly impractical for most of the applications. Actually most of Plasma constructions assume no such collateral at all.

systems[10] and simplify identifying proposals that are not sound (e.g., because they claim to achieve the best of both worlds).

Let us also stress that our Thm. 1 does *not* rely on any assumptions of complexity-theoretic type and does *not* use a concept of "black-box separations" [58]. This means that the lower bound that we prove cannot be circumvented by introducing any kind of strong cryptographic assumptions. Hence, of course, it also holds for Plasmas that use non-interactive zero-knowledge (like NOCUST-ZKP [61] or StarkDEX [48]). Moreover, we manage to generalize our lower bound. Thm. 1 even holds for Plasma systems where the operator deposits a certain amount of coins for compensating parties for malicious behavior (e.g., it could be used when a malicious operator does not explain commitments).

For completeness, we also describe (in the full version of this paper [40]) "positive" results, i.e., two protocols that satisfy our security definition (Plasma Cash and Fungible Plasma). We stress that we do not consider it to be a part of our main contribution, and we do not claim novelty with these constructions, as they strongly rely on ideas published earlier (in particular [27, 28, 48, 61, 64, 85]).

3.2.2.1 *Notation.*

For a formal definition of an *interactive (Turing) machine* and a *protocol*, see, e.g., [31]. In our modeling the communication between the parties is synchronous and happens in *rounds* (see Sect. 3.3). During the execution of the protocol a party P may send messages to a party P'. A *transcript* of the messages sent from P to P' is a sequence $\{(m_i, t_i)\}_{i=1}^{\ell}$, where each m_i was sent by P to P' in the t_i-th round. A transcript of messages sent from some set of parties to a different set of parties is a sequence $\{(W_i, W_i', m_i, t_i)\}_{i=1}^{\ell}$, where each m_i was sent by W_i to W_i' in the t_i-th round. By the *length of a transcript* we mean its bit-length (in some fixed encoding). We sometimes refer to it also as *communication size* (between the parties).

3.3 PLASMA PAYMENT SYSTEMS

A *Plasma payment system* (or "Plasma" for short) is a protocol Π consisting of a randomized non-interactive machine Ψ representing the *setup* of the system; deterministic[11] interactive poly-time machines $U_1, \ldots, U_n, Op$ representing the *users* and the *operator* of the Plasma system (respectively); and a deterministic interactive poly-time machine Γ, which represents the *Plasma contract*. We use the notation $\mathcal{U} = \{U_1, \ldots, U_n\}$ to refer to the set of users of the system. The contract machine Γ has no secret state, and moreover its entire execution history is known to all the parties. We can think of it as a Turing machine that keeps the entire log of its execution history, and moreover all the other parties in the system have a (read-only) access to

10 see https://ethresear.ch/c/plasma/.

11 We assume that these machines are deterministic, since all their internal randomness will be passed to them by Ψ.

this log. The Plasma system comes with a parameter $\gamma \in \mathbb{R}_{\geq 0}$ called *operator's collateral fraction*. Informally, this parameter describes the amount of coins that are held by the operator as a "collateral" (as a fraction of user's coins). These coins can be used to cover users' losses if the operator misbehaves. This is formally captured in Sect. 3.3.2 (see "limited responsibility of the operator"). If $\gamma = 0$ then we say that the operator is *not collateralized*. We introduce the notion of collateral in order to make our results stronger and to cover also cases of real-life systems that have such a collateral (e.g., NOCUST, see Sect. 3.2.1).

The protocol is attacked by a randomized poly-time adversary $\mathcal{A}$. We assume that $\mathcal{A}$ can corrupt any number of users and the operator except the contract Γ (hence Γ can be seen as a trusted third party). Once $\mathcal{A}$ corrupts a party P, she learns all its secrets, and takes full control over it (i.e. she can send messages on behalf of P). A party that has not been corrupted is called *honest*. An execution of a Plasma payment system Π is parametrized by the *security parameter* 1^λ.

To model the fact that users perform actions, we use the concept of an environment $\mathcal{Z}$ (which is also a poly-time machine) that is responsible for "orchestrating" the execution of the protocol. The environment can send and receive messages from all the parties (it also has full access to the state of Γ). It also knows which parties are corrupt and which are honest. For an adversary $\mathcal{A}$ and an environment $\mathcal{Z}$, a pair $(\mathcal{A}, \mathcal{Z})$ will be called an *attack* (on a given Plasma system Π). The contract machine Γ can output special messages $(\mathsf{attribute\text{-}fault}, P)$ (where $P \in \mathcal{U} \cup \{Op\}$). In this case we say that Γ *attributed a fault to* P. We require that the probability that Γ attributes a fault to an honest party is negligible in λ. An attack $(\mathcal{A}, \mathcal{Z})$ *has no attributable faults* if the probability that Γ attributes a fault to some party is negligible.

3.3.1 *Protocol operation*

Let us now describe the general scheme in which a Plasma payment protocol operates. In this section, we focus only on describing what messages are sent between the parties. The "semantics" of these messages, and the security properties of the protocol are described in Sect. 3.3.2. We assume that all the parties are connected by authenticated and secret communication channels, and a message sent by a party P in the ith round, arrives to P' at the beginning of the $(i+1)$th round. The communication is synchronous and happens in *rounds*. It consists of three stages, namely: "setup", "initialization", and "payments". The execution starts with the *setup stage*. In this stage parameter 1^λ is passed to all the machines in Π. Upon receiving this parameter, machine Ψ samples a tuple $(\psi_{U_1}, \ldots \psi_{U_n}, \psi_{Op}, \psi_\Gamma)$ (where each $\psi_P \in \{0,1\}^*$). Then for each $P \in \{U_1, \ldots, U_n, Op, \Gamma\}$ the string ψ_P is passed to P. Afterwards, the parties proceed to the *initialization stage*. In this stage the environment generates a sequence $(a_1^{\mathsf{init}}, \ldots, a_n^{\mathsf{init}})$ of non-negative integers and passes it to the contract Γ (recall that the state of Γ is public, and hence, as a consequence, all the parties in the system also learn the a_i^{init}'). Then the protocol proceeds to the *payment*

stage. This stage consist of an unbounded number of *epochs*. Each ith epoch (for $i = 1, 2, \ldots$) is divided into two *phases*.

PAYMENT PHASE. In this phase the environment sends a number of *payment orders* to the users (for simplicity we assume that this happens simultaneously in a single round). Each order has a form of a message "(send, v, U_i)", where $v \in \mathbb{Z}_{\geq 0}$, and $U_i \in \mathcal{U}$. It can happen that some users receive no payment orders in a given epoch. It is also ok if a user receives more than one order in an epoch. Informally, the meaning of these messages is as follows: if a user U_j receives a "(send, v, U_i)" message, then she is ordered to transfer v coins to user U_i. We require that this message can only be sent if none of b_i and b_j are equal to $\perp$ (i.e.: if none of U_i and U_j "exited", see below). The parties execute a multiparty sub-protocol. During this executions some of the users send a message "$(\mathsf{received}, v, U_i)$" to the environment $\mathcal{Z}$. This sub-protocol ends when Γ outputs a message payments-processed.

EXIT PHASE. In this phase the environment sends *exit orders* to some of the users (again: this happens in a single round). Each such order is simply a message "exit". Informally, sending this message to some U_i means that U_i is ordered to exit the system with all her coins. The environment can send an exit message to U_i only if $b_i \neq \perp$ (i.e. U_i has not already "exited", see below). The parties again execute a multiparty protocol. The protocol ends when Γ outputs a sequence

$$\{(\mathsf{exited}, U_{i_j}, v_{i_j})\}_{j=1}^{m}, \tag{3.1}$$

where m is some non-negative integer, and each $U_{i_j} \in \mathcal{U}$ and $v_{i_j} \in \mathbb{Z}_{\geq 0}$. For each U_{i_j} in Eq. (3.1) we say that U_{i_j} *exited (with v_{i_j} coins)*, and we let $b_{i_j} := \perp$. We require that no party can exit more than once. In other words: it cannot happen that two messages $(\mathsf{exited}, U_i, v)$ and $(\mathsf{exited}, U_i, v')$ are issued by Γ.

We make some assumptions on the communication between the parties. Informally we require that if U and U' are some honest users, then the procedure of transferring coins from U to U' is done by a "sub-protocol" involving only parties in the set U and U'. Since we do not have a concept of "sub-protocol" this is formalized as follows:

COMMUNICATION LOCALITY. Two honest users U and U' exchange messages only in epochs in which they do transactions between each other (i.e. a message (send, U, v) is sent by $\mathcal{Z}$ to U', for some v).

This requirement is very natural since Plasma is supposed to work even when an arbitrary set of users is corrupt. Hence, relying on the other users' help in financial transfers would be impractical. Up to our knowledge all "pure" Plasma proposals in the literature satisfy this requirement. On the other hand: it may *not* hold if we incorporate some techniques that assume some type of cooperation between larger sets

of parties (e.g. consensus mechanisms). Examples include: Buterin's Plasma Chash defragmentation (where a large set of users has to regularly cooperate in order to "clean-up" the system), and StarkDEX's "data availability committee" (see Sect. 3.2.1), if we treat the committee members as "users". One way to view our result is that it implies that such techniques are inherent for every fungible Plasma.

3.3.2 *Security properties*

During the interaction with the protocol, the environment keeps track of balances of *honest* users (we do not define balances of dishonest users). Formally, for each honest user U_i it maintains a variable $b_i \in \mathbb{Z}_{\geq 0} \cup \{\bot\}$ (a *balance* of U_i), where the symbol "$\bot$" means that a party exited. It also maintains a variable $t \in \mathbb{Z}_{\geq 0}$ (initially set to 0) that is used to keep track on the amount of coins that have been withdrawn. The rules for maintaining these variables are as follows. Initially, for each $i := 1, \ldots, n$ the environment $\mathcal{Z}$ lets $b_i := a_i^{\mathsf{init}}$. Whenever Γ outputs $(\mathsf{exited}, U_i, v)$ (for some U_i and v) we let $b_i := \bot$ and increment t by v. Each time $\mathcal{Z}$ receives a message $(\mathsf{received}, v, U_i)$ from some *honest* U_j, it adds v to b_j (recipient balance) and, if U_i (sender) is honest too, subtracts v from b_i. We require that the environment never issues an order if U_i or U_j exited (i.e. if $b_i = \bot$ or $b_j = \bot$). The environment also never sends an order exit to the same user more than once, and it never sends exit order to a user U_i that already exited (i.e. such that $b_i = \bot$). We have the following security properties.

RESPONSIVENESS TO "send" ORDERS. Suppose Op, U_i, and U_j are honest, and the environment issued an order (send, v, U_i) to U_j then in the same epoch party U_i sends a message "$(\mathsf{received}, v, U_j)$" to the environment.

CORRECTNESS OF "received" MESSAGES. Suppose U_i and U_j are honest and U_i outputs a message "$(\mathsf{received}, v, U_j)$", then environment has issued an order (send, v, U_i) to U_j in the same epoch.

RESPONSIVENESS TO "exit" ORDERS. Suppose U_i is honest and the environment issued an order exit to U_i then in the same epoch Γ outputs a message $(\mathsf{exited}, U_i, v)$ (for some v).

NO FORCED EXITS IF OPERATOR HONEST. Suppose Op and U_i are honest and Γ outputs message $(\mathsf{exited}, U_i, v)$ at epoch r, then environment has sent the order exit to U_i in the same epoch.

FAIRNESS FOR THE USERS. If Γ outputs a message $(\mathsf{exited}, U_i, v)$ (for some honest U_i) then $v \geq b_i$ (where b_i is the current balance of U_i).

LIMITED RESPONSIBILITY OF THE OPERATOR. If the operator is honest, then the total amount of coins that are withdrawn from the system is at most $a_1^{\mathsf{init}} + \cdots + a_n^{\mathsf{init}}$. Otherwise (if she is dishonest) the total amount of coins that are withdrawn from the system

is at most $\lceil (1+\gamma)(a_1^{\mathsf{init}} + \cdots + a_n^{\mathsf{init}}) \rceil$. This definition captures the notion of operator's collateral, and the fact that it is used (to cover users' losses) if the operator is caught cheating.

If an attack $(\mathcal{A}, \mathcal{Z})$ succeeds to violate any of the requirements from this section, then we say that $(\mathcal{A}, \mathcal{Z})$ *broke a given Plasma payment system*. We say that Π is *secure* if for every environment $(\mathcal{A}, \mathcal{Z})$ the probability that $\mathcal{A}$ breaks Π is negligible in 1^λ.

As explained in the introduction, certain attacks on Plasma are of particular importance, due to the fact that they are hard to handle in real life. We say that $(\mathcal{A}, \mathcal{Z})$ *force an on-chain action of size M (in some epoch i)* if the following happened. Let T be the set of honest parties that did *not* receive any order from $\mathcal{Z}$ in epoch i. Then the total length of messages sent by parties from T to Γ is at least M. As explained in the introduction, the term that is more standard than "forced on-chain action" is "mass *exit*". See 3.2.1 for a discussion why "forced on-chain action" is a better term when impossibility results are considered.

3.4 OUR MAIN RESULT

We now present Thm. 1, which is the main result of this paper. The main implication of this theorem is that for every non-collateralized Plasma system there exists an attack that provokes a mass forced on-chain action, i.e., it forces the honest users to make large communication with the contract even if they did not receive any exit order from the environment (see point 1 in the statement of the theorem), unless a given Plasma system has large exits (point 2). Moreover, this can be done by an attack that has *no* uniquely attributable faults. This fact cannot be circumvented by putting a collateral on the operator, unless this collateral is very large.

Theorem 1 (Mass forced on-chain actions or large exits without uniquely attributable faults are necessary)**.** Let Π be a secure Plasma payment system with n users and let $\gamma \geq 0$ be the operator's collateral fraction. Then either

1. there exists an attack on Π that causes a forced on-chain action of size greater than $(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$ with probability at least $1/16 + \mathsf{negl}[\lambda]$, or

2. there exists an attack on Π such that *one* honest user, when ordered to exit by the environment, makes communication to Γ of size at least $(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$ with probability at least $1/16 + \mathsf{negl}[\lambda]$.

Moreover, both attacks have no uniquely attributable faults.

One way to look at this theorem is as follows. First, consider a non-collateralized Plasma, i.e., assume that $\gamma = 0$. Let $\mathcal{P}^1$ be a class of non-collateralized Plasma that with overwhelming probability *do not* have uniquely attributable forced on-chain actions (of any size larger than 0). In this case point 1 cannot hold, and hence, every Plasma

$\Pi \in \mathcal{P}^1$ needs to satisfy point 2. This means that there exists an attack on every $\Pi \in \mathcal{P}^1$ such that one honest user, when ordered to exit by the environment, makes communication to Γ of size at least $(n-5)/4$ with probability around $1/16$. Or, in other words: every Plasma from class $\mathcal{P}^1$ must have a large exist size with noticeable probability. We know Plasma with such properties: it is essentially Plasma Cash (see the full version of this paper [40])

On the other hand, let $\mathcal{P}^2$ be a class of non-collateralized Plasmas that with high probability have no large exits, in the sense of point 2 of Thm. 1. This means that point 1 has to hold, which implies that every $\Pi \in \mathcal{P}^2$ needs to have large (at least around $(n-5)/4$) non-uniquely attributable mass forced on-chain actions. Plasma with such properties is called Fungible Plasma (see the full version of this paper [40]) Hence, informally speaking, Thm. 1 states that we cannot have the "best of two types of Plasma" simultaneously.

If we consider non-zero collaterals, i.e., we let $\gamma > 0$ then the situation does not improve much, unless the collateral fraction is large, i.e., the total collateral blocked by the operator is at least around $n \cdot \gamma = n/\log_2 n$[12]. This essentially means that we cannot get around the bounds from Thm. 1 by introducing collateral, unless the amount of coins blocked in operator's collateral is of roughly the same order as the total amount of coins stored by the users.

Note that even trivial versions of Plasma "fit" into Thm. 1. For example, consider Plasma in which the operator always puts all the transactions on-chain. Of course, the details would need to be worked out, but clearly such a Plasma can be made secure. The existence of such a trivial Plasma does not contradict our Thm. 1, since it clearly satisfies point 1: a large number of transfers in one epoch will cause a forced mass on-chain action (by the operator). The same also holds if every user needs to put each transaction on-chain.

3.5 PROOF OF THM. 1

Before we present the proof let us introduce some auxiliary machinery. This is done in the next section.

3.5.1 *Isolation scenario*

Let Π be a Plasma payment system, let $\mathcal{Z}$ be an environment, and let $\mathcal{W}$ be some subset of the users of Π. We now introduce a procedure that we call *isolation of* $\mathcal{W}$. In this scenario Π is executed as in the normal execution, except that we "isolate" the users $\mathcal{W} \subseteq \mathcal{U}$ from the operator. More precisely: all the messages sent between any $U \in \mathcal{W}$ and the operator Op are dropped, i.e., they never arrive to the destination. This scenario can be viewed as an "attack" although it does not fit into the framework from Sect. 3.3.1, since it violates the

12 This is because we need to have $\gamma \approx 1/\log_2 n$ to make the expression "$(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$" equal to 0.

assumption that messages sent by an honest party to another honest party always arrive to the destination.

Although the isolation scenario cannot be performed within our model, it can be "emulated" by corrupting either the operator Op, or the users from $\mathcal{W}$. In the first case we corrupt the operator and instruct him to behave as if she was honest, except that she does not send messages to the users in $\mathcal{W}$ and ignores all messages sent by these users. This will be called the *data unavailability (DU) attack against $\mathcal{W}$ by the operator*. In the second, symmetric case (the *pretended data unavailability (PDU) attack by $\mathcal{W}$ on the operator*) we corrupt the users in $\mathcal{W}$. Then, every user $U \in \mathcal{W}$ behaves as if she was honest, except that she does not send messages to Op, and ignores all messages from Op.

If this is the only type of malicious behavior, then "from the point of view" of all the other parties, and, most importantly, from the point of view of the contract machine Γ, it is impossible to say who is corrupt (the users in $\mathcal{W}$ or the operator Op). More precisely, we have the following.

Observation 1. Let Π be a Plasma payment system and consider the attack that isolates users in some set $\mathcal{W}$ from the operator. Let $\mathcal{Z}$ be an arbitrary environment and let $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{isolate}}$ be the random variable denoting the transcript of messages received by Γ. Moreover, let $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{PDU}}$ and $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{DU}}$ be the random variable denoting the transcripts of messages received by Γ in the PDU attack and in the DU attack (respectively), both with environment $\mathcal{Z}$. Then $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{DU}} \stackrel{d}{=} \mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{isolate}} \stackrel{d}{=} \mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{PDU}}$.

This fact is useful in the proof of the following simple lemma.

Lemma 1. Fix an arbitrary Plasma Π. Let $\mathcal{W}$ be some set of users. Suppose $\mathcal{A}$ performs a DU attack against $\mathcal{W}$ or a PDU attack by $\mathcal{W}$ (either by corrupting the operator or by corrupting the users), and let $\mathcal{Z}$ be arbitrary. Then the attack $(\mathcal{A}, \mathcal{Z})$ has no uniquely attributable faults.

Proof. From the security of Π we get that if the users are corrupt then the probability that Γ attributes a fault to them is negligible. Symmetrically, if the operator is corrupt then the probability that Γ attributes a fault to her is negligible. By Observation 1 the transcripts of messages received by Γ in both attacks are distributed identically, so the probability that Γ attributes *any* fault has to be negligible. □

3.5.2 *Proof overview*

Fix some secure Plasma payment system Π that works for n users. We construct either an attack such that

$$\Pr\left[\begin{array}{c}\text{the set of all honest users makes}\\ \text{communication to } \Gamma \text{ of size at least}\\ (n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4\\ \text{(without receiving an exit order from the}\\ \text{environment)}\end{array}\right] \geq 1/16 + \mathsf{negl}[\lambda],$$

(3.2)

or an attack such that

$$\Pr\left[\begin{array}{c}\text{user } U_1\text{, when ordered to exit by the}\\ \text{environment, makes communication to } \Gamma \text{ of}\\ \text{size at least}\\ (n-\lceil \gamma n\rceil \cdot \log_2 n - 5)/4\end{array}\right] \geq 1/16 + \mathsf{negl}[\lambda].$$

(3.3)

In both of these attacks the amount of coins given to the users is n, but our proof can be generalized to cover cases when it is required that the amount of coins is larger than n (we comment more on this at the end of Sect. 3.5). On the other hand, the proof does not go through in the (unrealistic) case when this amount is very small (sublinear in n).

The attacks that we construct in both cases ((3.2) and (3.3)) have no uniquely attributable faults. Note that for $n \leq 5$ Eq. (3.3) holds trivially, and therefore we can assume that $n > 5$. Let Y denote the family of all *non-empty proper* subsets of $\{U_2, \ldots, U_n\}$, i.e. sets $\mathcal{V}$ such that $\emptyset \subsetneq \mathcal{V} \subsetneq \{U_2, \ldots, U_n\}$ (note that $U_1 \notin \mathcal{V}$). Since we assumed that $n > 5$ we have that $\log |\mathrm{Y}| = \log_2(2^{n-1} - 2) \geq n - 2$, and, in particular, Y is non-empty. In the proof we construct an experiment (denoted by $\mathsf{Exp}(\mathcal{V})$ and presented in details in the full version of this paper [40]) and analyze its performance, assuming that $\mathcal{V}$ is sampled uniformly at random from Y. Depending on this analysis, the experiment $\mathsf{Exp}(\mathcal{V})$ can be "transformed" into an attack that satisfies Eq. (3.2) or Eq. (3.3).

Experiment $\mathsf{Exp}(\mathcal{V})$ "simulates" an execution of two epochs of Plasma Π. In the first epoch the adversary isolates the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ from the operator (in the attacks that we construct later this will be done either by corrupting these users, or the operator). The environment gives 1 coin to each user $U \in \mathcal{U}$. Then, in the "payment" phase of the first epoch all the users from $\mathcal{V}$ transfer their coins to U_1. In the "exit" phase of the first epoch user U_1 receives an exit order from the environment and consequently exits with all her coins. Note that in the first epoch every party behaved honestly (except of the isolation attack against the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$), and hence U_1 is guaranteed to successfully exit with her coins (she has 1 such coin from the "initialization" phase, and $|\mathcal{V}|$ coins that were transferred to her by the users in $\mathcal{V}$).

Of course the honest parties from $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ will usually realize that they are isolated from the operator. As a reaction to this they may send some messages to Γ. This, in turn can provoke the other parties to react by sending their messages to Γ. Hence, in general there can be a longer interaction between all the parties and Γ in this phase. Let $\mathcal{T}^1$ be the transcript of the messages sent by the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ to Γ in both phases, let $\mathcal{T}^2$ be the messages sent by the users in $\mathcal{V}$ and the operator to Γ in both phases, let $\mathcal{T}^3$ be the messages sent by U_1 to Γ in the "payment" phase, and finally let $\mathcal{T}^4$ be the messages sent by U_1 to Γ in the "exit" phase. The first epoch of the experiment $\mathsf{Exp}(\mathcal{V})$ and the transcripts are depicted on Fig. 3.1.

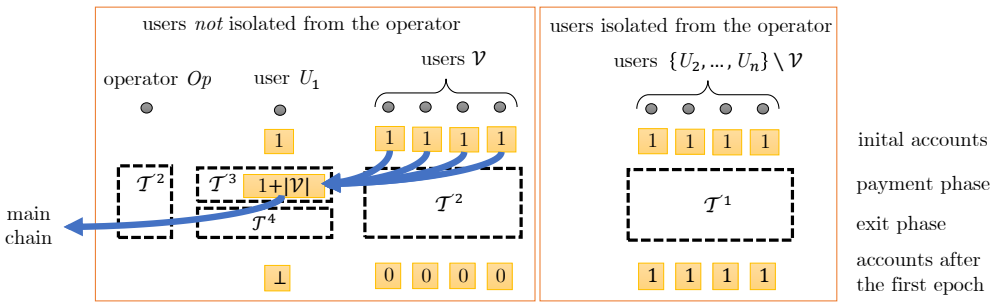


Figure 3.1: The first epoch of the experiment $\mathsf{Exp}(\mathcal{V})$. Gray circles denote the parties, and the $\mathcal{T}^i$'s denote the transcripts of the communication with Γ (see, e.g., Sect. 3.5.2 for their definitions).

Before discussing the second epoch of the experiment, let us note that in the first epoch the only way in which we deviate from the totally honest execution is the "isolation" of $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$. This will later allow us to be "flexible" and corrupt different sets of parties ($\{Op\}$ or $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$) depending on the results of our analysis of $\mathsf{Exp}(\mathcal{V})$. This will be different in the second epoch, where we always assume that parties from $\mathcal{V}$ are corrupt. This is ok because while constructing the attacks that satisfy (3.2) or (3.3) we will only use the first epoch of $\mathsf{Exp}(\mathcal{V})$. The only reason to have the second epoch of $\mathsf{Exp}(\mathcal{V})$ is to make sure that the users have to send large amounts of data to Γ during the first epoch, as otherwise corrupt $\mathcal{V}$ can steal the money (in the second epoch).

Let us now present some more details of the second epoch. Initially we corrupt all the users from $\mathcal{V}$ and "rewind" them to the state that they had at the beginning of the first epoch. This is done in order to let them "pretend" that they still have their coins. We then let all of them try to ("illegally") exit with these coins. Technically, "rewinding a user U" is done via a procedure denoted $\mathsf{Reconstruct}_U$. This procedure outputs the state that U would have at the end of the "payment" phase if she did not transfer her coins to U_1. To make it look consistent with the state of Γ, this procedure takes as input the transcripts defined above. Then, each user $U \in \mathcal{V}$ tries to exit (in the "exit" phase) from her state computed by $\mathsf{Reconstruct}_U$. Also the honest users try to exit (they receive an "exit" order from the environment). Let $\mathcal{Q}$ be the set of users that managed to exit with at least 1 coin. From the security of Plasma we get that $\mathcal{Q}$ is equal to the set of honest users ($\{U_2, \ldots, U_n\} \setminus \mathcal{V}$) plus a small (of size at most $\lceil \gamma n \rceil$) subset $\mathcal{D}$ of dishonest users.

The key observation is now that all that is needed to "simulate" the second epoch of $\mathsf{Exp}(\mathcal{V})$ are the transcripts $\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3$, and $\mathcal{T}^4$. On the other hand $\mathcal{V}$ can be approximately computed from $\mathcal{Q}$ (i.e., we can compute $\mathcal{V}$ with elements $\mathcal{D}$ missing, where $|\mathcal{D}| = \lceil \gamma n \rceil$). Hence the variable $(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)$ carries enough information to "approximately" describe $\mathcal{V}$. Thanks to this we can construct a "compression" algorithm that "compresses" a random $\mathcal{V} \leftarrow\!\!\$\ Y$ by simulating the first epoch of $\mathsf{Exp}(\mathcal{V})$ and obtaining $(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)$ and then "decompresses" it by simulating the second epoch, and computing the output as $\mathcal{V} := \{U_2, \ldots, U_n\} \setminus \mathcal{Q}$ (the additional $\lceil \gamma n \rceil$ elements can be

simply listed as an additional output of C and passed to D as input that has to be added to the output of D).

On the other hand, clearly (for completeness we show this fact in the full version of this paper [40]), a random $\mathcal{V} \leftarrow\!\!\$\ \mathrm{Y}$ with high probability cannot be compressed to a string that is significantly shorter than $\log |\mathrm{Y}| \geq n - 2$. This implies that with a noticeable probability $|(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)| \approx n - \lceil \gamma n \rceil \log_2 n$, where $\lceil \gamma n \rceil \log_2 n$ is the number of bits needed to describe set $\mathcal{D}$.

Obviously, the above fact implies that for at least one $i \in \{1, \ldots, 4\}$ we have that $\mathcal{T}^i \geq (n - \lceil \gamma n \rceil \log_2 n)/4$ with noticeable probability for concrete parameters). The rest of the proof of Thm. 1 is based on the case analysis of the implications of "$\mathcal{T}^i \geq n/4$" for different i's. More concretely, we show that in the first three cases ($i = 1, 2$, and 3) we can construct attacks that satisfy Eq. (3.2), and in case $i = 4$ — an attack that satisfies Eq. (3.3). All these attacks are based on the experiment $\mathsf{Exp}(\mathcal{V})$, but are only using its first epoch. In the proof we exploit the fact that the only malicious behavior that happens in this epoch is the "isolation" (i.e., not sending messages). Hence, we can use Observation 1 and "switch" between scenarios when different groups of parties are corrupt (while still getting the same transcripts $\mathcal{T}^i$). Moreover these attacks do not have uniquely attributable faults.

Remark. Our proof would also go through even if the total balance of the users a is arbitrarily large. The only difference would be that instead of giving 1 coin to every user, the environment would give to each user U_i (for $i > 1$) $\lfloor a/n \rfloor$ coins, and to user U_1 the environment would give the remaining coins (say). The rest of the proof would be essentially identical to the proof of Thm. 1.

Remark. Although the attack presented requires two epochs, the second epoch only captures the scenario where the underlying protocol is insecure and hence it can be seen as a "thought experiment". In other words, if the honest parties do not make large communication with the contact Γ in the first epoch, they risk loosing their coins in the second epoch. Therefore, under the assumption that the plasma system is indeed secure and consequently parties make large communication with Γ in the first epoch, the adversary cannot steal any coins in the second epoch and hence the second epoch would become obsolete.

CONCLUSION

The main contribution of this work is that we have shown that the distinction between Plasma Cash and Fungible Plasma is inherent, i.e., we ruled out the possibility of constructing Plasma that combines benefits of both Plasmas. We believe that, besides of the general scientific interest, our work (especially ruling out existence of some Plasma constructions) can help the practical blockchain community in developing Plasma protocols, and in general can bringing more understanding in what is possible and what is impossible in the area of off-chain protocols, and under what assumptions. It can also serve as a formal justification why "hybrid" approaches (such a "rollups") [29] may be

needed in real life. We also hope that this work may expand the scope of theory by identifying a new area where theoretical lower bounds can have direct impact on the real life problems.

4 A FORMALLY VERIFIED LIGHTNING NETWORK

AUTHORS

Grzegorz Fabianski, Rafal Stefanski, Orfeas Stefanos Thyfronitis Litos

4.1 ABSTRACT

In this work we use formal verification to prove that the Lightning Network (LN), the most prominent scaling technique for Bitcoin, always safeguards the funds of honest users. We provide a custom implementation of (a simplification of) LN, express the desired security goals and, for the first time, we provide a machine checkable proof that they are upheld under every scenario, all in an integrated fashion. We build our system using the Why3 platform.

4.2 INTRODUCTION

Bitcoin [79] is the oldest and consistently the highest valued blockchain. However, despite its value and prominence, it faces severe scalability issues [37] both in terms of throughput and of latency: it natively supports only up to 7 transactions per second and requires 1 hour from transaction submission to its verification.

To alleviate this limitation, the *Lightning Network* (LN) [86] has been developed. It lifts payments *off-chain*, i.e., it enables parties to securely pay each other without interacting with the blockchain. LN users enjoy latency and throughput limited only by the communication network between the transacting parties, avoid the hefty per-payment on-chain fees, and relieve other blockchain users from having to reach consensus on every single payment.

As of this writing, over $120M are contained in LN. This makes LN a security-critical application. Nevertheless, to the best of the authors' knowledge, its complex design[1] has never been formally verified for crucial properties such as safeguarding honest parties' funds. The current work aims to fill this gap. We employ the Why3 [23] formal verification platform to implement a subset of LN, formally specify the desired security properties and mechanically prove that our implementation adheres to them under all execution paths. Our system consists of 6060 lines of program code and 7420 lines of verification code.

1 https://github.com/lightning/bolts/blob/master/00-introduction.md

4.2.1 *Overview of LN*

The central construct of LN is the *payment channel*. Each channel has exactly two parties, each of which has some coins in it. One party can pay the other via an update of the channel state, which is done collaboratively with its counterparty. Each party is guaranteed that they can *unilaterally* close the channel on-chain and get their fair share of coins at any time. Channel updates are very cheap: the counterparties just need to exchange a few messages — no interaction with third parties or the blockchain is needed for paying within one's channel.

In slightly more detail, LN works as follows: To initially transfer coins from the blockchain to a new channel, the two parties collaboratively create a "joint account". This involves the two parties generating three Bitcoin transactions, with one submitted on-chain (the *funding transaction*) and the other two stored locally off-chain (the *commitment transactions*). The funding transaction transfers the initial coins to the joint account, whereas the commitment transactions safeguard each party's access to its coins: each party can publish on-chain one of the commitment transactions to move its coins back to the blockchain without needing the cooperation of its counterparty. This ensures that no trust between parties is needed. All in all, when published on-chain, the funding transaction opens the channel and the commitment transaction closes it.

On each payment, two things happen: Firstly a new pair of commitment transactions is generated, reflecting the balance of coins after the payment. Secondly, the previous pair of commitment transactions are *revoked*. A revoked commitment transaction corresponds to an old channel state and thus should never be published on-chain. A malicious party can nevertheless publish it — this could be beneficial if the older balance were in its favor. In such a case, the honest counterparty can publish the corresponding *revocation* transaction (which was generated when the offending commitment transaction was revoked) within a timeframe and punish its counterparty by confiscating all channel coins.

To sum up, a channel needs one on-chain transaction to open, one on-chain transaction to close and can support a practically unlimited number of off-chain payments. It offers low latency, high throughput, no fees and needs no extra trust requirements compared to the underlying blockchain.

The main focus of this work is the security of a single, two-party channel. Channel operations consist of funding a new channel, processing off-chain payments, and closing the channel in a way that the rightful funds are returned to the two parties on-chain. We do not model networks of multiple parties, nor the multi-hop payments supported by production implementations.

4.2.2 *Why3*

We implement and specify LN in Why3 [23], a framework for formally verifying high-level code. Why3 focuses on proof automation using SMT solvers. It is largely composed of two parts: Firstly, there is the WhyML language, used to express programs. It also supports expressing formal specifications in first-order logic in the form of inline assertions, separate lemmas, or function pre- and post-conditions; this allows for natural code constructs. Secondly, there is the *driver* which translates the verification conditions of WhyML code into SMT queries. The novelty of Why3 lies in the relatively seamless integration of a variety of SMT solvers. In the current work we have taken advantage of this feature by using both Alt-Ergo [35] and CVC4 [15].

Another important formal verification tool commonly used in the context of cryptographic protocols is EasyCrypt [17]. We note that we chose not to formalize LN using EasyCrypt because the latter is more geared towards low-level, probabilistic reasoning and computational assumptions — features that are used by most cryptographic protocols, but challenging for the general-purpose formal verification tools. In contrast, our approach assumes idealized low-level primitives (i.e., digital signatures) and proves security unconditionally (i.e., without a negligible probability of failure), therefore is largely incompatible with EasyCrypt. In fact, our approach of deterministically modeling signatures obviates the need for EasyCrypt, thereby simplifying our overall proof effort.

4.2.3 *Preliminaries on Cryptography*

LN relies on *digital signatures*, which consist of three algorithms: key generation, signing and verification. In our work, we adopt the *ideal functionality* defined in [32] to model these operations. Instead of running the algorithms, parties send queries to and receive outputs from the functionality, which internally keeps track of the signed messages and embodies the single source of truth, obviating the need for randomness and hard-to-model computational assumptions (see Subsec. 4.5.1). Since the functionality masks the need for such assumptions, we can strengthen our Why3 modeling of the adversary to an unbounded one. In fact, instead of modeling the adversary as a polynomially bounded Turing machine in the context of Why3, the security result proven by our code is against any sequence of choices made by the adversary, even non-computable ones. Although strengthening the adversary in this way might seem counterintuitive, it actually makes the verification effort simpler and clearer. The powers of such an adversary are directly expressed via *universal non-determinism*, which is readily available in Why3. We note that our high-level result (Def. 4.3.1) directly carries over to bounded adversaries. Therefore our definition also covers real-world systems which, due to their use of practical constructions of the digital signature algorithms, can only be secure against bounded adversaries.

4.2.4 *Our Contributions*

This work consists of the following contributions:

- We provide an implementation of the two-party LN interactions in Why3,
- We implement the subset of Bitcoin logic that is relevant to LN in Why3,
- We define *funds security* for an honest client in the presence of a Byzantine counterparty,
- We provide a machine-checkable proof of the funds security of LN channels,
- We demonstrate how to define *funds ownership* in the presence of multisig accounts and signed transactions stored off-chain,
- We identify crucial properties that any LN implementation should uphold to guarantee funds security and use them to modularize our proof. This provides a blueprint for proving the security of production LN implementations.

4.2.4.1 *Limitations.*

The main simplifications in our implementation compared to the real Lightning Network (LN) is that it only supports direct payments between two parties. As such it does not support multi-hop payments or HTLCs. Furthermore, we use a simplified model of Bitcoin which, in particular, does not include on-chain fees (see Section 4.4 for more details).

4.2.5 *Protocol Flow in Our Implementation*

Our work focuses on a single, two-party LN channel over Bitcoin. We implement a fragment of the LN functionality, namely direct payments between two parties. We do not include HTLCs and multi-hop payments in our implementation.

The operation of our channel closely follows that of production LN implementations. When one of the two parties (a.k.a. the *funder*) is instructed to open a new channel, the funding process begins. The funder notifies its counterparty (a.k.a. the *fundee*) and the two parties engage in a series of messages, at the end of which the channel is established. All initial funds belong to the funder and they originate from the funder's on-chain bitcoins.

Once a channel has been established, a payment can be initiated by either side. The payer specifies the amount of in-channel funds to transfer, which is subtracted from the payer's channel balance. After a few messages, the payee's balance is increased by the payment sum. Channels support an arbitrary number of sequential payments.

At any moment, either party can initiate the closing procedure. An honest party also closes its channel if its counterparty misbehaves (e.g.,

stops responding). An honest closing party is guaranteed to receive its channel balance on-chain after bounded delay. Internally, channel closure is initiated by submitting a single transaction on-chain that reflects the channel balance. This starts a timer, during which the counterparty can dispute the claimed channel balance if the closing party submitted an outdated channel state. Honest client operation guarantees that the honest party loses no funds due to failed disputes.

4.2.6 *Security Goals*

We consider two standard properties of LN: The first one, which we call *funds security*, states that a party can always close the channel and transfer its rightful funds on-chain within a bounded time frame. For the sake of funds security, we assume that the honest party interacts with a malicious, byzantine party that also controls the environment. The second property called *liveness* states that channel funding and channel payments between two honest parties will always complete in a bounded amount of time. For the sake of liveness, we assume that two parties are honest, but the adversary controls the network and payment requests (see Subsec. 4.2.7 for more details).

In this work, we only prove funds security, not liveness. In fact, we are aware that in a scenario when the two honest parties try to initiate payments simultaneously in both directions, our implementation might deadlock. (This does not violate the guarantee of funds security, as the deadlock can be resolved by either party closing the channel). Nonetheless, we do test that a simple channel opening and payment scenario completes successfully (see `twoHonestParties.mlw`). As the violation of funds security is far more detrimental than the violation of liveness, we believe that such an approach strikes a reasonable balance between proof complexity and practical relevance.

4.2.7 *Adversarial Model*

In our model, funds security is formalized via a 1-player game, following established cryptographic practice. The game is played by the adversary. Different moves in the game correspond to different actions that the adversary can carry out in real life. For example, it can choose to deliver a message to the ledger, or to have the corrupted channel party send arbitrary messages to its honest counterparty. These two moves correspond to the power to control the network and to corrupt one channel party respectively. Its control over the corrupted party is complete: it can send and sign arbitrary messages on its behalf at any point in time. On the other hand, its control over the network is encumbered with the responsibility to deliver transactions to the ledger within a specific time bound. Furthermore, it controls the moments in which the honest party is activated, but it must ensure that the delay between two activations does not exceed a specific time bound. These limitations correspond to the standard, reasonable network assumptions and the security of LN depends on them.

Last but not least, we give the adversary the power to choose the moment of channel opening and closure, as well as the timings, directions and amounts of payments while the channel is open. These additional adversarial powers correspond to the choices that the honest party can make in a practical deployment. We choose this approach as it models the worst-case scenario, making our results stronger as well as conformant with the standard approach in cryptography. Allowing these choices to be made by the adversary has the added benefit of ensuring that the game is strictly 1-player, keeping the model simpler. The adversary wins if a channel closure request is put forward but, after the aforementioned time bound, the honest party has not received its rightful funds on-chain.

4.2.8 *Architecture Overview*

We now discuss our code architecture at a high level — Appx. A of the full version [45] contains a relevant diagram. We provide a simplified implementation of Bitcoin called Γ, which focuses on the capabilities related to LN (Sec. 4.4 and `gamma.mlw`[2]). Furthermore, we implement the signature functionality [32]. It is used by the honest party, the adversary, and Γ to sign messages and verify signatures (Subsec. 4.2.3 and `signaturesFunctionality.mlw`). Our LN client can read the state of Γ and send transactions to it via a network queue controlled by the adversary. The honest client receives instructions to perform actions, such as to open a channel or carry out a payment. These instructions come from the adversary, but invalid instructions are ignored by the honest client (`honestPartyType.mlw` and `honestPartyInteractions.mlw`). Last but not least, we formally define funds security, as discussed in Sec. 4.3 (`honestPartyVsAdversary.mlw`).

In order to keep the proof tidy in the face of the complex interactions between the aforementioned components, we use a number of techniques (Sec. 4.5). We here highlight the two most salient ones. The LN client discussed above has to keep track of a complex state which includes many low-level details, such as which messages remain to be sent in order to conclude an in-flight payment. In order to keep these details separate from the proof, we identify a simple set of requirements on the client behavior (see Subsec. 4.5.2, Appx. C of the full version [45], and `honestPartyInterface.mlw`). We can thus separate our proof into two independent parts: We first prove that our LN implementation meets these requirements and we then prove that any implementation that satisfies these requirements enjoys funds security. We believe that our requirements must be satisfied by any secure LN implementation, therefore a proof of security of an existing production LN deployment boils down to just proving that it conforms to our requirements.

The second proof technique elucidates the meaning of coin ownership. Albeit a useful high-level concept, is not straightforward to

2 The full code can be found at https://anonymous.4open.science/r/LightingInWhy3-DD1C/ — `README.md` contains relevant documentation.

define due to signed but non-processed transactions and multisig accounts. To that end, we define the `Evaluator` (Subsec. 4.5.4, Appx. B of the full version [45], and module `Evaluator` in `gamma.mlw`) which formally answers the question "How much money does a party own?" This abstraction greatly simplifies the proof effort. We believe this approach can be reused in similar verification projects.

4.2.9 *Related Work*

4.2.9.1 *Formal verification of blockchain infrastructure and applications.*

Existing blockchain-related formal verification efforts revolve around two axes: Verifying consensus protocols and verifying smart contracts.

Formal verification of consensus protocols has been carried out in [65] and [24]. Similarly to both of these protocol analyses, the current work expresses the execution of multiple parties that communicate via the network, the delays of which are explicitly modeled. Moreover, similarly to [65] and [24], we verify end-to-end guarantees of the execution, not just static invariants.

Regarding the verification of smart contracts, a survey of tools for Ethereum smart contract analysis, including formal verification tools, can be found in [7]. Concrete examples are Manticore [77], EthVer [70], PRISM [66], and [57, 82, 101]. These tools focus on specific contract properties, whereas our work takes a whole-system approach, whereby the smart contract logic (i.e., the LN transactions) only form one part of the entire system, the other parts being the explicit modeling of parties, the communication network they use, and the process of signing transactions.

4.2.9.2 *Security analyses of LN.*

A survey of results on LN security and attacks against it can be found in [93]. LN security has been formally modeled and proven in the UC setting in [62]. Most comparable to our work is [53], which builds upon [52] in an effort to formally verify the security of LN using TLA+.

Both [62] and [53] model the intricate details of funds security in the multi-hop payment setting, while our work focuses on a single channel. Due to the lack of multi-hop payments, the wormhole and griefing attacks [93] do not apply in our work. With respect to funds security, the main difference between our work and [53, 62] is the framework used. Any small differences in the exact security guarantees result from framework differences, not from high-level security goals.

We now present a detailed comparison of our work with [53], starting with the similarities. Both works prove security for a specification of LN, as opposed to a specific implementation. Both approaches model signatures as ideal functionalities, removing their inherent randomness (the latter is problematic in the context of formal verification) in the process. Last but not least, in both cases non-determinism is

used to model the adversary, including its exact leeway for arbitrary behavior.

On the other hand, the current work differs from [53] in the following respects: We model security using a game-based definition, proving specific security properties, whereas [53] uses a simplified version of UC-based modeling. We use the Why3 first-order proof system, which allows us to model infinite system states, whereas [53] uses the TLA+ model checker and only checks all possible scenarios up to a specific number of actions. Our modeling is limited to simple payments within a single channel, whereas [53] further models multi-hop payments over multiple channels. We provide a custom implementation of LN, upon which we base our specification of LN, which happens to be more abstract than the official specification (BOLT), whereas [53] verifies the BOLT specification itself. Last but not least, our formal verification effort is complete, whereas [53] constitutes an intermediate report of a proof effort that is still underway.

4.3 SECURITY MODEL

The main security goal of this work is to guarantee that a Lightning party that honestly follows the protocol never loses money. In particular, an honest Lightning party that has sent and received a number of payments in the channel should be able to redeem on-chain its initial channel funds plus any funds received minus any funds sent within a bounded time after requesting channel closure. We call this *funds security*. Formally it is a property over 5 parameters:

- The honest party LN protocol `HonestParty`,
- The ledger protocol Γ,
- The party activation window `deltaWake` $\in \mathbb{N}$,
- The ledger delivery window `deltaNet` $\in \mathbb{N}$.
- The time needed for a channel to close `channelClosingTime` $\in \mathbb{N}$.

We define funds security in terms of a game, played between the adversary and the honest party. If the adversary cannot win no matter their actions, then funds security is guaranteed. This game is defined in the `honestPartyVsAdversary.mlw` file.

The game state consists of the following elements: the honest party state, the ledger state, the signature functionality state, and the collection of timestamped, pending ledger messages. It also tracks 4 variables: the last time the honest party was woken (`honestPartyLastWoken` $\in \mathbb{N}$), whether and when was the honest party first ordered to close its channel (`closeOrderTime` $\in \mathbb{N} \cup \{\bot\}$), the current time (`time` $\in \mathbb{N}$), and the expected funds of the honest party (`expectedAmount` $\in \mathbb{N}$).

Since channels are two-party constructions, the other party is presumed corrupted. We define an initial system state in which the ledger

contains some coins for each of the two parties and the channel is not yet opened.

We next describe the types of moves that the adversary can make. All actions apart from `IncrementTime` are parametrized by additional arguments that the adversary chooses along with the move type, as explained below.

1. `SignMsg`: This message enables the adversary to sign any number of messages on behalf of the corrupted party. It also models signature malleability by allowing the adversary to create new signatures for messages already signed by the honest party (see Subsec. *Allowing public modification of signatures* in [32]).

2. `SendMsgToGamma`: The adversary generates an arbitrary message which is immediately processed by Γ. This models the ability of a *rushing* adversary to add transactions to the blockchain at any time, "skiping the queue". As we mentioned, Γ is a parameter of the game. Upon receiving a message, it reads the current time, verifies any signatures with the signature functionality, and returns its updated state.

3. `DeliverMsgToGamma`: This action gives the adversary control of the delivery of messages from `HonestParty` to Γ. The adversary chooses a message from the collection of pending messages. The message is removed from the collection and is immediately processed by Γ.

4. `IncrementTime`: This action increases the current system time by 1 unit. This action is only available if (i) there are no pending messages older than $\texttt{deltaNet} - 1$ and (ii) `HonestParty` has been activated within the last $\texttt{deltaWake} - 1$ rounds (i.e., $\texttt{time} - \texttt{honestPartyLastWoken} < \texttt{deltaWake}$).

5. `SendMsgToParty`: This action lets the adversary interact with `HonestParty` on behalf of the corrupted one. The adversary chooses a message to be delivered to `HonestParty`, which the latter handles according to its implementation. The result of the `HonestParty` activation is accounted for as follows. The messages that `HonestParty` wants to send to Γ are added to the pending messages collection, `honestPartyLastWoken` is updated to `time`, and `expectedAmount` is updated if `HonestParty` has acknowledged receipt of a payment. During handling of the message, `HonestParty` may interact with the signature functionality. The exact format of messages that `HonestParty` expects depends on the concrete LN party protocol, which as mentioned before is a system parameter.

6. `ControlEnvironment`[3]: This action lets the adversary interact with `HonestParty` on behalf of the system. This action is handled by `HonestParty` in exactly the same way as adversarial

3 Actually, in the implementation both `SendMsgToParty` and `ControlEnvironment` are handled by `SendMsgToParty`, with appropriate arguments to distinguish between the two. We separate them here for clarity.

instructions (i.e., `SendMsgToParty`) are. The only difference is that `ControlEnvironment` additionally may cause the system state to be directly updated. Here is an exhaustive list of all possible actions the system can prompt `HonestParty` to perform:

a) `EnvOpenChannel`: Orders `HonestParty` to initiate opening a channel with the counterparty.

b) `TransferOnChain`: Orders `HonestParty` to transfer its funds using an on-chain transaction, i.e., by spending coins from a public-key account. It fulfills two functions: (i) funding of a new channel and (ii) direct on-chain payment to the counterparty[4]. The system then decreases the party's `expectedAmount` accordingly.

c) `TransferOnChannel`: Similar to above, but using an already open channel. Once again, the party's `expectedAmount` is decreased.

d) `CloseNow`: Orders `HonestParty` to initiate the closing procedure of a previously opened channel. The variable `closeOrderTime`is set to `time`.

e) `JustCheckGamma`: Wakes up `HonestParty` to give it the chance to do any recurrent bookkeeping. This is needed to satisfy the periodic `HonestParty` wake-up requirement, even when there is no concrete instruction for the party.

Those actions ensure that the interactions between the adversary and the honest party are modeled accurately. In particular, `expectedAmount` is tracked by the experiment (and not simply reported by the honest party), which guarantees that the value of `expectedAmount` matches its intuitive meaning: it decreases whenever the party is asked to pay the counterparty and increases every time the party acknowledges receiving a payment.

Last but not least, let us define the adversary's winning condition. It is also formally defined as `adversaryWinningState` in `honestPartyVsAdversary.mlw`.

Definition 4.3.1 (Funds Security)**.** The winning state for an adversary is one that satisfies the following:

1. $\texttt{closeOrderTime} \neq \bot$,
2. $\texttt{time} \geq \texttt{closeOrderTime} + \texttt{channelClosingTime}$,
3. The honest party has less coins than `expectedAmount` on-chain (i.e., not counting those in the channel), as output by Γ when its function `immediateAmountOnChain` is called.

We say that a system parametrized with the honest party LN protocol `HonestParty`, the ledger protocol Γ, `deltaWake`, `deltaNet`, and `channelClosingTime` achieves *funds security* if no adversary can reach the winning state, no matter which sequence of actions it follows.

4 Our work only needs the former function of `TransferOnChain` — still, the latter makes our model more complete for essentially no added code complexity.

Realistic LN implementations use practical constructions of digital signatures, therefore they can only achieve Funds Security against *probabilistic polynomial-time* (PPT) adversaries. As discussed earlier however, we model digital signatures as an ideal functionality, the security of which cannot be broken even by unbounded or non-computable adversaries. In the context of Why3 we model the adversary as unbounded for simplicity, but our ultimate security guarantee is with respect to practical digital signature constructions and therefore against bounded, PPT adversaries. Weakening the adversary from unbounded to PPT strictly shrinks the set of admissible adversaries, therefore the security guarantee obtained by the Why3 model is directly transferable to the bounded adversarial setting.

Let `channelTimelock` be the timelock after which a party can reclaim its funds from a commitment transaction. `channelTimelock` is a parameter of the `HonestParty` protocol. For any `deltaWake`, `deltaNet`, $\texttt{channelClosingTime} \in \mathbb{N} : \texttt{channelClosingTime} \geq 3 \cdot \texttt{deltaNet} + 2 \cdot \texttt{deltaWake} + \texttt{channelTimelock} + 1$ and $\texttt{channelTimelock} \geq \texttt{deltaWake} + \texttt{deltaNet} + 1$, we provide a concrete implementation of Γ that models a subset of the functionality of Bitcoin (Sec. 4.4), as well as an implementation of a subset of LN (Subsec. 4.2.5) such that funds security is guaranteed, as we formally verify (the main security result is lemma `honestPartyWins` in `honestPartyVsAdversary.mlw`).

4.4 MODELING BITCOIN

We use a high-level modeling of the ledger called Γ which captures the fragment of Bitcoin that is relevant to Lightning channels. In this section, we give a brief introduction to Bitcoin operation and then explain how we simplify it in order to express just what is necessary for the protocol.

In practical implementations, Bitcoin transactions are organized in a *directed acyclic graph* (DAG). Each transaction is a node with at least one *input* and one *output*. Each input is connected to exactly one output and each output to at most one input. Each output specifies the number of coins it contains. At any point in time, the DAG has some outputs that are not connected to any input: these are the *unspent transaction outputs* (UTXOs) and model all available coins. A new transaction can be added to the DAG if all its inputs are connected to UTXOs and the sum of the coins of its outputs are equal to the sum of the coins of the outputs that its inputs spend — this guarantees new coins are not created out of thin air. Furthermore, outputs contain *Script* that specify spending semantics. Script is a simple non-Turing-complete language with a limited expressiveness, including signature verification, hash checking and time checking. Each output locks its coins by specifying a Script statement. It can later be unlocked by providing an input with a corresponding witness. When a valid transaction is added to the DAG, the outputs it spends are not UTXOs anymore and its own outputs become UTXOs.

Our modeling keeps track of the set of transaction outputs, as well as whether they are unspent. It forgoes the DAG and the linearization of the transaction history. We limit our attention to a fixed set of scripts, enough to model all possible Lightning interactions:

- The public key account, known as "pay-to-public-key-hash" (P2PKH) in Bitcoin implementations[5],
- The 2-out-of-2 multisig — this is used to store the coins of an open Lightning channel during normal operation,
- The commitment transaction conditional output — this is used to unilaterally close an open channel.

We elaborate on the transitions associated with each script in the full version [45] and in our implementation (`gamma.mlw`).

Bitcoin has no official specification — this role is instead fulfilled by the reference implementation. A formal model for Bitcoin is presented in [10]. In the current work we intentionally avoid modeling parts of Bitcoin execution that are irrelevant to the Lightning participants. More specifically, we ignore all public keys other than those of the two channel parties and any script other than the aforementioned ones. We are confident that our simplified modeling is applicable to a full model of all Bitcoin transitions [10] when non-Lightning transactions are filtered out. We however leave proof of that our model is an abstract interpretation of [10] as future work.

For simplicity, we do not model on-chain fees in our work (this is in line with [10]). However, we believe that extending our work to include fees is conceptually straightforward. See Section 4.6 for more details.

4.5 PROOF STRATEGY

LN consists of multiple moving parts that are interconnected in intricate ways. In order to manage the complexity of these interactions, we adhere to the following principle. The system does not need to maintain the entire history of the past states, neither for the honest LN party nor for Γ. All useful knowledge on past states is instead compressed into constant-size invariants. The alternative would be to keep track of the entire execution history, which would be more aligned with human reasoning about distributed systems. Unfortunately, quantification over time complicates formally verified proofs, as it would require a number of interconnected inductive proofs. Our approach is more natural in the context of Hoare logic, as we adhere to a single induction step per protocol step.

5 Similarly to [32], our modeling foregoes the use of public keys, opting for destination identifiers instead — this simplifies the model by avoiding the explicit formalization of the mechanics of cryptographic signatures.

4.5.1 *Digital Signatures Ideal Functionality*

From a cryptographic perspective, the security of any digital signatures construction is only *computational*, i.e., it can be broken by a sufficiently powerful adversary. As a standard, cryptographic literature only considers PPT adversaries. This is normally necessary because of the underlying computational assumptions, which are breakable by an unbounded adversary. In fact, replacing the aforementioned ideal functionality with a real signature scheme is only secure against such PPT adversaries. Unfortunately, quantifying over all PPT machines has historically proven exceedingly difficult [68] in the context of formal verification. We would thus like to categorically rule out the exponentially small probability of a successful forgery in a principled manner. We achieve this in two steps: Firstly, we allow the adversary in Why3 to be any function (even non-computable) using the `any` keyword of Why3. We then replace the signatures construction with an *idealized* digital signatures functionality (`signaturesFunctionality.mlw`) which never permits forgeries (not even with *negligible* probability), as defined in [32]. At a high level, this functionality plays the role of a "trusted signatures server" that is common for all parties. Parties request the signing and verification of messages. The functionality stores the message-party pair on signature request and responds whether the given message-party pair has been stored on verification request. Crucially, the functionality is incorruptible, making forgeries impossible even by a non-computable adversary.

A salient question is: why is the replacement of the construction with the functionality valid? The answer, coming from the *simulation-based cryptographic paradigm*, employs an *indistinguishability argument*, according to which *every* external PPT observer is unable to distinguish between interactions with the construction versus the functionality, except with negligible probability. Indeed, our ultimate security guarantee is provided with respect to a realistic signatures scheme (not an ideal functionality) and a PPT adversary (not an unbounded one). Using a non-computable adversary is merely a proof technique, necessary to tractably model the adversarial behavior in Why3. The indistinguishability argument enables us to transfer the ideal-world security property (formally verified by Why3) to the real world of practical digital signatures and PPT adversaries.

4.5.2 *Encapsulation of Implementation-aware Properties*

Our work includes a novel implementation of a subset of the LN mechanism, an implementation of Bitcoin on which our LN implementation relies, as well as a formally verified proof of the *funds security* of our implementation. We chose to organize our proof so as to clearly separate the particularities of our custom LN implementation on the one hand and the general logic that every secure LN implementation must uphold on the other. Our proof is therefore separated into three parts: an implementation-specific part (`honestPartyType.mlw`

and `honestPartyImplementation.mlw`) which is aware of the complexities and details of our LN implementation, an abstract specification which encapsulates a set of simpler invariants that any secure LN implementation must uphold (`partyInterface.mlw`), and an implementation-agnostic part of the proof that depends only on the specification (`honestPartyVsAdversary.mlw`).

Our abstract specification focuses on the high-level properties that clients have to satisfy in order to uphold security, such as the need to publish on-chain a known revocation transaction as soon as it becomes valid. This is very different from BOLT, which specifies the required behavior of clients, (e.g., reactions to specific messages), but does not stipulate logical properties. For more details on the interface, see Appx. C of the full version [45] or `partyInterface.mlw`.

4.5.3 *goodSimpleParty*

As our `HonestParty` LN protocol implementation progressed, it became increasingly complex and highly coupled with the rest of the system. Changes to low-level details of `HonestParty` LN protocol would ripple to the invariants and from there to the rest of the system. To mitigate this issue, we decided to introduce a simple, implementation-agnostic invariant that captures the essence of the properties that any secure LN implementation should satisfy (`goodSimpleParty` in `partyInterface.mlw`). This invariant is aware of only a subset of the information held by the `HonestParty` implementation, such as the received revocations and the commitment transaction that can be used to close the channel. This is the data that any reasonable LN implementation should be able to produce.

Unfortunately, one cannot prove that the `goodSimpleParty` invariant is preserved by the `HonestParty` transitions without additional dependencies. For this reason we also define a stronger, implementation-aware invariant (`partyInvariant` in `honestPartyType.mlw`) that implies `goodSimpleParty`. This invariant is visible only to `HonestParty`, not to the rest of the system. `partyInvariant` is more complex than `goodSimpleParty`: the former spans approximately 65 lines of code, whereas the latter just around 15 lines.

4.5.4 *Funds Ownership*

In this subsection we discuss what it means for an `HonestParty` to own coins in Γ. This is a subtle point, since, from the perspective of the chain, all channel funds are locked behind a 2-of-2 multisig, which usually does not imply exclusive ownership. However, an honest channel party should be able to consider some of these funds as its own. As a further example of funds ownership complications, consider an on-chain output controlled by a single public key *pk* and a signed, valid, published transaction that spends this output but is not (yet) part of the chain. These funds should not be considered as owned by the holder of the private key of *pk*. Our approach to resolving such

ambiguities to ownership is to consider as owned those coins that a party can reliably and unilaterally move to its public key account — this way the tension in both of these scenarios vanishes.

To formalize this concept of ownership, consider a specific state of the Bitcoin ledger. Let an *Extractor* be a party which has some local state (e.g., private keys, signatures received by others) and can communicate with the ledger. We also consider an *Obstructor* with access to the ledger and the ability to take any action of the adversary, as described in Sec. 4.3. The Extractor has a single ID with respect to the signature functionality, whereas the Obstructor controls all other IDs. The Extractor's goal is to maximize the funds it can extract from the ledger into its private account(s), whereas the Obstructor tries to minimize them. Let *extractable value* be the value of such a game when both parties play optimally. We can now define the value owned by a given party at any instance of the execution of some Bitcoin-based protocol by identifying this party with the Extractor (thus allowing the party to diverge from the protocol) and the rest of the parties (along with the adversary) with the Obstructor. We then say that the funds owned by the party are equal to the extractable value. Observe that this definition of ownership does not depend on any specific protocol thus it can be used in settings other than LN.

Next, we discuss how the concept of ownership is leveraged in our proof of funds security. We say that the `HonestParty` has *solvency* at some point during the execution of the funds security game if the funds it owns are at least equal to `expectedAmount` (Sec. 4.3). Solvency has the intuitively appealing property that it refers to any given moment *during* the execution, as opposed to funds security which only concerns the *final state*. It is not hard to see that solvency implies funds security when the channel is closed. More precisely, funds security follows from solvency and the fact that the `HonestParty` can close the channel in bounded time. The proof of the latter fact is relatively straightforward (it is formalized as lemma `closingWorks` in `honestPartyVsAdversary.mlw`). The solvency part of the proof is more involved and it is discussed in Subsec. 4.5.5.

Finally, let us point out an important technical detail: Although the game-based definition of ownership is, in our opinion, both intuitive and theoretically appealing, it is also challenging to compute and reason about, as it involves evaluating an arbitrary min-max tree. Therefore, in our proof, we replace funds ownership with a more straightforward (i.e., free of min-max trees) but less intuitive `Evaluator` function (implemented as `partyExpectationsFull` in `gamma.mlw`), which provides a pessimistic under-approximation of the funds owned by the party. For further details, see Appx. B of the full version [45]. Similarly, we replace solvency with *evaluator solvency*, which states that the `Evaluator` never drops below `expectedAmount` during the execution of the protocol.

4.5.5 *Evaluator Solvency Overview*

As discussed in Subsec. 4.5.4, a central point of our effort is proving the preservation of evaluator solvency across state transitions. In this section we focus on the crux of the proof, which lies in the transitions of Γ and the `HonestParty`.

For the transitions of Γ, it is sufficient to prove that the `Evaluator` does not decrease (as `expectedAmount` does not change during such transitions). This is not hard to prove as the `Evaluator` mirrors the crucial aspects of funds ownership which, by definition, does not decrease during transitions of Γ. The proof is formalized in `gamma.mlw`.

The transitions of the `HonestParty` are harder to handle. To help with the proof, we define a relation `goodTransition`, which connects the states of the `simplePartyT` before and after the transition, as well as the messages sent by the party to Γ during the transition. This relation abstracts the properties that any LN implementation must guarantee to ensure evaluator solvency. In fact, it is agnostic of the exact implementation of `HonestParty`, which allows us to separate the proof from the implementation details.

4.6 CONCLUSION & FUTURE WORK

In this work we successfully modeled Lightning [86] channels using first-order logic in Why3 and proved funds security for honest channel parties. First-order logic allowed us to express naturally the security property and functionality of Lightning channels, as well as the low-level invariants encountered during formalization. During the proof effort, we realized that an in-depth understanding of funds ownership was necessary, which led us to define it robustly and formally. We believe that this approach can be reused in similar efforts in the future. Another relevant outcome is the `partyInterface` design, which was the result of a number of modularization attempts. It cleanly separates LN party implementation details with the interface it should provide, simplifying the proof.

There are a number of future directions for strengthening our modelling and architecture. To begin with, we believe that our work could be extended to two-party channels supporting HTLCs with a reasonable amount of effort. This will be a significant stepping stone towards formalizing the security of real-world LN implementations. Moreover, our modeling of Bitcoin can be improved to bring it to parity with already existing formalizations such as [10]. We expect that this task also needs a reasonable effort, but could face more unexpected roadblocks than adding HTLCs.

Another way of improving our Bitcoin model is to include transaction fees, which, for simplicity, we currently do not model. This would require the honest parties to set aside some on-chain funds to pay any fees that may arise. These funds would not be transferable by the `TransferOnChain` environment order. Thanks to the small maximum number of per-channel on-chain transactions (at most 3) and given

a (weak) assumption of a maximum per-transaction fee, the amount to set aside is fixed. Thus, funds security could also be proved in the presence of transaction fees.

Last but not least, a more ambitious project would be to extend the security analysis to multi-hop payments over a network of LN channels. To that end, new ideas and insights would be required.

Acknowledgements.

This work has been partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666). Furthermore, this work was partly supported by the German Research Foundation (DFG) via the DFG CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Disclosure of Interests.

The authors have no competing interests to declare that are relevant to the content of this article.

5

COMPUTATIONALLY SOUND SYMBOLIC CRYPTOGRAPHY IN LEAN

AUTHORS

Stefan Dziembowski, Grzegorz Fabiański, Daniele Micciancio, Rafal Stefański

5.1 ABSTRACT

We present a formally-verified (in Lean 4) framework for translating *symbolic* cryptographic proofs into *computational* ones. Symbolic cryptography is a well-established approach to support reasoning about protocols' security in an abstract way, and it is relatively easy to verify using proof assistants. However, it does not capture the full power of realistic computational adversaries that may not respect the symbolic abstraction. Computational cryptography, on the other hand, provides security guarantees against these more powerful and realistic adversaries, but at the cost of security proofs that are more complex to come up with and much more challenging to formally verify.

We deal with this challenge by formalizing the translation from symbolic to computational cryptography in Lean 4. Our framework is based on the work of Micciancio [73], which suggested the use use of co-induction (instead of induction) for reasoning symbolically about the adversary's knowledge in a way that is tightly connected to computational cryptography. Our contributions include (1) the formalization of the symbolic cryptography framework, (2) the formalization of the computationally sound cryptography framework, and (3) the formalization of the translation between the two. Building on the work of Li and Micciancio [67], we also provide (4) an extended example that applies our framework to circuit garbling, a well-known cryptographic protocol frequently used in secure multi-party computation.

We believe that our work will serve as a foundation for future research in the area of formal verification of cryptographic protocols, as it enables reasoning about cryptographic protocols more abstractly while still providing a formally verified connection to the computational aspects of real-world cryptography.

5.2 INTRODUCTION

Indistinguishability [50, 51] is a core concept in cryptography, capturing the fact that many important cryptographic primitives (e.g., symmetric and public-key encryption, commitment schemes, zero-knowledge proof systems) should conceal any partial information about their input. In a nutshell, indistinguishability asserts that two objects (such

as probability distributions over bitstrings) cannot be told apart with non-negligible probabilistic advantage by any adversary with bounded computational power. Traditionally, such adversaries are modeled as probabilistic polynomial-time (PPT) interactive Turing machines, and the "non-negligible advantage" is defined as a function that decreases faster than any polynomial in the size of the input. Unfortunately, formalized reasoning using proof assistants is notoriously difficult in these settings. For this reason, even specialized proof assistants targeted at security protocols (like Proverif [22] and Tamarin [72]) adopt an idealized model of cryptography, and analyze security in a symbolic execution model that abstracts away the computational and probabilistic details of concrete implementations. Unfortunately, this provides limited security guarantees, which hold only against adversaries that respect the symbolic abstraction. In practice, security should hold when applications are run in the presence of adversaries (even those not known at protocol design time) that deliberately try to subvert the system, and are not bound by the rules of the symbolic execution model. So, while symbolic analysis can be helpful to check correctness properties, and detect certain types of attacks, it falls short of providing the security guarantees expected by cryptographers.

Other formal analysis tools (like EasyCrypt [16, 17] CryptoVerif [21], and Squirrel [11]) bridge this gap by expressing computational security definitions and cryptographic assumptions, within a formal, machine-checked environment. A clear advantage of this method is its generality and flexibility. Supporting the definition of fairly general security assumptions and properties, these frameworks allow for the expression and analysis of a wide range of cryptographic applications. However, while the use of a formal framework and mechanized proof verification provides high security assurance, the process of analyzing the security of a protocol remains relatively complex, even harder than manual security proofs.

A third approach, and the focus of this paper, is based on *computationally sound symbolic analysis*, and aims to offer the best of both worlds: simple proofs of indistinguishability based on symbolic manipulation, while at the same time offering strong security guarantees against computational adversaries. The approach was pioneered by Abadi and Rogaway [1], who described a simple symbolic syntax for cryptographic expressions with the remarkable property that when two expressions are symbolically equivalent (i.e. as shown by simple symbolic manipulation), their computational interpretations (obtained by implementing and evaluating the expressions using standard algorithms for encryption and other cryptographic primitives) are indistinguishable by any computationally bounded algorithm. Since then, the viability of the approach has been investigated in several follow-up papers, including extensions with other cryptographic primitives [73], and various applications to key distribution protocols [74, 75, 81], access control of XML databases [2], password guessing attacks [18], and even the construction of *Garbled Circuits* [67], a general technique for secure two-party computation [69, 102]. However, the approach so far has received considerably less attention than general-purpose

symbolic and computational security analysis tools, especially when it comes to formalized and automated reasoning (one exception is [36] which briefly describes a formalization of the original Abadi-Rogaway framework in Coq). Finally, it might be worth cautioning the reader that this symbolic approach is very different from the symbolic execution models used in tools like Proverif and Tamarin, as it does not rely on equational theories or execution traces. In this paper whenever we mention the symbolic approach or symbolic equivalence, we refer to the computationally sound symbolic cryptography pioneered by Abadi and Rogaway [1].

5.2.1 *Our Contribution*

One of the central goals of computationally sound symbolic cryptography is to provide tools for simple yet rigorous verification of cryptographic protocols. The existing literature has developed the theoretical foundations necessary to achieve this goal; however, the tools and frameworks to apply these results for formal verification are still lacking. In particular, prior work were mostly focused on testing-based approaches that could not provide full guarantees.

The main contribution of this work is to close this gap: we show how to turn the theoretical foundations of symbolic cryptography into a usable, formal framework, implemented and mechanically verified in Lean 4. Here is the high-level summary of our contributions:

1. We formalize the symbolic expressions of [67, 73] in Lean.[1]
2. We formally prove a computational soundness theorem (similar to [73, Theorem 1]), which states that symbolic security guarantees imply computational security.
3. As a case study for our framework, we formalize the symbolic proof of security of the garbling procedure of [67]. By combining this with the computational soundness theorem, we obtain a mechanized proof of the security of garbled circuits in the computational model, which is a non-trivial result (although this is not the first mechanized proof of security of garbled circuits, see Section 5.2.2). It is worth noting that thanks to the use of the symbolic framework, our proof is very intuitive.

We stress that our results go beyond a straightforward transcription of existing proofs. In several cases, direct formalization would have been infeasible, and we had to rethink the definitions and reconstruct the arguments at a much deeper level of detail. Below, we present a low-level list of our contributions:

1. We simplify the symbolic construction for circuit garbling from [67]: the original symbolic construction used pseudorandom generators (PRGs), which are not usually used in garbling schemes. In our simplified construction, we remove the PRGs.

1 A similar formalization had already been given in [67, 73], but in a far less expressive functional language (Haskell) which does not support the reasoning capabilities required in this work. See Section 5.2.2 for details.

2. We develop a new, simpler symbolic-equivalence argument: the original proof from [67] would have been significantly harder to formalize directly.

3. We introduce the *pseudo-fixpoint technique* for reasoning about symbolic expressions. This tool turns out to be essential for handling recursive structures in our proof.

4. We extend the Lean's VCVio library [95] with *seeded oracles*, replacing stateful oracles by a once-and-for-all randomized environment, simplifying the formalization effort.

5. We formalize computational indistinguishability in Lean for both distributions and seeded oracles. This requires explicitly lifting the existing definitions to families indexed by the security parameter.

6. We develop a semi-formal method to reason about polynomial-time computations, which allows to combine formal run-time analysis with traditional (pen-and-paper, informal) complexity analyses of specific reductions[2].

Let us also comment on our choice of the platform for our formalization. We use Lean 4 [78] — a general-purpose theorem prover — to formalize computationally sound symbolic security proofs of complex protocols, demonstrating that these tools are particularly well suited for the task. The use of general-purpose theorem provers has multiple advantages over specialized proof assistants. Firstly, general-purpose provers enable the use of a rich set of mathematical libraries that are growing at a rapid pace due to the large community of users. Additionally, they may be easier to use for cryptographers without a strong background in formal verification, as multiple tutorials and resources for learning are available. Finally, their popularity means that they are better suited for future enhancements by machine-learning tools (e.g., [100]) whose quality depends on the size of the training sets. In fact, already in our work, we have used the GitHub Copilot tool[3] to assist us in writing types and definitions. (We also tried to use the tool to produce proofs, but it was not very helpful on that front.)

5.2.2 *Related Work*

As mentioned above, the most closely related work is [67], which introduced the approach that we formalize in Lean in this paper. In turn, [67] builds on the earlier work of Micciancio [73], who introduced the idea to use co-induction to translate the symbolic indistinguishability into computational indistinguishability.[4] The authors of [67] had already partially automated their security proof. Specifically, they were

2 To the best of our knowledge, this is similar to the approach followed by several other popular tools, such as Easycrypt, which simply assumes that every piece of code written by the user works in polynomial time.

3 See https://github.com/features/copilot.

4 The intuition behind the use of co-induction (put forward in [73], and followed in subsequent works like [67] and our paper), is that computational security works by

able to perform symbolic security checks of the garbling procedure, for any given circuit and input, and used this procedure for random testing on a circuit-by-circuit basis. However, they stopped short of a complete mechanized proof of security for all circuits, relying instead on a pen-and-paper proof. Mechanizing and formally verifying the symbolic security proof for all circuits and inputs, possibly within a general-purpose theorem prover, was left as an open problem in [67]. This is the challenge we solve in this paper.

Formalizing cryptographic protocols and security proofs has been an active area of research for many years, with several different approaches and tools being proposed. We have mentioned a few of them in the introduction, and now we would like to discuss two of them in more detail: EasyCrypt [16, 17] and Squirrel [11]. EasyCrypt is a general-purpose tool for formal verification of cryptographic designs, based on a powerful probabilistic Hoare logic. It allows users to write security proofs using security reductions, in the style of traditional "paper-and-pencil" proofs employed in theoretical cryptography papers. However, the process of writing such proofs is often complex and requires significant manual effort, as the user has to guide the proof search and provide detailed arguments for each step. Despite this apparent complexity, it is worth noting that EasyCrypt has been successfully used to formalize numerous complex cryptographic proofs. In particular, the security of garbled circuits (which we also formalize in this paper) had already been formalized in EasyCrypt, [4–6, 42, 55] allowing us to compare our approach with theirs. While EasyCrypt's approach is based on direct security reductions, our symbolic framework allows us to reason about the security of garbled circuits at a higher level of abstraction, using symbolic equivalence, with the security reductions present only in the proof of the general computational soundness theorem (not specifically related to garbled circuits). We believe that this modular approach simplifies the proof significantly, as it allows us to focus on the high-level structure of the protocol, instead of the details of the security reductions.

The next tool that we would like to discuss is Squirrel [11]. It is based on the Computationally Complete Symbolic Attacker (CCSA) framework [13, 14], which encodes arbitrary adversarial behaviors using symbolic (uninterpreted) functions. The Squirrel tool allows users to write security proofs in a logic that includes a predicate for indistinguishability of terms, and provides axioms and rules for reasoning about it. Thanks to this approach, Squirrel proofs are no longer based on series of reductions, but consist of derivations from provided axioms (similarly to the symbolic approach that we employ). The term language of Squirrel is quite general: It allows users to express a number of cryptographic primitives, including hash functions, symmetric and asymmetric encryption, signatures, and more. Remarkably, the indistinguishability relation in Squirrel is also computationally sound

reasoning on what information is guaranteed not to be known by an hypothetical adversary. So, defining the symbolic adversary's knowledge by co-induction (in the form of a greatest fixed point computation) is a better match for computational security.

– if two terms are indistinguishable in Squirrel, then their computational interpretations are also indistinguishable, provided that the underlying cryptographic primitives are secure. However, the proof of this soundness theorem has not been formalized in Lean or any other proof assistant, and due to the complexity of Squirrel's term logic, it seems that such a formalization would be quite challenging. Moreover, in the case of garbled circuits, we need to construct specific symbolic terms based on the circuit to be garbled, and it is not clear to us whether this can be done in Squirrel. (This was possible in our expressively limited framework, thanks to the underling programming language of Lean, which allows us to write functions that transform circuits into symbolic expressions.)

Several other recent works have used Lean to formalize cryptographic protocols and security proofs. In particular, [95] formalized a framework for reasoning about and manipulating oracle access (which we utilize in our work), the so-called forking lemma, and the Fiat-Shamir transform. Another example is [12], which formalized the soundness of SNARKs [20]. Finally, [71] formalized differential privacy.

Finally, we would like to compare the possibility of automation offered by various tools for formal cryptographic proofs. The symbolic tools that do not provide computational guarantees (like Proverif [22] and Tamarin [72]) usually offer high level of automation, often based on equational theories. Computationally sound tools such as EasyCrypt and Squirrel mostly manual, though some form of automation is available based on various heuristic procedures, which can significalty facilitate the proof process. (Notably, both tools offer support for SMT solvers.) In our framework, we do not develop any new automation techniques, but expect that the existing and future automation capabilities of Lean will be leveraged to facilitate the proof process. Our hope is that hiding the complexity of the IND-CPA assumption and security reductions behind a (computationally sound) symbolic framework, will make indistinguishability proofs more amenable to such general-purpose automation techniques. Finally, it might be worth noting that in our symbolic framework, the symbolic equivalence of two fixed expressions is decidable and effective. However, in our proof of garbled circuits, we cannot directly exploit this fact as we need to show the equivalence of two arbitrary expressions that depend on the input circuit. So, instead of using a decision procedure for specific expressions (e.g., as done in [67] to test the security of garbling on a circuit-by-circuit basis), we need to construct the equivalence proof by induction on the structure of the circuit given as input to the garbling algorithm.

5.2.3 *Limitations*

We now discuss the limitations of our proposed approach. The most significant limitation is the lack of support for interactive execution. In our model, the adversary's role is restricted to distinguishing between

two expressions, and cannot influence generated messages. Thus, our approach cannot capture scenarios where the adversary interacts with the protocol. Despite this limitation, our framework provides an effective method for analyzing complex expressions involving encryption operations, as demonstrated by our case study on circuit garbling.

The second limitation is that our formalized language of expressions supports only symmetric encryption. While the original work [67] describes a richer system that includes pseudo-random generators (PRGs), we omitted them for simplicity. We are confident that our modeling approach could be extended to include such primitives. However, supporting features such as digital signatures would require at least some support for interactive protocols.

5.2.4 *Lean Code*

The complete Lean code of our formalization is available on GitHub[5]. The code is equipped with a README file that explains its structure. Of course, the text of this paper is self-contained and can be read independently of the code. However, if the reader is interested in the details of our formalization, we recommend reading the code after reading the paper, as the paper provides a high-level overview of the formalization effort.

Additionally, we illustrate this paper with several Lean code snippets — some (Figures 1-7) are included in the main text, while others (Figures A.1-A.12) are deferred to the appendix. They are intended to give the reader a better idea of how our formalization looks like in practice. The snippets serve only an illustrative purpose — they are not essential for understanding the main ideas of our work, and can be safely skipped if the reader is not interested in the low-level details. However, we encourage the interested reader to look at them, as they provide a more concrete understanding of our formalization effort. We hope that the snippets can be understood intuitively, even by readers not familiar with Lean, as we have tried to keep them as self-explanatory as possible. In fact, the Lean syntax in the snippets has been slightly simplified for clarity, e.g. by removing implicit arguments, namespaces, or some type annotations. For the full, unabridged version of the snippets, we refer the reader to the supplementary material. For a survey on Lean references, see `https://leanprover-community.github.io/learn.html`.

5.3 SYMBOLIC CRYPTOGRAPHY

As highlighted above, indistinguishability is a cryptographic notion that expresses the inability of an adversary with bounded computational power to distinguish two objects (such as probability distributions over bitstrings) with non-negligible probabilistic advantage. These adversaries are typically modeled as probabilistic polynomial-

5 `https://github.com/ravst/SymbolicCryptographyLean`

time (PPT) interactive Turing machines. However, reasoning about such adversaries using proof assistants like Lean is notoriously difficult. Fortunately, a symbolic approach to indistinguishability already exists and offers a compelling alternative. This approach abstracts away computational complexity, making it significantly more amenable to formal verification and better aligned with the goals of formal methods.

At the core of the symbolic approach are *cryptographic expressions*, which are formal terms representing cryptographic operations such as encryption. On one hand, these expressions have a concrete *computational semantics*, allowing each expression to be compiled into a probability distribution over bitstrings that can be sampled and transmitted over a network. On the other hand, cryptographic expressions admit a notion of *symbolic indistinguishability* — a formal relation that does not rely on probabilistic polynomial-time adversaries. This relation is supported by a *soundness theorem*, which states that symbolically indistinguishable expressions yield computationally indistinguishable distributions when evaluated, allowing one to prove indistinguishability properties in a purely symbolic manner.

In this section, we present the definition of cryptographic expressions, their computational semantics, and their symbolic indistinguishability relation. The definitions are taken from [67] and formalized in Lean, but we aim to make this presentation self-contained and do not assume prior knowledge of [67] or Lean.

5.3.1 *Cryptographic Expressions*

Cryptographic expressions is a formal language of terms representing some common cryptographic constructions. Before presenting the formal definition, let us go trough a few examples:

1. **Key variables.** The term `VarK i` represents a variable storing a cryptographic key, where $i \in \mathbb{N}$ is the variable's identifier (or name). Intuitively, each such key variable represents a (uniformly) random key (see Subsection 5.3.2 for more details).

2. **Bit constants.** Term `Bit 0` and `Bit 1` represent bit constants `0` and `1`, respectively

3. **Encryption.** The constructor `Enc` represents encrypted values. For example `Enc (VarK 0) (Bit 1)` represents the bit `1` encrypted with a random key.

4. **Pairs.** The terms can be paired using the constructor $(\!(_, _)\!)$. For example, the expression $(\!($`VarK 0, Enc (VarK 0) (Bit 1)`$)\!)$ represents a pair of a random key and the bit `1` encrypted with that key.

5. **Reusing Keys.** The keys can be reused. For example the expression $(\!($`Enc (VarK 0) (Bit 1), Enc (VarK 0) (Bit 0)`$)\!)$ represents a pair of two ciphertexts encrypted with the same random key.

6. **Bit Variables.** The term `VarB i` represents a bit variable, where $i \in \mathbb{N}$ is the variable's name. Similarly to key variables, each bit variable represents a random bit.

7. **Conditional Swap.** The expression `Perm B` X_1 X_2 represents ((`X`$_1$`, X`$_2$)) or ((`X`$_2$`, X`$_1$)) depending on the value the bit `B`.

8. **Negation.** The bits can be negated using the constructor `Neg`. For example, the expression ((`Neg (VarB 0), VarB 0`)) represents a pair of two bits, where the first one is the negation of the second one.

9. **Empty expression.** Finally, `Eps` denotes the empty expression. It does not carry any information, but can be useful as a placeholder (esp. in the formalized setting).

Formally the expressions are defined by the following grammar:

$$\begin{aligned} \mathit{Expr} ::= {} & \texttt{Bit}\ \{0,1\} \mid \texttt{VarB}\ \mathbb{N} \mid \texttt{VarK}\ \mathbb{N} \mid ((\mathit{Expr}, \mathit{Expr})) \\ & \mid \texttt{Perm}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \mid \texttt{Neg}\ \mathit{Expr} \mid \texttt{Enc}\ \mathit{Expr}\ \mathit{Expr} \mid \texttt{Eps} \end{aligned}$$

There is also one more formal constructor called `Hidden k`, which represents unknown values encrypted with particular key. It is used to model the adversary's view of an expression, but it should not appear in user-defined expressions (see also Paragraph 5.3.3.2). We model the cryptographic expressions in Lean in a straightforward way using an inductive type, the definition can be found in file `Expression/Defs.lean` of the Lean codebase.

We only want to consider well-shaped expressions. For this reason, we define *shapes*, which serve as types for the expressions. Here are the possible shapes:

$$\begin{aligned} \mathit{Shape} ::= {} & \texttt{BitS} \mid \texttt{KeyS} \mid \texttt{EncS}\ \mathit{Shape} \\ & \mid \texttt{PairS}\ \mathit{Shape}\ \mathit{Shape} \mid \texttt{EpsS} \end{aligned}$$

The intended meaning of the shapes is natural. For example: the shape of ((`VarK 0, Enc (VarK 0) (Bit 1)`)) is `PairS KeyS (EncS BitS)`. The Lean formalization of shapes is part of the definition of cryptographic expressions (as the expressions are indexed by shapes) and can also be found in `Expression/Defs.lean`.

5.3.2 *Computational Semantics*

In this section, we equip the cryptographic expressions with a computational semantics. This serves two purposes: The first one is to clarify the intuitive meaning of cryptographic expressions, and the second one is to set out the foundation for the computational soundness theorem presented in Section 5.5.

To evaluate a cryptographic expression, we need to select an encryption function of the following type:

```
encrypt : BitVector κ → BitVector n →
    PMF (BitVector (encLength n))
```

In this type, `BitVector n` denotes a bit vector of length n, and `PMF X` denotes a distribution over `X`, given by a probability mass function. It follows that `encrypt` is a probabilistic function that takes a key of length κ a message of an arbitrary length $n \in \mathbb{N}$, and returns a ciphertext of a length that depends only on n (modelled by `encLength n`), but not on the content of the message. (For the full definition of an encryption scheme, that introduces κ and `encLength`, see the file `Expression/ComputationalSemantics/encryptionIndCpa.lean`.)

As mentioned earlier, the intended meaning of `VarK` and `VarB` is that they represent random keys and random bits. For this reason, in order to evaluate an expression, we draw a random value for each variable (see Figure 5.2). Once we are given the values of the variables, and the encryption function, we can evaluate the expression recursively in a straightforward way. One case that is slightly arbitrary is `Hidden k`, which we evaluate by using k to encrypt a bitstring of ones (i.e. `111...`), which servers as a placeholder for an unknown value. It is worth noting that the shape of an expression determines the length of the resulting bitstring, which allows us to chose the correct length of the placeholder.

The Lean definition of `evalExpr` is presented in Figure 5.1. In this listing, notation `let x ← X` denotes sampling x from a distribution X. (As a side node, let us mention that formally it is a monadic bind operation on the monad `PMF`).

Let us finish this section with a few examples of cryptographic expressions and their computational semantics:

1. The expression (`(Bit 0, Bit 1)`) results in the bitstring `01`, with probability 1.

2. The expression (`(VarB 0, VarB 1)`) results in the uniform distribution over the bitstrings `00`, `01`, `10`, and `11`, each with probability 1/4. This is because the two elements of the pair are independent random bits.

3. The expression (`(VarB 0, VarB 0)`) results in the uniform distribution over the bitstrings `00` and `11`, each with probability 1/2. This is because both the first and the second coordinate of the pair are the same random value.

4. The expression `Perm (VarB 0) (Bit 1) (Bit 0)` results in the uniform distribution over the bitstrings `10` and `01`, each with probability 1/2. This is because `0` and `1` are swapped according to the random value of `VarB 0`.

5.3.3 *Symbolic Indistinguishability*

The final piece of the puzzle is the *symbolic indistinguishability relation*. On one hand, it can be seen as a formal way of capturing the intuitive notion of indistinguishability of cryptographic expressions. On the other hand, it can be seen as a sufficient condition for the computational indistinguishability of the resulting probability distributions

```
def evalExpr (enc : encryptionFunctions κ) (kVars : ℕ →
    BitVector κ)
    (bVars : ℕ → Bool) (e : Expression s) : PMF (BitVector
    (shapeLength κ enc s)) :=
  match e with
  | Enc (VarK k) e => do
    -- Encrypt e using the key k.
    -- Lean type system enforces that
    -- the first argument is of the form (VarK k).
    let e' ← evalExpr enc kVars bVars e
    let key := kVars k
    enc.encrypt key e'
  | Pair e₁ e₂ => do
    -- Concatenate e₁ and e₂
    let e₁' ← evalExpr enc kVars bVars e₁
    let e₂' ← evalExpr enc kVars bVars e₂
    return (List.Vector.append e₁' e₂')
  | BitE b => do
    -- Lift a bit to a vector
    let b' := evalBitExpr bVars b
    return (List.Vector.cons b' List.Vector.nil)
  | VarK k => do
    -- Read the value of a key variable
    return (kVars k)
  | Perm (BitE b) e₁ e₂ => do
    -- Conditional swap:
    -- Return (e₁, e₂) if b is equal to 1 or (e₂, e₁) otherwise
    let b' := evalBitExpr bVars b
    let e₁' ← evalExpr enc kVars bVars e₁
    let e₂' ← evalExpr enc kVars bVars e₂
    if b' then
      return (List.Vector.append e₂' e₁')
    else
      return (List.Vector.append e₁' e₂')
  | Eps =>
    -- Empty bitstring
    return (List.Vector.nil)
  | Hidden (VarK k) => do
    -- Encode an arbitrary vector (e.g. only ones) using the key
    k.
    -- The idea is that since k is unknown to the adversary,
    -- it does not matter what is encrypted
    let key := kVars k
    enc.encrypt key ones
```

Figure 5.1: Definition of computational semantics, given the encryption scheme and variable valuations. (`Expression/ComputationalSemantics/Def.lean`.)

```
def exprToDistr
  (enc : encryptionFunctions κ)
  (e : Expression s):
  PMF (BitVector (shapeLength κ enc s)) := do
    let v := getMaxVar e + 1
    -- 'Fin v' denotes the set {0, ..., v - 1}
    let bvars ←
      uniformOfFintype (Fin v -> Bool)
    let kvars ←
      uniformOfFintype (Fin v -> BitVector κ)
    evalExpr enc
      (extendFin ones kvars)
      (extendFin false bvars) e
```

Figure 5.2: Full definition of the computational semantics of cryptographic expressions with random variable valuations.

(provided that the encryption scheme used to evaluate the expressions is secure). In this part of the paper, we will focus on the definition of the symbolic indistinguishability relation, and we will focus on its formal guarantees in Section 5.5.

5.3.3.1 *Examples*

Before presenting the formal definition of symbolic indistinguishability (in Section 5.3.3.2), we begin with a few illustrative examples. Consider first the following pair of expressions:

`Enc (VarK 0) (Bit 0)` and `Enc (VarK 0) (Bit 1)`

Those two expressions should be indistinguishable as both of them represent a bit encrypted with a random[6], undisclosed key. Next, let us consider a similar situation, but this time the encryption key is published:

`((VarK 0, Enc (VarK 0) (Bit 0)))` and
`((VarK 0, Enc (VarK 0) (Bit 1)))`

Those two expressions should count as distinguishable, because the adversary can decrypt the second coordinate using the key from the first coordinate, and check if it is `0` or `1`. As a side note, observe that the two examples show that indistinguishability is not a congruence — even if an expression Y is symbolically indistinguishable from Y', it does not mean that `((X, Y))` is symbolically indistinguishable from `((X, Y'))`.

The encryption can be chained. For example, the following pair of expressions should be indistinguishable:

`((Enc (VarK 0) (VarK 1), Enc (VarK 1) (Bit 0)))` and
`((Enc (VarK 0) (VarK 1), Enc (VarK 1) (Bit 1)))`

6 Remember that each `VarK i` represents a uniformly random key.

But if we prepend them both with the key `VarK 0`, they should become distinguishable.

As our next example, consider the following expressions:

```
((Enc (VarK 0) (Bit 0), Enc (VarK 0) (Bit 1)))   and
((Enc (VarK 0) (Bit 0), Enc (VarK 0) (Bit 0)))
```

Those two expressions should be indistinguishable. This is because our encryption function is probabilistic, so even though the first and second coordinates of the second expression are equal, they might (and probably will) result in different ciphertexts.

Let us now focus on a slightly counterintuitive aspect of encryption security: As it turns out, the usual IND-CPA notion of encryption security does not protect the security of messages encrypted cyclically [3, 88]. In particular, we should assume that encrypting a key with itself can potentially leak all information about the key. It follows that the following two expressions should count as distinguishable:

```
((Enc (VarK 0) (VarK 0), Enc (VarK 0) (Bit 0)))   and
((Enc (VarK 0) (VarK 0), Enc (VarK 0) (Bit 1)))
```

The cycle might be more complex than just encrypting a key with itself. For example, we should assume that the following expression leaks the keys `VarK 0` and `VarK 1`:

```
((Enc (VarK 0) (VarK 1), Enc (VarK 1) (VarK 0)))
```

Finally, let us discuss a few examples that have to do with bits and the conditional swap operations. First, we consider the following pair:

```
VarB 0   and   Neg (VarB 0)
```

Those two expressions are indistinguishable, because both of them result in the uniform distribution over `0` and `1`. Next, consider the following pair:

```
Perm (VarB 0) (Bit 0) (Bit 1)   and
   Perm (VarB 0) (Bit 1) (Bit 0)
```

Those two expressions are also indistinguishable, because they both result in the uniform distribution over the bitstrings `01` and `10` (each with probability 1/2). If we publish the value of `VarB 0` in the first coordinate, we get the following pair:

```
((VarB 0, Perm (VarB 0) (Bit 0) (Bit 1)))   and
((VarB 0, Perm (VarB 0) (Bit 1) (Bit 0)))
```

Those two expressions are not indistinguishable, because the adversary can use the first coordinate to reverse the conditional swap of the second coordinate and check if it is `01` or `10`.

5.3.3.2 *Definition*

We are now ready to present the formal definition of symbolic indistinguishability. As mentioned earlier, it is a simplified (PRG-free) version of the definition from [67]. We aim to present it here in a self-contained way, providing relevant snippets of Lean formalizations along the way.

The definition consists of three parts:

1. `adversaryView` function, which hides all parts of the expressions that are not accessible to the adversary, i.e. are encrypted with a key unknown to the adversary.
2. `applyVarRenaming` function, which allows us to rename both key and bit variables in the expressions.
3. `normalizeExpr` function, which performs all possible simplifications of the bit expressions.

Let us now discuss each of those parts in more detail.

ADVERSARY VIEW This is the is central notion of the symbolic cryptography. Its goal is to analyze the expression and decide which of its parts are available to the adversary. As first observed by Micciancio [73], this analysis must be carried out *co-inductively*, that is, using a *greatest fixpoint* rather than the more intuitive *least fixpoint*. Before presenting the construction formally, we give an overview of the main idea. The main part of computing the adversary's view of an expression is computing the set of keys available to the adversary. This is done in an iterative manner, where in each iteration we answer the question *If the adversary knows the set of keys K, what keys can they extract from the expression e?* and use the answer to update the set K. Importantly, this update may either add keys that become provably extractable or remove keys whose extractability cannot be justified. Computing the *least fixpoint* of this update corresponds to a process where we initialize the iteration with the empty set of keys and repeatedly apply this operator until no more keys can be added. This approach models adversaries that respect the symbolic structure of the expression and places the burden of proof on the adversary to prove which keys can be extracted. As it turns out, this is too weak to guarantee computational soundness. Instead, as observed in [73], we should compute the *greatest fixpoint*, which corresponds to a dual process where we initialize the iteration with the full set of keys and repeatedly remove those keys that cannot be extracted by the adversary until no more keys can be removed. This greatest-fixpoint approach shifts the burden of proof from the adversary, as it is now our job to demonstrate that certain keys are indeed inaccessible in order to take them away from the adversary. As shown in [73], this greatest fixpoint approach guarantees computational soundness (see Section 5.5.2.3 for an overview of the proof).

Let us now present the construction in more details. We define the iterative step of the computation as the function `keyRecovery`, which is given as a composition of two functions. The first one, called

`hideEncrypted K e`, answers the following question: *What parts of an expression e are available to an adversary who can decrypt only using the keys from the set K?* More formally, it takes as input an expression e and a set of keys K, and replaces every subexpression of the form `Enc k e'` with `Hidden k` whenever $k \notin K$, making the plaintext `e'` unavailable to the adversary. For example, for `E1 := Enc (VarK 0) (Bit 0)` and `E2 := Enc (VarK 1) (VarK 2)`, we have

```
hideEncrypted {VarK 1} ((E1, E2)) =
    ((Hidden (VarK 0), E2))
```

The second function, `extractKeys e`, answers the question: *What keys can an adversary learn from an expression e?* In other words, it extracts all keys that occur in the expression, subject to two important caveats. First, `extractKeys` looks inside subexpressions of the form `Enc k e'` (but not `Hidden k`). Although this may appear counterintuitive, hiding parts of the expression that are unavailable to the adversary is the responsibility of `hideEncrypted` while `extractKeys` merely operates on the already filtered view. Second, `extractKeys` does not extract keys that are used solely as encryption keys and do not appear elsewhere in the expression. This should be clear, as using a key for encryption does not make it accessible to the adversary. Both points are illustrated by the following example:

```
extractKeys (Enc (VarK 0) (VarK 1)) = {VarK 1}
```

The key, `VarK 0` is not extracted because it is used only as an encryption key. In contrast, `VarK 1` is extracted because, although encrypted, it appears as data in the expression.

Combining the two functions, we define `keyRecovery e K` as `extractKeys (hideEncrypted K e)`. Intuitively, this computes the set of keys that an adversary can extract from an expression e, given that it initially knows the keys in the set K. This corresponds to one iteration of the key-extraction process described in the first paragraph of this section. It is not hard to see (or to prove in Lean) that for a fixed expression e, the function `keyRecovery e` is monotone with respect to set inclusion. Therefore, by the Knaster–Tarski theorem, it admits both a least and a greatest fixpoint. As discussed above, we define the set of keys available to the adversary, denoted `adversaryKeys e`, as the greatest fixpoint of `keyRecovery e`. Finally, we define the adversary's view of the expression, `adversaryView e`, as the part of e that is visible to an adversary who knows the keys `adversaryKeys e`, i.e. `hideEncrypted (adversaryKeys e) e`. The Lean formalization of this construction can be found in Figure 5.4.

Let us finish this section by discussing Figure 5.3, which presents an example of computing `adversaryKeys` and `adversaryView`. The figure also compares the greatest- and least-fixpoint strategies and shows that they can produce different results. Observe that this mismatch arises from cyclic encryption. In fact, this is the general source of such discrepancies: one can show that in the absence of cyclic encryption, the greatest and least fixpoints always coincide.

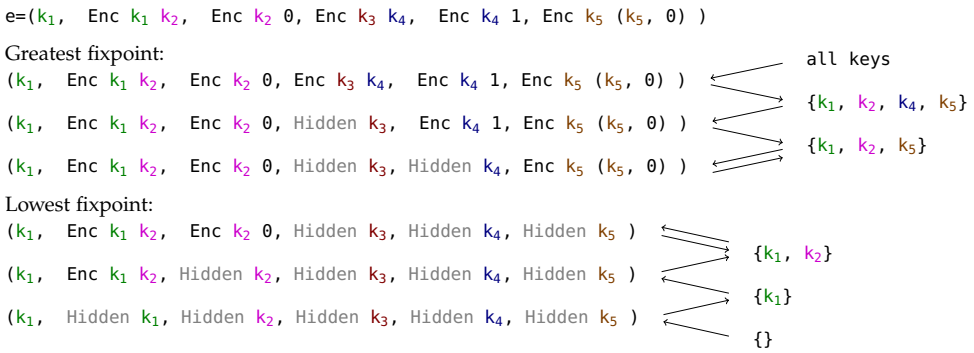


Figure 5.3: Example of the computation of `adversaryView`: The expression e is the input expression. Arrows directed from left to right represent the set of keys that can be extracted from a given expression (as defined by the `extractKeys` function). Arrows from right to left take a set of keys K as input and map it to `hideEncrypted e K`, where e is the fixed input expression.

VARIABLE RENAMING A *variable renaming* consists of two parts: a key variable renaming and a bit variable renaming. Key renaming is relatively straightforward: valid key renaming is a bijection that maps each key variable to another key variable. It is modeled as a bijection of type $\mathbb{N} \to \mathbb{N}$, and it is applied to an expression by applying it pointwise to every key variable. The formalization can be found in `Expression/SymbolicIndistinguishability.lean`.

`BitRenaming` is slightly more complex — it is a function that maps each bit variable to either another bit variable or its negation. (The ability to negate is essential because we want `VarB 0` and `Neg (VarB 0)` to be indistinguishable.) A bit renaming is *valid* if it is bijection of bit variables after ignoring negations. Similar to key renaming, a bit renaming is applied to an expression pointwise to every bit variable. (We omit the formal definition here; it can be found in file `Expression/SymbolicSemantics.lean`.)

Finally, a `varRenaming` is a pair consisting of a key renaming and a bit renaming. It is valid if both of its components are valid, and it is applied to an expression by independently applying the key and bit renamings.

NORMALIZE The function `normalizeExpr` simplifies expressions by applying the following rules.

1. `Neg (Bit 0)` simplifies to `Bit 1`, and `Neg (Bit 1)` simplifies to `Bit 0`.
2. `Neg (Neg X)` simplifies to `X`.
3. `Perm (VarB 0) X Y` simplifies to `((Y, X))`, and `Perm (VarB 1) X Y` simplifies to `((X, Y))`.
4. `Perm (Neg B) X Y` simplifies to `Perm B Y X`.

The Lean formalization of `normalizeExpr` can be found in `Expression/SymbolicIndistinguishability.lean`.

```
def hideEncrypted (keys : Finset (Expression KeyS)) (e :
    Expression s) : Expression s :=
  match e with
  | Pair e1 e2 => Pair (hideEncrypted keys e1) (hideEncrypted
    keys e2)
  | Perm b e1 e2 => Perm b (hideEncrypted keys e1)
    (hideEncrypted keys e2)
  | Enc k e =>
    if k ∈ keys
    then Enc k (hideEncrypted keys e)
    else Hidden k
  | e => e -- In all other cases, we leave the expression as is.

def extractKeys (e : Expression s) : Finset (Expression KeyS) :=
  match e with
  | VarK e => {VarK e}
  | Expression.Pair p1 p2 => (extractKeys p1) ∪ (extractKeys p2)
  | Expression.Perm _ p1 p2 => (extractKeys p1) ∪ (extractKeys
    p2)
  | Expression.Enc _ e => (extractKeys e) -- We omit the key
    used for encryption
  | _ => ∅ -- In all other cases, we return the empty set.

def keyRecovery (e : Expression s) (S : Finset (Expression
    KeyS)) : Finset (Expression KeyS) :=
  extractKeys (hideEncrypted S e)

def adversaryKeys (e : Expression s) : Finset (Expression KeyS)
    :=
  -- 'greatestFixpoint f X Y Z' returns the greatest fixpoint of
    'f',
  -- the other arguments are only needed to prove termination,
    and do not influence the output.
  greatestFixpoint (keyRecovery e) _ _ _

def adversaryView (e : Expression s) : Expression s :=
  hideEncrypted (adversaryKeys e) e
```

Figure 5.4: Definition of `adversaryView`. (`Expression/SymbolicIndistinguishability.lean`.)

SYMBOLIC INDISTINGUISHABILITY Finally, we combine the three parts to define the symbolic indistinguishability relation, by saying that two expressions e_1 and e_2 are symbolically indistinguishable, if we can rename the variables in e_1, in such a way that the two expressions are equal after normalizing and applying the adversary's view. See `Expression/SymbolicIndistinguishability.lean` for the Lean formalization.

5.4 EXTENDED EXAMPLE — CIRCUIT GARBLING

In this section, we present a use case for our framework: an implementation of circuit garbling in Lean, together with a formal proof of symbolic security (following [67]). This example serves two primary purposes. First, it demonstrates the potential of the symbolic framework for formalizing cryptographic properties in Lean. Second, it provides a concrete symbolic proof that can serve as a starting point for further formalization efforts.

The structure of this section is as follows. In Section 5.4.1, we provide background on circuit garbling and multi-party computation. In Section 5.4.2, we present our modeling of the circuit garbling protocol and formally state its security property in the symbolic framework. Finally, in Section 5.4.3, we present a high-level overview of our proof of symbolic security, focusing on the *pseudo-fixpoint* reasoning technique that we developed to facilitate the formalization.

5.4.1 *Multi-Party Computation and Circuit Garbling*

Multi-party computations (MPCs) [49, 102] (see, also [43]) is a family of cryptographic protocols that allow multiple parties to jointly evaluate a function without revealing their inputs to each other. In this paper, we are interested in the *two*-party version of this notion. Specifically, consider a scenario in which two parties, Alice and Bob, wish to evaluate a publicly-know logical circuit where some input wires belong to Alice and others to Bob. The MPC protocol enables them to perform this evaluation without revealing anything more about their inputs than what can be inferred from the circuit's output. Two important properties of an MPC protocol are security and correctness. An MPC protocol is secure if it guarantees the privacy of the inputs, and correct if it ensures that the parties always reach the correct output. See [43] for a more detailed discussion of MPC, including the formal definitions of security and correctness.

One of the techniques for achieving secure multi-party computation is through Yao's circuit garbling [102]. The core of a garbling protocol (see also [67, Definition 5]) is a probabilistic function:

$$\mathsf{Garble}(\mathsf{circuit},\ \mathsf{input}) = (\mathsf{gCircuit},\ \mathsf{gInput})$$

Here, *circuit* is a representation of a logical circuit (for a formal definition, see, e.g, [67, Definition 4]), and *input* is a bit vector representing the inputs to the circuit (the Garble procedure does not distinguish

between input wires belonging to Alice or Bob). The output is a pair *gCircuit* and *gInput*, both encoded as bit vectors. The idea is that *gCircuit* and *gInput* should obscure the original circuit and input, in a way that allows evaluating the circuit ("correctness") while revealing nothing about the original input beyond what can be inferred from the output ("security").

Formally, the security of the garbling protocol is defined in terms of indistinguishability. In order for a garbling protocol to be secure (see also [67, Definition 6]), there must exist a function *Simulate*:

$$\mathsf{Simulate}(\mathsf{circuit},\ \mathsf{output}) = (\mathsf{sCircuit},\ \mathsf{sInput})$$

Such that for every circuit C and input x, the following distributions are indistinguishable:

$$\mathrm{Garble}(C, x) \quad \text{and} \quad \mathrm{Simulate}(C, C(x))$$

Indeed, if it is possible to simulate the output of $\mathrm{Garble}(C, x)$ without knowing the input x, then it is impossible to learn anything more about x from $\mathrm{Garble}(C, x)$ than what can already be inferred from $C(x)$.

For the correctness of the garbling protocol (see also [67, Definition 6]), we require that there exists an explicit and efficient function *Evaluate*:

$$\mathsf{Evaluate}(\mathsf{gCircuit},\ \mathsf{gInput}) = \mathsf{output}$$

such that for every circuit C and input x, we have:

$$\mathrm{Evaluate}(\mathrm{Garble}(C, x)) = C(x)$$

Usually, standard two-party computation constructions based on garbled circuits require the garbling scheme to be *projective*. To explain what this means, let us first define the function:

$$\mathrm{proj} : \left(T^2\right)^n \times \{0,1\}^n \to T^n,$$

which takes a list of pairs and a selector string (both of length n) and selects one element from each pair according to the corresponding bit in the selector string (where T is an arbitrary type). We say that a garbling function is a *projective scheme* if it is expressed as:

$$\mathsf{Garble}(\mathsf{circuit},\ \mathsf{input}) := (\mathsf{gCircuit},\ \mathsf{proj}(\mathsf{gInputPairs},\ \mathsf{input})),$$

where *gCircuit* and *gInputPairs* are produced by a function that does not depend on the input:

$$\mathsf{preGarble}(\mathsf{circuit}) = (\mathsf{gCircuit},\ \mathsf{gInputPairs})$$

A projective garbling scheme is secure if it satisfies the same indistinguishability condition as above. Observe, in particular that in a secure projective scheme $\mathsf{proj}(\mathsf{gInputPairs},\ \mathsf{input})$ cannot reveal anything about *input* beyond what can be inferred from the output of the circuit.

A secure projective garbling protocol can be used to implement a two-party computation protocol — see [19, Figure 3] or [19, Section 7.1]. For the sake of completeness, and to justify the notion of projective schemes, let us briefly sketch one way to achieve this. The construction requires a cryptographic primitive known as *oblivious transfer* [44, 87] (see [43]). Oblivious transfer works as follows: Alice has two values v_0 and v_1, and Bob has a bit $b \in \{0,1\}$. With the oblivious transfer protocol, Bob learns v_b without learning the other value $v_{\neg b}$, while Alice does not learn b. See [80]. Several efficient implementations of oblivious transfer exist, based on multiple plausible assumptions, such as the *quadratic residuosity* [44, 87], *decisional Diffie-Hellman* [80], or *learning with errors* [83]. Using the oblivious transfer primitive, we can implement the two-party version of the garbling protocol as follows:

1. Alice runs the *preGarble* function on C, obtaining *gCircuit* and *gInputPairs*.
2. Alice sends *gCircuit* to Bob.
3. For each bit in her input, Alice sends the corresponding value from *gInputPairs* to Bob.
4. For each bit in his input, Bob uses oblivious transfer to receive from Alice the corresponding value from *gInputPairs*.
5. Bob now has *gCircuit* and *gInput*, so he can evaluate the circuit and send the output to Alice.

Let us finish this section by noting that garbling circuits is an important and active area of research. We refer the reader to [19] for a survey of protocols and applications. One particularly active line of work focuses on optimizing the garbling protocol to make it more efficient, e.g., the *free XOR* [63] and the *row-reduction* [84] techniques. Interestingly, the original paper on circuit garbling by Yao [102] did not provide a formal proof of security, and it took over two decades before a (pen-and-paper) proof of security was provided by Lindell and Pinkas [69] and by Bellare et al. [19]. This may serve as an illustration of the complexity of the problem, and the need for formalizing security proofs in a proof assistant.

5.4.2 *Security of Circuit Garbling*

In this section, we explain how we modelled the circuit garbling protocol in Lean, and how we formalized its symbolic security statement. Our implementation of symbolic garbling is based on the construction from [67] and we will not repeat it here. However, during formalization, we found that the use of pseudo-random generators (PRGs), as employed in [67], was unnecessary. In the original approach, when a circuit diagram had an output wire connected to several inputs, PRGs were used to expand the keys associated with a given wire so that each input wire had a different key. This use of PRGs is non-standard. We observed that even in the symbolic model, PRGs are not needed. Our

```
def preGarble (c : Circuit inWires outWires) :
  (garbledCircuitType c) × (projLabelType inWires) := [...]

def Garble (c : Circuit inWires outWires) (input : bundleBool
    inWires) :=
  let (gCircuit, gInputPairs) := preGarble c
  garbleOutputToExpr (gCircuit, makeProjection gInputPairs input)

def SimulateG (c : Circuit inWires outWires) (output :
    bundleBool outWires) := [...]

lemma garblingSecureSymbolic :
    ∀ (c : Circuit inWires outWires)
      (input : bundleBool inWires),
        symIndistinguishable
          (Garble c input)
          (SimulateG c (evalCircuit c input))
```

Figure 5.5: Types of the garble function. (`Garbling/GarbleDef.lean`)

formalization relies solely on encryption, following standard practice. This not only improves efficiency but also simplifies the formalization effort. For further details, we refer the reader to our implementation.

We implemented `preGarble` in Lean and used it to define `Garble`, as required by the definition of a *projective scheme*. Importantly, `Garble` does not produce a distribution over bit vectors directly. Instead, it returns a cryptographic expression, which can later be compiled into a distribution using the computational semantics introduced in the previous section. (Since cryptographic expressions do not support lists, we represent lists using nested tuples instead.) We also implemented the function `SimulateG`, which simulates the output of `Garble` given only the circuit and its output; this function also returns a cryptographic expression. Finally, we proved that for every circuit C and input x, the expressions `Garble C x` and `SimulateG C (evalCircuit C x)` are symbolically indistinguishable. See Figure 5.5 for the definition of `Garble` (in terms of `preGarble`), the types of `preGarble` and `SimulateG`, and the statement of the symbolic indistinguishability property.

5.4.3 *Proof Technique for Working with Fixpoints*

While [67] provides a pen-and-paper proof of correctness, our proof differs from their approach so that it is more amenable to formalization in Lean. However, due to space constraints, we do not present the entire proof here. Instead, we focus on its high-level structure including the pseudo-fixpoint reasoning technique. For the full proof, please refer to the supplementary material.

Let us start by briefly outlining our proof technique for establishing symbolic indistinguishability of garbled circuits. Let g be the expression produced by `Garble` and s the one produced by `SimulateG`. We begin the proof by giving explicit (i.e. free of fixpoints) definitions for `adversaryKeys g` and `adversaryKeys s`. (This is the most difficult

step, as it involves reasoning about fixpoints.) The remaining steps are relatively straightforward: we use the definitions of `adversaryKeys` to produce explicit formulas for `adversaryView` of g and s, and then provide a variable renaming r such that applying r to `adversaryView g`, followed by normalization, yields `adversaryView s`. The main technical challenge is reasoning about the `adversaryKeys` function, which is defined via a fixpoint computation.

In the remainder of this section, we explain our general technique for reasoning about fixpoints. To illustrate it, consider an expression `e = (⟨e₁, e₂⟩)`. The core difficulty lies in the fact that `adversaryKeys` is not compositional: computing it separately for `e₁` and `e₂` tells us little about `adversaryKeys e`. For example, suppose `e₁` is the key variable `k₁`, and `e₂ = Enc k₁ k₂`. Then `adversaryKeys e₁ = {k₁}` and `adversaryKeys e₂ = ∅`, but `adversaryKeys e = {k₁, k₂}`, with `k₂` appearing seemingly from nowhere.

However, expressions produced by actual protocols, such as `Garble` or `SimulateG`, are highly structured. This allows us to use a powerful fixpoint reasoning technique. First, those expressions do not contain cyclic encryptions, so we can prove that `keyRecovery` has a unique fixpoint, and not worry about `adversaryKeys` being the greatest one.

Another key property of g and s is that they are usually non-overlapping: if `(⟨e₁, e₂⟩)` is a subexpression of g or s, then the sets of keys written in `e₁` and `e₂` (i.e. the ones returned by `extractKeys`) are disjoint — although `e₂` may still use keys from `e₁` for encryption. To take advantage of this, we introduce the following notion: a set of keys K is a *pseudo-fixpoint with respect to U on e* if

```
(keyRecovery e K) ∩ U = K ∩ U
```

That is, the keys in K agree with the result of `keyRecovery` on the subset U.

We use this definition as follows. To prove that a set X is a fixpoint of `keyRecovery` on `(⟨e₁, e₂⟩)`, it suffices to show that X is a pseudo-fixpoint on `e₁` with respect to `extractKeys e₁`, and on `e₂` with respect to `extractKeys e₂`. Of course, this only works when `(⟨e₁, e₂⟩)` satisfies the sortedness condition above. Interestingly, this technique turned out to be powerful enough to reason about `adversaryKeys` for g and s, significantly simplifying the symbolic indistinguishability proof.

5.5 COMPUTATIONAL INDISTINGUISHABILITY

In this section, we revisit the notion of indistinguishability from the standard computational perspective. We begin by discussing the conventional IND-CPA (indistinguishability under chosen-plaintext attack) definition of encryption security and how we formalize it in Lean. Next, we formally state and prove the computational soundness theorem, which establishes a connection between symbolic and computational indistinguishability. This theorem enables us to lift symbolic indistinguishability results to the computational setting; in particular, it allows us to derive the computational security of garbled circuits from their symbolic proof.

5.5.1 *Indistinguishability-Based Security of Encryption Schemes*

The standard notion of encryption security is called *IND-CPA*, which stands for *indistinguishability under chosen-plaintext attack*. This definition captures the idea that an adversary should not learn anything about the plaintext, even when the same encryption key is used to encrypt multiple messages chosen by the adversary. IND-CPA security can be formalized in various equivalent ways; here, we follow the definition given in [67, Definition 2], which defines it in terms of two oracles, $\mathcal{O}_L$ and $\mathcal{O}_R$. Both oracles are initialized with a randomly chosen key $\mathtt{k}$, and respond to queries of the following form:

Input:	A pair of messages (m_L, m_R)
Output of $\mathcal{O}_L$:	m_L encrypted with $\mathtt{k}$
Output of $\mathcal{O}_R$:	m_R encrypted with $\mathtt{k}$

We say that an encryption scheme is *IND-CPA secure* if no probabilistic polynomial-time (PPT) adversary can distinguish between the two oracles with non-negligible probability. However, the definitions of polynomial time and negligible probability both require a parameter that measures the complexity of the oracles. This parameter is known as the *security parameter*[7] κ, which in our case is equal to the length of the encryption key. For this reason, we define an encryption scheme as a *family* of encryption functions indexed by the security parameter κ. (For Lean formalization see `Expression/ComputationalSemantics/encryptionIndCpa.lean`). Based on such an encryption scheme, we construct the families of oracles $\mathcal{O}_L^\kappa$ and $\mathcal{O}_R^\kappa$, and say that the encryption scheme is IND-CPA secure if no PPT adversary can distinguish the outputs of the two oracles with non-negligible probability, both with respect to κ. (A probability is called *negligible* if it decreases faster than the inverse of any polynomial in κ.)

5.5.1.1 *Oracles in Lean — VCVio*

To talk about *oracles* and *oracle computations* in Lean, we use the VCVio[8] library [95], which provides a general framework for modeling oracle-based computations. In our formalization, $\mathcal{O}_L$ and $\mathcal{O}_R$ are implemented as VCVio oracles, and the adversary is modeled as an oracle computation that interacts with them. Below, we briefly describe how VCVio defines these two notions. The interface between oracles and oracle computations is defined by the oracle's specification, `OracleSpec I`. It is parameterized by a type `I`, which represents the set of query labels that can be asked to the oracle. For each $i \in I$, the specification assigns a pair of types:

$$(\text{Input type of the query } i,\ \text{Output type of the query } i)$$

7 In the literature, the security parameter is often denoted by λ. However, since in Lean λ is a reserved keyword, we denote it by κ instead.

8 See: github.com/dtumad/VCV-io.

As an example, let us discuss the specification of the oracles $\mathcal{O}_L$ and $\mathcal{O}_R$. (Observe that since the two oracles answer the same type of query, they must share the same specification.) Although they intuitively implement only one type of query, for technical reasons, it is convenient to take $I = \mathbb{N}$ and define a separate query for each possible message length. The n-th query then inputs a pair of messages of length n and returns a ciphertext of length `encLength n`. The specification is formally defined as `oracleSpecIndCpa` in `Expression/ComputationalSemantics/encryptionIndCpa.lean`.

An *oracle* is defined as an implementation of the oracle specification and has the type `QueryImpl (spec : OracleSpec I) M`, which, to every $i \in I$, associates a function of type:

```
In i → M (Out i)
```

where `In i` and `Out i` are the input and output types of the query i, as specified by `spec`. For example, in the case of the oracles $\mathcal{O}_L$ and $\mathcal{O}_R$, `In n = BitVector n × BitVector n`, and Out n is `BitVector (encLength n)`. Observe that the output type of the oracle query is modified by a functor M — this should be a monad that represents the computational effects used by the oracle, such as state or nondeterminism. In our case, we always use `M a` defined as `PMF (Option a)`, which models probabilistic computations that may fail. (This monad is also commonly written as `(OptionT PMF) a`.) We use the `PMF` monad to model the probabilistic nature of encryption schemes, while the `Option` monad is required by VCVio, which assumes that oracle computations can fail. (However, the use of `Option` is purely technical: we will only consider oracles that never fail, i.e., return `none` with probability zero.)

Finally, an oracle computation that interacts with an oracle and returns a value of type `A` is represented by the type `OracleComp (spec : OracleSpec I) A`. It can be seen as an abstract computation that is designed to work with any oracle implementing the specification `spec` (formally, it is defined as a *free monad*, see e.g. [90, Section 6]). It comes equipped with a function `simulateQ` that, given an actual oracle implementation `impl : QueryImpl spec M`, performs the computation and returns a value of type `M A`.

5.5.1.2 *Oracles' indistinguishability in Lean*

We are now going to define computational indistinguishability for *families of seeded oracles* — a notion we will use to formalize IND-CPA security of encryption schemes. While the previous paragraphs discussed existing definitions from VCVio, the rest of this section presents our original contributions. A *family of seeded oracles* (formally defined in Figure 5.6) consists of one oracle for each security parameter κ, and each oracle is initialized with a random value drawn from a specified distribution. For example, the oracles $\mathcal{O}_L$ and $\mathcal{O}_R$ are initialized with a uniformly random encryption key.

An adversary that interacts with a family of seeded oracles is represented as a *family of oracle computations* (formalized as `famOracleComp` in Figure 5.6). Such a family consists of one oracle computation for

```
-- A family of seeded oracles is parametrized
-- by a family of specifications.
structure famSeededOracle
    (Spec : ℕ → OracleSpec I) where
        -- 1. Type of the seed:
        Seed : (κ : ℕ) → Type
        -- 2. Initial distribution:
        seedDistr : (κ : ℕ) →
          OptionT PMF (Domain κ) Seed
        -- 3. Query implementations:
        queryImpl : (κ : ℕ) → Seed κ →
          QueryImpl (Spec κ) (OptionT PMF)

def famOracleComp (Spec : ℕ → OracleSpec I)
  (Output : ℕ → Type) := (κ : ℕ) →
    OracleComp (withRandomI Spec κ) (Output κ)

-- Run ‘famOracleComp‘ on ‘famSeededOracle‘.
-- ‘addRandom‘ implements randomness queries.
def compToDistrGen
  (oracle : famSeededOracle Spec)
  (comp : famOracleComp Spec Output) (κ : ℕ)
  : (OptionT PMF (Output κ)) := do
    let seed ← oracle.seedDistr κ
    OracleComp.simulateQ (addRandom (oracle.queryImpl κ seed))
    (comp κ)
```

Figure 5.6: Definition of a family of seeded oracles. (`ComputationalIndistinguishability/Def.lean`)

each value of the security parameter κ. Since we want the adversary to be a probabilistic computation, it is, in addition to its input oracle, also given access to a randomness oracle, which generates random values from specified distributions.

An important property of families of seeded oracles and families of oracle computations is that they can be plugged together using `simulateQ`, producing a family of distributions over output values (parametrized by κ). This plugging function is formalized as `compToDistrGen` in Figure 5.6.

We say that two families of seeded oracles are *computationally indistinguishable* if no probabilistic polynomial-time distinguisher can tell them apart with non-negligible probability. A distinguisher is a special kind of family of oracle computations that returns a Boolean for all values of the security parameter κ (i.e., `famOracleComp Spec (fun _ => Bool)`). Formalizing the definition of negligibility is straightforward (see `negl` in Figure 5.7). The most challenging part is defining which families of oracle computations run in polynomial time — we postpone this discussion to Section 5.5.2.2. For now, the definition of indistinguishability simply takes a predicate `(IsPolyTime : PolyFamOracleCompPred)` as a parameter. See Figure 5.7 for the Lean definition of oracle indistinguishability.

```
-- A type of predicates on ‘famOracleComp‘.
-- Used to represent the poly-time predicate.
def PolyFamOracleCompPred : Type :=
    famOracleComp Spec Output -> Prop

-- Definition of negligible probability.
-- This is a less standard definition,
-- but it is equivalent to the usual one
-- (see the codebase for details).
def negl (f : ℕ -> ℝ₊) : Prop :=
  ∀ k, ∃ (B : ℝ), ∀ i, (f i) * (i^k) <= B

-- The adversary's distinguishing advantage
-- between two Boolean distributions.
def advantage (x y : ℕ → PMF (Option Bool)) : ℕ → NNReal :=
  fun κ =>
    -- ‘distance p q‘ is defined as | p - q |.
    distance (getPMF (x κ) (some True))
             (getPMF (y κ) (some True))

def CompIndistinguishabilitySeededOracle
  (IsPolyTime : PolyFamOracleCompPred)
  (o1 o2 : famSeededOracle Spec)
  : Prop :=
    -- All distinguishers ...
    ∀ distinguisher :
        famOracleComp Spec (fun _ => Bool),
    -- ... that run in polynomial time ...
    (IsPolyTime distinguisher) ->
    -- ... only achieve negligible advantage.
    negl (advantage
      (compToDistrGen o1 distinguisher)
      (compToDistrGen o2 distinguisher))
```

Figure 5.7: Definition of oracle indistinguishability. (`ComputationalIndistinguishability/Def.lean`)

With the notion of indistinguishability of families of seeded oracles in place, we can now formally define the IND-CPA security of an encryption scheme by directly formalizing the definition given in Section 5.5.1 — see `Expression/ComputationalSemantics/encryptionIndCpa.lean`.

5.5.2 *Computational Soundness Theorem*

Having formalized the IND-CPA security assumption for the encryption scheme, we are now ready to formalize the computational soundness theorem, which states that symbolic indistinguishability implies computational indistinguishability (as long as the encryption scheme used in the computational semantics is IND-CPA secure). In this subsection, we will: (a) formalize symbolic indistinguishability for families of distributions; (b) discuss our axiomatic approach for handling polynomial-time computations; (c) state the computational soundness theorem; and (d) explain our proof strategy.

5.5.2.1 *Computational Indistinguishability of Distributions*

The computational soundness theorem is formulated in terms of indistinguishability between families of distributions, rather than seeded oracles. We therefore begin by formalizing distribution indistinguishability. Although simpler and closely related to oracle indistinguishability, it will play a role in our axiomatic approach to polynomial-time computations.

A *family of distributions* is a function `(κ : ℕ) → PMF (Option (Domain κ))`, where `Domain` is a family of types. The adversary is modeled as a family of functions `Domain κ → PMF (Output κ)` for each κ. To apply such a family of computations f to a family of distributions d, we sample a value x from d_κ for each κ, and compute $f_\kappa(x)$. See `ComputationalIndistinguishability/Def.lean` for Lean formalization.

Instead of introducing a new abstract predicate for polynomial-time families of simple computations, we reuse the one defined for families of oracle computations. To do this, we observe that every family of simple computations f_κ can be lifted to a family of oracle computations that queries a single oracle for a value of type `Domain` κ, applies f_κ to this value, and then uses the randomness oracle to sample from the resulting distribution. Thanks to this lift, we can apply any abstract predicate `p : PolyFamCompPred` defined for families of oracle computations to families of simple computations. See `ComputationalIndistinguishability/Def.lean` for Lean formalization.

Finally, we say that two families of distributions are indistinguishable if no polynomial-time distinguisher (modeled as a family of simple computations d_κ : `Domain`$_\kappa \to$ `PMF Bool`) can distinguish them with non-negligible probability. See `ComputationalIndistinguishability/Def.lean` for Lean formalization.

5.5.2.2 *Axiomatizing Polynomial Time*

The notion of polynomial-time computation is central to much of modern cryptography, but it is challenging to model in formal theorem provers such as Lean (or Easycrypt). This is because they do not have a built-in notion of a function's running time, so polynomial time must be defined from scratch using Turing machines or some other equivalent model. Dealing with *probabilistic polynomial-time adversaries* is particularly challenging, as they are usually framed as *probabilistic* and *interactive* Turing machines — a fairly natural but impractical model for formal verification. For this reason, in this work we adopt an alternative, axiomatic approach and model it as an abstract predicate on families of oracle computations (i.e., of type `PolyFamOracleCompPred`) that satisfies the following two axioms:

IT IS CLOSED UNDER COMPOSITION Observe that families of oracle computations (`famOracleComp`) can be composed with families of simple computations (`famComp`), resulting in a new element of `famOracleComp`. This is because, just before returning its result, a `famOracleComp` can apply a `famComp` transformation to it. This axiom states that if both the input `famOracleComp` and the input `famComp` run in polynomial time, then the resulting `famOracleComp` also runs in polynomial time. See Figure 5.8 for the formal statement.

IT CONTAINS SPECIFIC COMPUTATIONS The second axiom is more ad hoc than closure under composition: it postulates that a specific `famOracleComp`, called `reductionHidingOneKey`, which we use in the proof, runs in polynomial time. The reduction is discussed in Section 5.5.2.3 and formalized as `reductionHidingOneKey` in `Expression/ComputationalSemantics/SoundnessProof/HidingOneKey.lean`. This axiom serves as a way of lifting a pen-and-paper complexity analysis into a formal framework, while relying on the reader to verify this particular complexity claim provided as a commend at the end of `Expression/ComputationalSemantics/SoundnessProof/HidingOneKey.lean` or in the full version of this paper.

5.5.2.3 *Computational Soundness Theorem*

We are now ready to state the computational soundness theorem. It asserts that symbolically indistinguishable expressions yield computationally indistinguishable distributions — provided that the underlying encryption scheme is IND-CPA secure. Its Lean statement is given in Figure 5.9. Let us now explain a high level overview of its formalized proof (to see the full formalized proof, please refer to the code in supplementary materials).

We start by unfolding the definition of symbolic indistinguishability, obtaining that for some valid renaming `r`, we have:

```
normalizeExpr (applyVarRenaming r (adversaryView expr1)) =
              normalizeExpr (adversaryView expr2)
```

```
def composeOracleCompWithSimpleComp
  (oracleComp : famOracleComp Spec Domain)
  (f : famComp (Domain) Output)
  : famOracleComp Spec Output := fun κ => do
    let val : (Domain κ) ← oracleComp κ
    let ret ← sample (f κ val)
    return ret

def PolyTimeClosedUnderComposition
    (isPolyTime : PolyFamOracleCompPred) :=
  ∀ I (Spec : (κ : ℕ) → OracleSpec I),
  ∀ Domain Output,
  ∀ (oracleComp : famOracleComp Spec Domain),
  ∀ (f : famComp Domain Output),
  isPolyTime oracleComp →
  polyTimeFamComp isPolyTime f →
  isPolyTime (composeOracleCompWithSimpleComp
    oracleComp f)
```

Figure 5.8: The composition axiom for polynomial time. (ComputationalIndistinguishability/Def.lean)

```
theorem symbolicToSemanticIndistinguishability :
  -- For every predicate 'polyTime',
  (polyTime : PolyFamOracleCompPred)
  -- satisfying our axioms
  (HPolyTime : PolyTimeClosedUnderComposition IsPolyTime)
  (Hreduction : ∀ enc shape (expr : Expression shape) key₀,
    IsPolyTime (removeOneKeyReduction enc expr key₀))
  -- For every encryption scheme,
  (enc : encryptionScheme)
  -- that is IND-CPA secure (for 'polyTime')
  (HEncIndCpa : encryptionSchemeIndCpa IsPolyTime enc)
  -- If a pair of expressions
  (expr1 expr2 : Expression shape)
  -- is symbolically indistinguishable
  (Hi : symIndistinguishable expr1 expr2) :
  -- it is also
  -- computationally indistinguishable
  CompIndistinguishabilityStrict IsPolyTime
    (exprToFamDistr enc expr1)
    (exprToFamDistr enc expr2)
```

Figure 5.9: The statement of the computational soundness theorem. (Expression/ComputationalSemantics/Soundness.lean).

We then show that both `normalizeExpr` and `applyVarRenaming r` (for a valid `r`) preserve the computational semantics — that is, they do not affect the resulting distributions:

- The proof for `normalizeExpr` proceeds by a rather straightforward structural induction on expressions.
- The proof for `applyVarRenaming r` relies on two observations: (a) variable renaming can be simulated by permuting the corresponding values before evaluation, and (b) shuffling a uniformly random vector of values does not alter the resulting distribution.

Since computational indistinguishability is both transitive[9] and reflexive, it remains only to show that the distributions produced by `expr` and `adversaryView expr` are computationally indistinguishable. We explain how to prove this in the remainder of this section.

We start by introducing the operation `removeOneKey`, which takes an expression `e` and a single key `k`, and replaces all occurrences of `Enc k X` with `Hidden k`. Next, we observe that we can reach `adversaryView expr` by repeatedly applying `removeOneKey` to `expr`, using keys that are not written in the expression—i.e., not returned by `extractKeys`. (Note that as we proceed, `extractKeys` might return fewer keys, making more keys available for removal.) Thanks to this observation, we can apply transitivity once again, reducing the proof to showing that if `k` $\notin$ `extractKeys expr`, then the distributions produced by `expr` and `removeOneKey expr k` are computationally indistinguishable.

In order to prove this, we make use of the following observation: Let $\mathtt{O}_1$ and $\mathtt{O}_2$ be two computationally indistinguishable families of seeded oracles, and let `F` be a family of oracle computations, then the distributions $\mathtt{D}_1$ `= compToDistrGen` $\mathtt{O}_1$ `F` and $\mathtt{D}_2$ `= compToDistrGen` $\mathtt{O}_2$ `F` are also computationally indistinguishable. Indeed, if $\mathtt{D}_1$ and $\mathtt{D}_2$ were distinguishable, then the adversary could distinguish the oracles $\mathtt{O}_1$ and $\mathtt{O}_2$ by first running `F` and then the simple computation that distinguishes $\mathtt{D}_1$ and $\mathtt{D}_2$. (Here we use the axiom that polynomial time oracle computations are closed under compositions.) This observation is formalized as `lemma reduction` in `ComputationalIndistinguishability/Lemmas.lean`.

Equipped with this observation, we can now show that `expr` and `removeOneKey expr k` are indistinguishable, provided that `k`$\notin$`extractKeys expr`. For this, we will define a family of oracle computations `reductionHidingOneKey` that, when run on the oracles $\mathcal{O}_L$ and $\mathcal{O}_R$ from the definition of IND-CPA security, will produce respectively `expr` and `removeOneKey expr k`. Most of the time, `reductionHidingOneKey` behaves like the computational semantics (i.e. `exprToDistr`). However, when it encounters an expression encrypted with the key that is being removed (i.e. `Enc k X`), it does not encrypt it directly, but instead sends the following pair of messages to the oracles:

9 This is formalized as the lemma `indTrans` in `ComputationalIndistinguishability/Lemmas.lean`. Observe that, since computational indistinguishability is defined on κ-indexed families of distributions, the type system prevents us from applying this lemma an unbounded, κ-dependent number of times. This is, of course, a desirable property, as otherwise the lemma would be false.

(Bit vector corresponding to `X`, Bit vector of 1's)

The Lean definition of `reductionHidingOneKey` is given in `Expression/ComputationalSemantics/SoundnessProof/HidingOneKey.lean` and its pen-and-paper complexity analysis required by the second axiom of polynomial time is provided as a comment at the end of the Lean file (and also in the full version of this paper).

Observe that the left oracle $\mathtt{O}_l$ will return the encryption of `X` with a uniformly random key, which corresponds to the semantics of `Enc k X`. On the other hand, the right oracle $\mathtt{O}_r$ will return the encryption of a vector of ones, which corresponds to the semantics of `Hidden k`. Moreover, both oracles will consistently use the same key `k` (chosen uniformly at random) to answer all the queries. (It is important that `k` $\notin$ `extractKeys expr`, as we cannot extract `k` from the oracles, so we would not be able to output it.) It follows that `compToDistrGen` $\mathtt{O}_l$ `reductionHidingOneKey` will produce the same distribution as `expr`, and `compToDistrGen` $\mathtt{O}_r$ `reductionHidingOneKey` will produce the same distribution as `removeOneKey expr k`. Since the oracles are indistinguishable (by the IND-CPA assumption), we can conclude that `expr` and `removeOneKey expr k` result in computationally indistinguishable distributions (as long as `k` $\notin$ `extractKeys expr`), thus concluding the proof of the computational soundness theorem.

5.6 CONCLUSION AND FUTURE WORK

We have presented a formalization of the symbolic cryptography framework using Lean, including a computational soundness theorem for symbolic indistinguishability. We have also demonstrated that this framework can be used to prove security of circuit garbling – a fairly complex cryptographic protocol. We hope to have convinced the reader that the symbolic framework offers a clean, intuitive, and aesthetically pleasing way to reason about cryptographic properties. Below, we outline a few directions for further development:

PROVING MORE PROPERTIES The most immediate next step would be to prove the security of more cryptographic protocols using the symbolic framework. One exciting direction is to formalize the security of optimization techniques for garbled circuits, such as *free XOR* [63][10] and *row reduction* [84]. However, these techniques rely on security assumptions beyond IND-CPA and operations that go beyond our current symbolic expression language. They would, therefore, require a major redesign of the symbolic framework, making the endeavor particularly challenging.

EXTENDING THE LANGUAGE OF CRYPTOGRAPHIC EXPRESSIONS This leads to a second direction: extending the symbolic expression

10 We thank the anonymous reviewer for pointing us to [96], which presents an obstacle to modeling the XOR operation in a computationally sound symbolic framework. After an initial reading, we hope that our symbolic framework avoids this pitfall due to its non-interactive nature, but we leave a thorough investigation of this question to future work.

language to support additional cryptographic primitives, such as pseudo-random generators or key xor operations. A pen-and-paper version of a computationally sound symbolic framework with pseudo-random generators already appears in [67], and it would be a natural next step to incorporate this into our Lean formalization.

A CONCRETE DEFINITION OF POLYNOMIAL TIME While the axiomatic approach to formalizing polynomial time is clean and simple, it places the burden on the reader to verify that the axioms (including specific reductions) are valid. A more robust alternative would be to adopt a concrete definition of polynomial time. The calculi proposed in [9] and [68] appear promising in this regard. However, applying them in our framework seems very challenging and would probably outweigh its other parts. It is, however, possible that further developments in the Lean (or other proof assistants) ecosystem will make this easier in the future.

BIBLIOGRAPHY

[1] Martín Abadi and Phillip Rogaway. "Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)." In: *J. Cryptology* 20.3 (2007), p. 395.

[2] Martín Abadi and Bogdan Warinschi. "Security analysis of cryptographically controlled access to XML documents." In: *J. ACM* 55.2 (2008), 6:1–6:29. DOI: 10.1145/1346330.1346331.

[3] Tolga Acar, Mira Belenkiy, Mihir Bellare, and David Cash. "Cryptographic Agility and Its Relation to Circular Encryption." In: *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 403–422. DOI: 10.1007/978-3-642-13190-5_21. URL: https://doi.org/10.1007/978-3-642-13190-5_21.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Implementations for Secure and Verifiable Computation." In: *IACR Cryptol. ePrint Arch.* (2014), p. 456. URL: http://eprint.iacr.org/2014/456.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. "A Fast and Verified Software Stack for Secure Function Evaluation." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 1989–2006. DOI: 10.1145/3133956.3134017.

[6] José Bacelar Almeida, Manuel Barbosa, Manuel L. Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. "Machine-checked ZKP for NP relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head." In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 2587–2600. DOI: 10.1145/3460120.3484771.

[7] Monika di Angelo and Gernot Salzer. "A Survey of Tools for Analyzing Ethereum Smart Contracts." In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 2019, pp. 69–78. DOI: 10.1109/DAPPCON.2019.00018.

[8] N. Asokan, Matthias Schunter, and Michael Waidner. "Optimistic Protocols for Fair Exchange." In: *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*. Ed. by Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong. ACM, 1997, pp. 7–17. DOI: 10.1145/266420.266426.

[9] Robert Atkey. "Polynomial Time and Dependent Types." In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 2288–2317. DOI: 10.1145/3632918. URL: https://doi.org/10.1145/3632918.

[10] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "A Formal Model of Bitcoin Transactions." In: *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*. Ed. by Sarah Meiklejohn and Kazue Sako. Vol. 10957. Lecture Notes in Computer Science. Springer, 2018, pp. 541–560. DOI: 10.1007/978-3-662-58387-6_29. URL: https://doi.org/10.1007/978-3-662-58387-6_29.

[11] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Joseph Lallemand. "The Squirrel Prover and its Logic." In: *ACM SIGLOG News* 11.2 (2024), pp. 62–83. DOI: 10.1145/3665453.3665461. URL: https://doi.org/10.1145/3665453.3665461.

[12] Bolton Bailey and Andrew Miller. "Formalizing Soundness Proofs of Linear PCP SNARKs." In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bailey.

[13] Gergei Bana and Hubert Comon-Lundh. "Towards Unconditional Soundness: Computationally Complete Symbolic Attacker." In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Vol. 7215. Lecture Notes in Computer Science. Springer, 2012, pp. 189–208. DOI: 10.1007/978-3-642-28641-4_11. URL: https://doi.org/10.1007/978-3-642-28641-4_11.

[14] Gergei Bana and Hubert Comon-Lundh. "A Computationally Complete Symbolic Attacker for Equivalence Properties." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM, 2014, pp. 609–620. DOI: 10.1145/2660267.2660276. URL: https://doi.org/10.1145/2660267.2660276.

[15] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification - 23rd*

International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1_14.

[16] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella-Béguelin. "Computer-Aided Cryptographic Proofs." In: *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. Ed. by Lennart Beringer and Amy P. Felty. Vol. 7406. Lecture Notes in Computer Science. Springer, 2012, pp. 11–27. DOI: 10.1007/978-3-642-32347-8_2. URL: https://doi.org/10.1007/978-3-642-32347-8_2.

[17] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer." In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 2011, pp. 71–90. DOI: 10.1007/978-3-642-22792-9_5.

[18] Mathieu Baudet, Bogdan Warinschi, and Martín Abadi. "Guessing attacks and the computational soundness of static equivalence." In: *Journal of Computer Security* 18.5 (2010), pp. 909–968. DOI: 10.3233/JCS-2009-0386.

[19] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. "Foundations of garbled circuits." In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 784–796. DOI: 10.1145/2382196.2382279. URL: https://doi.org/10.1145/2382196.2382279.

[20] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. "The Hunting of the SNARK." In: *J. Cryptol.* 30.4 (2017), pp. 989–1066. DOI: 10.1007/S00145-016-9241-9.

[21] Bruno Blanchet. "A Computationally Sound Mechanized Prover for Security Protocols." In: *IEEE Symposium on Security and Privacy*. Oakland, California, May 2006, pp. 140–154.

[22] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif." In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, 2013, pp. 54–87. DOI: 10.1007/978-3-319-10082-1_3.

[23] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd Your Herd of Provers." In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. https://hal.inria.fr/hal-00790310. Wrocław, Poland, Aug. 2011, pp. 53–64.

[24] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. "Tendermint Blockchain Synchronization: Formal Specification and Model Checking." In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2020, pp. 471–488. ISBN: 978-3-030-61362-4.

[25] Vitalik Buterin. *A Note on Limits on Incentive Compatibility and Griefing Factors*. URL: https://vitalik.ca/files/extortion_griefing_bounds.pdf.

[26] Vitalik Buterin. *Scalability, Part 2: Hypercubes*. URL: https://blog.ethereum.org/2014/10/21/scalability-part-2-hypercubes/.

[27] Vitalik Buterin. *Minimal Viable Plasma*. 2018. URL: https://ethresear.ch/t/minimal-viable-plasma.

[28] Vitalik Buterin. *Plasma Cash: Plasma with much less per-user data checking*. 2018. URL: https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298.

[29] Vitalik Buterin. *The Dawn of Hybrid Layer 2 Protocols*. https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html. (Accessed on 02/08/2020). Aug. 2019.

[30] Christian Cachin and Jan Camenisch. "Optimistic Fair Secure Computation." In: *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*. Ed. by Mihir Bellare. Vol. 1880. Lecture Notes in Computer Science. Springer, 2000, pp. 93–111. DOI: 10.1007/3-540-44598-6_6. URL: https://doi.org/10.1007/3-540-44598-6_6.

[31] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols." In: *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. IEEE Computer Society, 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[32] Ran Canetti. "Universally Composable Signature, Certification, and Authentication." In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 2004, p. 219. DOI: 10.1109/CSFW.2004.24. URL: https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.24.

[33] Shu-Jie Cao and Dongning Guo. "Security, Latency, and Throughput of Proof-of-Work Nakamoto Consensus." In: *IEEE Trans. Inf. Theory* 71.6 (2025), pp. 4708–4731. DOI: 10.1109/TIT.2025.3557761. URL: https://doi.org/10.1109/TIT.2025.3557761.

[34] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. "On the Security of the "Free-XOR" Technique." In: *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*. Ed. by Ronald Cramer. Lecture Notes in Computer Science. Springer, 2012, pp. 39–53. DOI: 10.1007/978-3-642-28914-9_3. URL: https://doi.org/10.1007/978-3-642-28914-9_3.

[35] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. "Alt-Ergo 2.2." In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. Oxford, United Kingdom, July 2018. URL: https://inria.hal.science/hal-01960203.

[36] RJ Corin. "Computational soundness of formal encryption in coq." In: *3rd Workshop on Formal and Computational Cryptography, FCC 2007*. INRIA. 2007.

[37] Kyle Croman et al. "On Scaling Decentralized Blockchains." In: *Financial Cryptography and Data Security*. Ed. by Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125. ISBN: 978-3-662-53357-4.

[38] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. "FairSwap: How To Fairly Exchange Digital Goods." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 967–984. DOI: 10.1145/3243734.3243857.

[39] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. "Multi-party Virtual State Channels." In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. Lecture Notes in Computer Science. Springer, 2019, pp. 625–656. DOI: 10.1007/978-3-030-17653-2_21. URL: https://doi.org/10.1007/978-3-030-17653-2_21.

[40] Stefan Dziembowski, Grzegorz Fabiański, Sebastian Faust, and Siavash Riahi. *Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma*. Cryptology ePrint Archive, Report 2020/175. https://eprint.iacr.org/2020/175. 2020.

[41] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. "General State Channel Networks." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 949–966. DOI: 10.1145/3243734.3243856.

[42] Karim Eldefrawy and Vitor Pereira. "A High-Assurance Evaluator for Machine-Checked Secure Multiparty Computation." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 851–868. DOI: 10.1145/3319535.3354205.

[43] David Evans, Vladimir Kolesnikov, and Mike Rosulek. "A Pragmatic Introduction to Secure Multi-Party Computation." In: *Found. Trends Priv. Secur.* 2.2-3 (2018), pp. 70–246. DOI: 10.1561/3300000019. URL: https://doi.org/10.1561/3300000019.

[44] Shimon Even, Oded Goldreich, and Abraham Lempel. "A Randomized Protocol for Signing Contracts." In: *Commun. ACM* 28.6 (1985), pp. 637–647. DOI: 10.1145/3812.3818.

[45] Grzegorz Fabiański, Rafał Stefański, and Orfeas Stefanos Thyfronitis Litos. *A Formally Verified Lightning Network*. 2025. arXiv: 2503.07200 [cs.CR]. URL: https://arxiv.org/abs/2503.07200.

[46] Georg Fuchsbauer, Michele Orrù, and Yannick Seurin. "Aggregate Cash Systems: A Cryptographic Investigation of Mimblewimble." In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. Lecture Notes in Computer Science. Springer, 2019, pp. 657–689. DOI: 10.1007/978-3-030-17653-2_22. URL: https://doi.org/10.1007/978-3-030-17653-2_22.

[47] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications." In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10. URL: https://doi.org/10.1007/978-3-662-46803-6_10.

[48] Lior Goldberg and Oren Katz. *StarkDEX Deep Dive: Contracts & Statement - StarkWare - Medium*. https://medium.com/starkware/tagged/starkdex-specs. (Accessed on 02/08/2020). 2019.

[49] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[50] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption." In: *J. Comput. Syst. Sci.* 28.2 (1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9.

[51] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof Systems." In: *SIAM J. Comput.* 18.1 (1989), pp. 186–208. DOI: 10.1137/0218012.

[52] Matthias Grundmann and Hannes Hartenstein. "Verifying Payment Channels with TLA+." In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022, pp. 1–3. DOI: 10.1109/ICBC54727.2022.9805487.

[53] Matthias Grundmann and Hannes Hartenstein. *Towards a Formal Verification of the Lightning Network with TLA+*. 2023. arXiv: 2307.02342 [cs.LO].

[54] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. "SoK: Layer-Two Blockchain Protocols." In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*. Ed. by Joseph Bonneau and Nadia Heninger. Vol. 12059. Lecture Notes in Computer Science. Springer, 2020, pp. 201–226. DOI: 10.1007/978-3-030-51280-4_12. URL: https://doi.org/10.1007/978-3-030-51280-4_12.

[55] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. "Computer-Aided Proofs for Multiparty Computation with Active Security." In: *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 2018, pp. 119–131. DOI: 10.1109/CSF.2018.00016.

[56] Stuart Haber and W. Scott Stornetta. "How to Time-Stamp a Digital Document." In: *J. Cryptology* 3.2 (1991), pp. 99–111. DOI: 10.1007/BF00196791.

[57] Everett Hildenbrandt et al. "KEVM: A Complete Semantics of the Ethereum Virtual Machine." In: *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[58] Russell Impagliazzo and Steven Rudich. "Limits on the Provable Consequences of One-Way Permutations." In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. Ed. by David S. Johnson. ACM, 1989, pp. 44–61. DOI: 10.1145/73007.73012.

[59] Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design." In: *14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22-23 June, 2015*. 2015. URL: http://www.econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf.

[60] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[61] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Report 2018/642. https://eprint.iacr.org/2018/642. 2018.

[62] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. "A Composable Security Treatment of the Lightning Network." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 778. URL: https://eprint.iacr.org/2019/778.

[63] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications." In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*. Ed. by Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 486–498. DOI: 10.1007/978-3-540-70583-3_40. URL: https://doi.org/10.1007/978-3-540-70583-3_40.

[64] Georgios Konstantopoulos. *Plasma Cash: Towards more efficient Plasma constructions*. 2019. URL: https://www.gakonst.com/plasmacash.pdf.

[65] Vladimir Kukharenko, Kirill Ziborov, Rafael Sadykov, and Ruslan Rezin. "Verification of HotStuff BFT Consensus Protocol with TLA+/TLC in an Industrial Setting." In: *Informatics and Cybernetics in Intelligent Systems*. Ed. by Radek Silhavy. Cham: Springer International Publishing, 2021, pp. 77–95. ISBN: 978-3-030-77448-6.

[66] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591. ISBN: 978-3-642-22110-1.

[67] Baiyu Li and Daniele Micciancio. "Symbolic security of garbled circuits." In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 147–161.

[68] Kevin Liao, Matthew A. Hammer, and Andrew Miller. "ILC: a calculus for composable, computational cryptography." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 640–654. DOI: 10.1145/3314221.3314607. URL: https://doi.org/10.1145/3314221.3314607.

[69] Yehuda Lindell and Benny Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation." In: *J. Cryptology* 22.2 (2009), pp. 161–188. DOI: 10.1007/s00145-008-9036-8.

[70] Łukasz Mazurek. "EthVer: Formal Verification of Randomized Ethereum Smart Contracts." In: *Financial Cryptography and Data Security. FC 2021 International Workshops*. Ed. by Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 364–380. ISBN: 978-3-662-63958-0.

[71] Markus de Medeiros, Muhammad Naveed, Tancrède Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetzsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. "Verified Foundations for Differential Privacy." In: *IACR Cryptol. ePrint Arch.* (2024), p. 2040. URL: https://eprint.iacr.org/2024/2040.

[72] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols." In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[73] Daniele Micciancio. "Computational Soundness, Co-induction, and Encryption Cycles." In: *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*. 2010, pp. 362–380.

[74] Daniele Micciancio and Saurabh Panjwani. "Corrupting One vs. Corrupting Many: The Case of Broadcast and Multicast Encryption." In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*. 2006, pp. 70–82. DOI: 10.1007/11787006_7.

[75] Daniele Micciancio and Saurabh Panjwani. "Optimal communication complexity of generic multicast key distribution." In: *IEEE/ACM Trans. Netw.* 16.4 (2008), pp. 803–813. DOI: 10.1145/1453698.1453703.

[76] Rajarshi Mitra. *Plasma Breakthrough: OmiseGO (OMG) announces the launch of Ari*. https://www.fxstreet.com/cryptocurrencies/news/plasma-breakthrough-omisego-omg-announces-the-launch-of-ari-201904120245. (Accessed on 02/08/2020).

[77] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1186–1189. DOI: 10.1109/ASE.2019.00133.

[78] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language." In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André

Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37.

[79] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: http://bitcoin.org/bitcoin.pdf.

[80] Moni Naor and Benny Pinkas. "Efficient oblivious transfer protocols." In: *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. Ed. by S. Rao Kosaraju. ACM/SIAM, 2001, pp. 448–457. URL: http://dl.acm.org/citation.cfm?id=365411.365502.

[81] Saurabh Panjwani. "Tackling Adaptive Corruptions in Multicast Encryption Protocols." In: *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*. 2007, pp. 21–40.

[82] Daejun Park, Yi Zhang, and Grigore Rosu. "End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract." In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 151–164. ISBN: 978-3-030-53288-8.

[83] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. "A Framework for Efficient and Composable Oblivious Transfer." In: *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*. Ed. by David A. Wagner. Vol. 5157. Lecture Notes in Computer Science. Springer, 2008, pp. 554–571. DOI: 10.1007/978-3-540-85174-5_31.

[84] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation Is Practical." In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 250–267. DOI: 10.1007/978-3-642-10366-7_15. URL: https://doi.org/10.1007/978-3-642-10366-7_15.

[85] Joseph Poon and Vitalik Buterin. *Plasma: Scalable Autonomous Smart Contracts*. 2017. URL: http://plasma.io/plasma.pdf.

[86] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. https://lightning.network/lightning-network-paper.pdf. Jan. 2016.

[87] Michael O. Rabin. "How To Exchange Secrets with Oblivious Transfer." In: *IACR Cryptol. ePrint Arch.* (2005), p. 187. URL: http://eprint.iacr.org/2005/187.

[88] Ron Rothblum. "On the Circular Security of Bit-Encryption." In: *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*. Ed. by Amit Sahai. Vol. 7785. Lecture Notes in Computer Science. Springer, 2013, pp. 579–598. DOI: 10.1007/978-3-642-36594-2_32. URL: https://doi.org/10.1007/978-3-642-36594-2_32.

[89] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Nusret Tas, and David Tse. "Three Attacks on Proof-of-Stake Ethereum." In: *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*. Ed. by Ittay Eyal and Juan A. Garay. Lecture Notes in Computer Science. Springer, 2022, pp. 560–576. DOI: 10.1007/978-3-031-18283-9_28. URL: https://doi.org/10.1007/978-3-031-18283-9_28.

[90] Wouter Swierstra. "Data types à la carte." In: *J. Funct. Program.* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758. URL: https://doi.org/10.1017/S0956796808006758.

[91] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. Extropy Magazine. 1996.

[92] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. "Blockchain Scaling Using Rollups: A Comprehensive Survey." In: *IEEE Access* 10 (2022), pp. 93039–93054. DOI: 10.1109/ACCESS.2022.3200051.

[93] Anqi Tian, Peifang Ni, Yong Liu, and Liang Huang. "Blockchain-Based Payment Channel Network: Challenges and Recent Advances." In: *2023 International Conference on Blockchain Technology and Information Security (ICBCTIS)*. 2023, pp. 187–194. DOI: 10.1109/ICBCTIS59921.2023.00036.

[94] Trustnodes. *Ethereum Transactions Fall Off the Cliff, Three Plasma Projects Close to Release Says Buterin*. 2018. URL: https://www.trustnodes.com/2018/07/05/ethereum-transactions-fall-off-cliff-three-plasma-projects-close-release-says-buterin.

[95] Devon Tuma and Nicholas Hopper. "VCVio: A Formally Verified Forking Lemma and Fiat-Shamir Transform, via a Flexible and Expressive Oracle Representation." In: *IACR Cryptol. ePrint Arch.* (2024), p. 1819. URL: https://eprint.iacr.org/2024/1819.

[96] Dominique Unruh. "The impossibility of computationally sound XOR." In: *Cryptology ePrint Archive* (2010).

[97] Wikipedia. *Trusted timestamping*. URL: https://en.wikipedia.org/wiki/Trusted_timestamping.

[98] Joon Ian Wong. *The ethereum network is getting jammed up because people are rushing to buy cartoon cats on its blockchain*. Quartz. 2017. URL: https://qz.com/1145833/cryptokitties-is-causing-ethereum-network-congestion/.

[99] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. 2016. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[100] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. "LeanDojo: Theorem Proving with Retrieval-Augmented Language Models." In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine. 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

[101] Zhe Yang, Meiyi Dai, and Jian Guo. "Formal Modeling and Verification of Smart Contracts with Spin." In: *Electronics* 11.19 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11193091. URL: https://www.mdpi.com/2079-9292/11/19/3091.

[102] Andrew Chi-Chih Yao. "Protocols for Secure Computations (Extended Abstract)." In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 1982, pp. 160–164.