SECURE MULTI-PARTY COMPUTATION IN PRACTICE

Marcella Christine Hastings

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

ACKNOWLEDGMENT

This dissertation would have been much less pleasant to produce without the presence of many people in my life. I would like to thank my advisor and my dissertation committee for their helpful advice, direction, and long-distance phone calls over the past six years.

I would like to thank my fellow PhD students at the University of Pennsylvania, especially the ever-changing but consistently lovely office mates in the Distributed Systems Laboratory and my cohort. Our shared tea-time, cookies, disappointments, and achievements provided an essential community that brought me great joy during my time at Penn.

I would like to thank the mentors and colleagues who hosted me at the Security and Privacy Lab at the University of Washington in 2018, the Software & Application Innovation Lab at Boston University in 2019, and the Cryptography and Privacy Research group at Microsoft Research in 2020. My career and happiness greatly benefited from spending these summers exploring fresh research directions and rediscovering the world outside my own work.

I would like to thank profusely my colleagues at Bolt Labs, who have been relentlessly supportive of my professional goals throughout our collaboration, and who provided mentorship and direction at a time in my PhD when I was particularly uncertain.

I would like to thank my family and friends who insisted on regular updates on my research progress, enthusiastically attended events typically reserved for professional colleagues, and reassured me that our relationships were not conditional on the number of degrees I held.

Finally, I would like to thank my partner, who has simply been the best.

ABSTRACT

SECURE MULTI-PARTY COMPUTATION IN PRACTICE

Marcella Christine Hastings

Nadia Heninger

Secure multi-party computation (MPC) is a cryptographic primitive for computing on private data. MPC provides strong privacy guarantees, but practical adoption requires high-quality application design, software development, and resource management. This dissertation aims to identify and reduce barriers to practical deployment of MPC applications.

First, the dissertation evaluates the design, capabilities, and usability of eleven state-of-the-art MPC software frameworks. These frameworks are essential for prototyping MPC applications, but their qualities vary widely; the survey provides insight into their current abilities and limitations. A comprehensive online repository augments the survey, including complete build environments, sample programs, and additional documentation for each framework.

Second, the dissertation applies these lessons in two practical applications of MPC. The first addresses algorithms for assessing stability in financial networks, traditionally designed in a full-information model with a central regulator or data aggregator. This case study describes principles to transform two such algorithms into data-oblivious versions and benchmark their execution under MPC using three frameworks. The second aims to enable unlinkability of payments made with blockchain-based cryptocurrencies. This study uses MPC in conjunction with other privacy techniques to achieve unlinkability in payment channels. Together, these studies illuminate the limitations of existing software, develop guidelines for transforming non-private algorithms into versions suitable for execution under MPC, and illustrate the current practical feasibility of MPC as a solution to a wide variety of applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

From the first introduction of secure multi-party computation in the 1980s, theorists have developed robust, efficient algorithms; researchers have implemented optimized software frameworks; and newly identified applications abound in the real world. However, connecting these threads into effective, privacy-preserving deployments reveals a new set of challenges. This dissertation aims to identify and solve some of these challenges to encourage wider-spread adoption of this useful, privacy-enhancing primitive.

Secure multi-party computation (MPC) is a cryptographic tool that allows a group of mutually distrustful parties to compute a function on their joint inputs without revealing anything beyond the output of the computation. This umbrella encompasses a wide variety of functions; some algorithms optimize for a specific function, like set intersection or signing. Others are capable of computing an *arbitrary* function. This dissertation focuses mainly on the latter category, trading off application-specific optimization for broad, reusable utility across many settings. However, this theoretic expressive capability requires a great deal of practical infrastructure before it can be used with real data. Lindell writes [Lin21]:

> "MPC still requires great expertise to deploy, and additional research breakthroughs are needed to make secure computation practical on large data sets and for complex problems, and to make it easy to use for nonexperts."

This dissertation explores the question of *why* MPC is difficult to deploy.

Although challenges to practical application exist, the academic community has successfully collaborated with practitioners to develop software for MPC applications: the first software framework implementing general-purpose secure computation was introduced in 2004 [MNPS04], and the first known deployment was a blind auction for Danish sugar beets [BCD$^+$09]. Recent applications illustrate a range of motivating factors for using privacy enhancing technologies. The ZCash foundation used MPC to compute verifiably random parameters for zk-SNARKS [BGM17, BD18]; the goal of the computation was to compute parameters with randomness that relies on only one of the 87 participants' honest behavior. The Boston Women's Workforce Council computed statistics on gender and racial wage gaps across Boston [LJDA$^+$18], without compromising on participants' concerns of liability for data leakage or negative publicity. Other proposed deployments [BJSV15, SvA$^+$19, RQA$^+$18] are motivated by similar concerns about the legal and security implications of hosting sensitive data and a general reluctance to identify a single trusted party to perform data aggregation and computation in the clear.

However, as Lindell suggests, many of these deployments required a team of expert cryptographers to develop a custom MPC protocol or implementation for their specific use case. Secure computation has been proposed as a solution for a wide range of practical applications, from general computations of secure statistical analysis [BKL$^+$14, BKLS14, BNTW12, DA01a, DA01b, DA01c, DHC04, LP09b], to more domain-specific uses like financial oversight [AKL12, BKK$^+$16, BTW12, FKOS13], biomedical computations [CWB18, CH15, HCB18, JKS08, JWB$^+$17, KBLV13, Zel14] and satellite collision detection [HLOI16, HWB14, KW14]. However, MPC has yet to be deployed successfully across this broad application spectrum. The work in this dissertation aims to identify specific barriers to adoption of MPC, develop tools and techniques for simplifying deployments by various types of practitioners, and present several novel applications with proof-of-concept deployment software.

1.1 Overview

This dissertation examines barriers to practical deployment of secure multi-party computation practical with three main studies: (1) analyzing the capabilities and maturity of software tools for MPC; (2) adapting existing, non-private algorithms to be efficient under MPC; and (3) designing and implementing a component-wise integration of MPC into a larger protocol. Overall, this dissertation aims to show that software analysis and practical case studies are effective methods for identifying limitations of current MPC frameworks, even as they demonstrate the feasibility of current technologies for practical use.

Chapter 2 introduces the cryptographic building blocks that underlie MPC protocols, surveys the main varieties of secure computation protocols, and outlines the simulation paradigm used in this dissertation to prove the security of protocols.

Chapter 3 surveys software frameworks for MPC. These *end-to-end* frameworks include a full toolchain for describing and securely executing a function among multiple parties. Development of such software is nontrivial; it requires implementation of cryptographic protocols, complex distributed systems, and language compilers that can translate high-level function descriptions into the simplified representations consumed by an MPC algorithm. The survey describes eleven end-to-end software frameworks, recording characteristics including threat and computational models; data types, operations, and data structures supported; and software architecture. It also defines a set of usability criteria based on documentation and software examples.

One finding of the survey is that simple usability issues are a major barrier to effective use of the software frameworks. A severe lack of documentation and complex software requirements to build and execute the tools introduce a significant up-front cost to a potential deployment. In order to reduce these issues, Chapter 3 is accompanied by an open-source software repository with complete build environments (Docker containers) and simple programs for each framework, along with additional documentation derived while using the tools. This work has significant impact: the repository has 272 stars and over 70 forks[1] and the paper [HHNZ19] has been cited more than 40 times in two years, illustrating widespread community interest and active use of these resources.

Implementing sample programs illuminated other features and limitations of the frameworks that would not have been obvious from simply reading their accompanying papers. The development process identified software bugs in several of the tools. In other cases, the capabilities and claims made in the papers seemed to require an incredibly deep understanding of the software, and lack of such knowledge prevented successful implementation of the sample programs for this work. This was particularly common in frameworks that integrated tools and techniques from other fields, *e.g.* formal guarantees about the custom programming language, or circuit compilers typically used for designing hardware circuits. Although some errors are to be expected with software produced in

[1]See `https://github.com/mpc-sok/frameworks`, as of 20 January 2020.

a academic-focused environment, this finding indicates that particular care must be taken as the complexity of the software increases.

Chapter 4 explores a use case drawn from the literature on financial networks. Existing work in this setting typically relies on a central data aggregator to collect information from each node in a network and execute some algorithm on it. This chapter centers on the problem of identifying and mitigating systemic risks in financial networks. The setting assumes a group of asset-owning participants, such as banks, who construct a network of liabilities among themselves. The existing literature [EGJ14, EN01] determines whether a network is *solvent* if all participants attempt to pay back their debts, but it requires each participant to share a great deal of potentially sensitive information about their institutional interdependencies with a central agent.

This chapter converts network analysis algorithms to data-oblivious versions suitable to execute under MPC. It describes implementations using three separate MPC software frameworks with different threat models and discusses the overall costs and feasibility of this approach to network analysis. This is one of the first practical implementations of such an algorithm and the first to measure realistic benchmarks. This work also presents two contributions that aim to simplify future MPC applications: it (1) defines a clear set of principles for transforming an arbitrary algorithm into an MPC-friendly representation and (2) explores the limitations of existing MPC software on complex algorithms. These contributions aim to solve problems for two groups of people. First, experts in non-cryptographic fields now have a clearer roadmap to scoping appropriate MPC applications within their domain. Second, MPC framework designers can consider this case study as they improve on existing frameworks. This application significantly exceeds the complexity of the sample programs from Chapter 3, with arithmetic and comparison operations over fixed-point numbers and complex matrix operations. This work identifies additional usability issues and efficiency bottlenecks that may generalize to other computations, and are worth addressing in future work on MPC frameworks.

Chapter 5 describes the design and implementation of a protocol for *unlinkable payment channels* that address privacy, scalability, and latency issues that arise in traditional cryptocurrencies and digital payment systems. The zkChannels protocol uses MPC in conjunction with other privacy technologies to achieve security and usability properties that cannot be efficiently implemented with MPC alone. A payment channel allows two parties to escrow funds on an existing blockchain and make payments directly to each other off-chain. Channels guarantee that honest parties cannot lose money with an on-chain arbitration process. The zkChannels protocol adds an unlinkability property to these guarantees: a merchant maintaining channels with a set of customers cannot link any given payment with a specific customer or to any other payment. The protocol is flexible with respect to the underlying arbiter and can be used to add additional privacy guarantees on top of almost any existing cryptocurrency.

zkChannels is a three-phase protocol: users interact with a cryptocurrency network to Establish and Close a channel, but only communicate among themselves to Pay. Payments use MPC to produce an updated closing transaction signed by both parties while maintaining privacy on their identities and signing keys. However, zkChannels includes other techniques to prevent double-spending, ensure that both parties can close on the correct balances, and prevent denial-of-service attacks.

This work implements a complex MPC computation with specific requirements, which may provide new challenges to framework designers. The complexity in this design arises from the practical requirements of the protocol; for example, the MPC solution must provide different outputs for different parties and efficiently support public input values. To maintain unlinkability, the setting does not allow participants to generate significant preprocessing data ahead of time, so any "function-independent" data must be efficient to compute. Further, the protocol integrates with other software components and requires a framework with a flexible development environment.

These chapters rely on software frameworks developed, for the most part, in academia (including [WMK16, AKR$^+$19, Sch]), Chapters 3 and 5 include open-source, proof-of-concept implementations of their applications. This software provide reasonable guarantees for academic work, but deployments on highly sensitive data require high quality software to ensure that the privacy guarantees of MPC are not compromised by vulnerabilities. Chapter 6 describes work on one such framework developed at Microsoft Research, which includes an extensive testing framework, significant documentation, and an integration into a more general platform for deploying secure computation. This chapter describes one attempt to break down a major barrier to practical adoption of secure computation: the lack of high-quality software suitable for use with private data.

Finally, Chapter 7 concludes with specific takeaways from the studies, illuminating concrete areas for improvement and future work.

1.2 Contributions

Chapter 3 is drawn largely from joint work with Brett Hemenway Falk, Daniel Noble and Steve Zdancewic [HHNZ19]. My contributions include design of the survey criteria, analysis and software implementations for 9 of 11 frameworks, and ongoing maintenance of the repository.

Chapter 4 is drawn largely from joint work with Brett Hemenway Falk and Gerry Tsoukalas [HHFT20]. My major research contributions include implementing the network algorithms in two software frameworks, benchmarking them, and discussing the advantages and limitations of each option.

Chapter 5 is drawn from joint work with J. Ayo Akinyele, Gabe Kaptchuk, Ian Miers, Colleen M. Swanson, Darius E. Parvin, and Gijs Van Laer [AGH$^+$21]. My major research contributions include designing, implementing, and benchmarking the MPC implementation, designing the $\mathcal{F}_{\mathsf{zkChannels}}$ ideal functionality, and refining and debugging the zkChannels protocol.

Chapter 6 is drawn from a summer 2020 internship project at Microsoft Research, where I was mentored by Hao Chen. I was supported in this work by Melissa Chase, Esha Ghosh, Radames Cruz Moreno, James Hill, and Kristin Lauter.

CHAPTER 2

Preliminaries

This chapter introduces the cryptographic underpinnings of MPC protocols, describes three basic classes of MPC protocol, and briefly covers the simulation paradigm used to describe the security of MPC. This background material provides useful context for the complexities of implementing protocols.

2.1 Cryptographic building blocks

This section describes secret sharing and oblivious transfer, techniques used to represent and share data in many MPC frameworks. Their MPC context is described in greater detail in Section 2.2. It also briefly covers the common threat models used to describe the security guarantees of cryptographic protocols.

2.1.1 Secret sharing

Cryptographic secret sharing protocols [Sha79, Bla79] allow a dealer to break a secret value into *shares* and distribute these shares to group of recipients with the property that any unqualified set of recipients learns nothing about the secret, while any qualified set of recipients can reconstruct the secret from their shares. In practice, most secret-sharing protocols are *threshold* protocols, where any collection of fewer than t shares reveals nothing about the secret – and any subset of size at least t can reconstruct the secret. Many general secret sharing schemes exist [BL90, Bri90, BI93, VD97, CF02], as well as constructions of secret-sharing schemes for general (non-threshold) access structures [Gen96, HM00]. See Beimel [Bei11] for a survey of secret-sharing protocols.

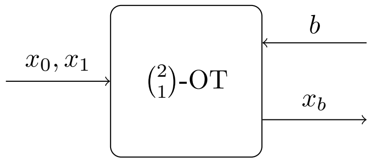


Figure 2.1: Oblivious Transfer

In practice, most MPC protocols rely on a linear secret sharing scheme: either simple, additive secret sharing, Shamir sharing, or replicated secret sharing. Each satisfies linearity: the sum of two secret shares is equal to the share of the sum.

2.1.2 Oblivious transfer (OT)

Oblivious Transfer (OT) [Rab81, Wie83, EGL82] is a cryptographic protocol that allows a party to choose k of n secrets from another party, without revealing which secret they have chosen. Figure 2.1 shows $\binom{2}{1}$-OT, where one secret is chosen from two options.

From a theoretical standpoint, MPC protocols can be built from OT alone [Kil88, IPS08], but the key feature that makes OT suitable for building *efficient* MPC protocols is that OT is equivalent to a seemingly weaker functionality called *Random OT* (ROT) [Cré88], where the choice bit b is not provided as an input, but instead is randomly generated by the protocol itself. The output of a ROT protocol is two correlated pairs of bits (x_0, x_1) and (b, x_b), where x_0, x_1, b are uniformly distributed in $\{0, 1\}$. Given a random OT correlation (the pairs (x_0, x_1) and (b, x_b)) Alice and Bob can compute the OT functionality using only three bits of communication.

Since ROT implies OT, parties can compute all the necessary ROTs needed for a protocol in advance, in an input-independent pre-processing ("offline") phase. Then, in the "online" phase, they consume these pre-generated OTs and execute the protocol with minimal communication cost and no computationally expensive public-key operations. This offline-online separation makes the online phase of such a protocol extremely efficient, but there is still the problem of making the pre-processing phase efficient. There are two fundamentally different approaches to handling the offline phase, either through a *trusted dealer* or a cryptographic batched correlation-generation protocol.

In the trusted dealer model, a semi-trusted dealer simply distributes correlated randomness to all the parties. The trusted dealer has no inputs, and never handles any secret information, thus the dealer need only be trusted to generate and distribute random values to the appropriate parties. In the presence of a trusted dealer, the offline phase of many MPC protocols can be extremely efficient.

Even without a trusted dealer, OTs can be generated efficiently through *OT extension*, where a small number of "base" or "seed" OTs are converted into a massive number of ROTs [IKNP03] through the use of efficient symmetric-key primitives (e.g. AES). Since its introduction, there have been many variants of OT-extension [ALSZ13, IPS08, HIKN08, NNOB12, Lar15, KOS15] and OT-extension has become an essential feature in almost all MPC implementations.

2.1.3 Threat models

MPC most commonly deals with two types of adversary. *Semi-honest* participants, also known as passive or honest-but-curious adversaries, execute the protocol correctly but attempt to glean additional information from the input and messages they receive. A *malicious* or active adversary may break the protocol arbitrarily in order to learn information about other inputs or to cause the protocol to output an incorrect result. Some schemes also describe *covert* adversaries, who behave arbitrarily but wish to avoid revealing themselves as an adversary. For example, a covert participant would avoid aborting or sending obviously invalid messages during a protocol execution. However, the protocols and implementations in this dissertation do not use this model.

The adversary model is independent of the number of parties that can be corrupted. An honest majority requires that fewer than half of parties are corrupt; a dishonest majority allows up to $n-1$ corruptions (in the number of participants n). If multiple parties are corrupted (for either semi-honest or malicious adversaries), they may pool their data to attempt to learn additional information.

2.2 Secure multi-party computation

Secure multi-party computation protocols allow a group of mutually distrustful stakeholders to securely compute *any* function of their joint inputs in such a way that the execution of the protocol provably reveals nothing beyond the result of the computation. Security is often defined using a *simulation paradigm*, described in Section 2.3. In this model, MPC cryptographically emulates a trusted party who accepts each participant's private input, computes the desired function, and returns the result.

Early MPC protocols used a circuit model for secure computation, first representing the target function as either a Boolean or arithmetic circuit (over some finite field), then securely evaluating the circuit gate-by-gate. Many software frameworks still use this circuit model of computation, but others have expanded to a more robust model. In the remainder of this section, we sketch key design characteristics of the major protocol families that underpin modern MPC systems.

2.2.1 Garbled circuits

Circuit garbling is a method for secure two-party computation, originally introduced by Yao [Yao82, Yao86]. In this framework, there are two participants, a *garbler* and an *evaluator*. The participants begin by expressing the desired function as a *Boolean circuit*. The garbler then proceeds to garble the circuit gate-by-gate using a standard symmetric-key cryptosystem (usually AES), as follows. For each wire in the circuit, the garbler selects two uniformly random and independent "wire tokens." The garbler then expresses each gate in the circuit as a truth table by encrypting each output wire token with the two input wire tokens that generate it (for Boolean circuits with fan-in two, each truth table will have four rows). The garbler permutes the rows of the truth-table, and sends the entire collection of "garbled" gates to the evaluator. The garbling procedure is designed so that learning one wire token for each input wire of a gate allows you to decrypt exactly one row of its garbled truth-table, revealing a single wire token belonging to the output-wire of that gate. Thus an evaluator that learns a single wire token for each input wire of the circuit can iteratively produce the wire token for each wire in the circuit, and completely evaluate the circuit.

To provide its secret inputs, the garbler sends the correct wire tokens directly to the evaluator. For each bit of the evaluator's inputs, the garbler and evaluator engage in an oblivious transfer where the evaluator secretly selects one of two wire tokens offered by the garbler. Once the evaluator has a wire token for each input bit, it can evaluate the circuit (performing symmetric-key decryptions for every gate) and learn the result. In the semi-honest model, the evaluator forwards the result of the computation to the garbler. For a formal description of the garbling procedure, see Bellare *et al.* [BHR12].

The initial garbled circuit protocol provided security against semi-honest adversaries [LP09a], but there exist many different improvements and implementations that provide security against fully malicious adversaries (*e.g.* [LR14, Lin13, WRK17a, WMK17, LR15]. Similarly, the vast majority of garbled circuit protocols are for two parties, but recent work has expanded support to three or more parties [LPSY15, MRZ15, LSS16, WRK17b].

Two key performance improvements are the “free XOR trick” [KS08], which evaluates XOR gates in a single round without any cryptographic operations required by the garbled tables; and Half-Gates [ZRE15], which reduce the number of ciphertexts required to garble AND gates. The addition of the AES-NI instruction set made computing AES encryptions on modern processors extremely efficient, and has dramatically reduced the computation cost (but not the communication cost) of garbled-circuit-based protocols.

Key Features: Circuit garbling is typically (but not exclusively) a two-party protocol, and requires only a constant number of rounds of communication (independent of the circuit depth). The number of (expensive) public-key operations depends only on the *input size* (OT is a public-key primitive) and the number of private key operations depends on the number of gates. The total communication cost is proportional to the size of the circuit. Since garbled circuit protocols represent each gate by its truth table, the circuit size grows quadratically with the field size when garbling arithmetic circuits. Thus almost all garbled circuits protocols operate on Boolean circuits. There are different methods for garbling arithmetic circuits over large fields [AIK14], but these have never been implemented.

2.2.2 Multi-party circuit-based protocols

The GMW [GMW87], BGW [BGW88] and CCD [CCD88] protocols allow an arbitrary number of parties to securely compute a function represented as a circuit. In these protocols, each party uses a linear secret sharing scheme to share its input, and the parties engage in a protocol to compute the answer gate-by-gate. Each gate computation securely transforms secret-shares of the gate’s inputs to secret-shares of the gate’s outputs. For each addition gate in the circuit, participants can locally compute shares of the output using the linearity of the secret-sharing scheme. Evaluating multiplication gates requires communication, and the schemes differ in how they handle multiplication.

The GMW protocol can evaluate either Boolean or arithmetic circuits, and multiplication gates are executed using Oblivious Transfer for Boolean circuits and with Oblivious Polynomial Evaluation [NP06] or Oblivious Linear Evaluation [DGN$^+$17] for arithmetic circuits. See Ishai *et al.* [IPS09] for a summary of methods for securely computing multiplication gates.

Oblivious communications for multiplication gates dominate the cost of circuit evaluation. All practical GMW-based implementations have taken steps to reduce their overhead. Whether evaluating arithmetic or Boolean circuits, the approach is the same: in an offline pre-computation phase, the participants generate correlated randomness (or receive it from a trusted dealer), and in the online phase, they use these random correlations as masks, or one-time-pads, to compute shares of the output of a multiplication gate based on the shares of the inputs.

Boolean-circuit GMW-based protocols use OT-extension [IKNP03] to pre-compute ROT correlations, which are then consumed in the online phase of the protocol. Arithmetic-circuit GMW-based protocols usually generate some form of "Beaver Multiplication Triples" (secret shares of random triples $(a, b, a \cdot b)$, where a, b are field elements) [Bea92] that are used as masks in the online phase.

Information-theoretic protocols, like BGW [GMW87] and CCD [CCD88] rely on secret-sharing schemes supporting *strong multiplication* [CDM00, CC06, PCCX09, CDN15] rather than on public-key primitives. These protocols can be faster, since they do not require computationally expensive public-key operations, but require an honest majority of participants. They generally do not benefit from including a pre-computation phase of the protocol.

Key Features: Multi-party circuit-based protocols can support an arbitrary number of participants. The number of rounds of communication is proportional to the multiplicative depth of the circuit, and the total amount of communication depends on the number of multiplication gates in the circuit. These protocols allow independent computation parties to receive input and pass output to other parties without compromising security.

2.2.3 Hybrid models

Recent systems have moved away from a strict circuit representation and instead use a hybrid model, where functions compile into a set of optimized subprotocols for common operations. This set of primitives, which may include traditional circuit-based operations, is seamlessly described in a single protocol. Hybrid systems often represent intermediate values as secret shares over a large finite field. They may use a mix of information-theoretic and cryptographic protocols, and as such, the number of computation parties and threat models vary.

Hybrid models allow for very different performance characteristics than strict circuit-based models. For example, in a finite field, operations like comparisons, bit-shifts, and equality tests are expensive to represent as an arithmetic circuit. However, specialized sub-protocols that operate on secret shares (e.g., [DFK$^+$06, DGK08, Cou18]) can compute a sharing of the result far more efficiently.

2.3 Simulation security

The security of cryptographic protocols is often defined using the notion of *indistinguishability* of random variables (Definition 1). Roughly, two families of random variables (indexed by natural numbers) are computationally indistinguishable, if the probability that any polynomial-time algorithm can successfully distinguish a single sample of one distribution from a single sample of the other decreases super-polynomially in the instance size.

Definition 1 (Indistinguishability)**.** *Two families of random variables $\{X_k\}_{k\in\mathbb{N}}$, $\{Y_k\}_{k\in\mathbb{N}}$ are said to be* computationally indistinguishable*, denoted $X_k \overset{c}{\approx} Y_k$, if for all probabilistic polynomial-time algorithms A, and all $c > 0$, there is an K such that for all $k > K$,*

$$|\Pr[A(X_k) = 1] - \Pr[A(Y_k) = 1]| < k^c. \tag{2.1}$$

A multi-party computation protocol is called *secure* if the execution of the protocol itself reveals nothing more about the participants' (private) inputs than what is revealed by the output alone. In protocols where there is no output (or the output is secret-shared), the participants should learn *nothing* about the inputs. Roughly, this means that each player's *view* of the protocol (*i.e.*, the messages they send and receive) should be independent of the other participants' private inputs (after conditioning on the output).

This ensures that each participant (or certain colluding subsets of participants) learn nothing from executing the protocol that they could not have learned from the output alone. In this way, MPC cryptographically emulates a trusted party who accepts each participant's private input, computes the desired function and returns the result.

In the simulation paradigm, a protocol is said to be secure if there exists an efficient (polynomial-time) simulator that, when interacting with an *ideal functionality* (an incorruptible third party that always executes the protocol correctly), can produce views for each participant that are indistinguishable from the transcript created by a real execution of the protocol.

To formalize this notion of security, in a setting with n participants, define, for $i = \{1, \ldots, n\}$, a function of interest that needs to be computed in a privacy-preserving way:

$$f_i : X^n \to Y$$
$$(x_1, \ldots, x_n) \mapsto y_i.$$

The variables x_i, y_i are the input and output of participant i. Note that in many scenarios all f_i (and thus, y_i) are equal, *i.e.*, all participants receive the same output.

The *view* of player i in a multi-party protocol on inputs $x_1, \ldots, x_n$, denoted $\mathsf{view}_i(\mathbf{x})$, is the collection of all messages sent and received by player i when the protocol is run on input $\mathbf{x}$.

A protocol is (t-n)-secure if any set of at most t participants can gain no information about the remaining $n-t$ players' private inputs by colluding together. Setting $t = n-1$ provides the strongest notion of security, *i.e.*, each individual's privacy is preserved even if *all* other participants collude against her.

The proof defines two modes of execution. In the **Ideal** experiment, all parties send their inputs to an *ideal functionality* $\mathcal{F}$, a incorruptible party that always produces the correct output on a set of inputs. Corrupted parties communicate with the ideal functionality via a *simulator* Sim that translates regular inputs into the format accepted by $\mathcal{F}$. In the **Real** experiment, all parties collaborate to run the real protocol. The output of each experiment is full view of the corrupted parties and the outputs of the honest parties.

Definition 2 (Simulation-based security)**.** *An n-party computation protocol for computing the function $f \overset{def}{=} (f_1, \ldots, f_n)$ is said to be (t-n)-secure against a malicious adversary if for every probabilistic polynomial-time real-world adversary $\mathcal{A}$, there exists a probabilistic polynomial-time ideal-world adversary* Sim *such that for any corrupted parties $T \subset [n]$, with $|T| \leq t$,*

$$\{\mathbf{Ideal}_{\mathsf{Sim},\mathcal{F},T}(1^\lambda, \mathbf{x})\}_{\mathbf{x}} \overset{c}{\approx} \{\mathbf{Real}_{\mathcal{A},\Pi,T}(1^\lambda, \mathbf{x})\}_{\mathbf{x}}.$$

where the security parameter $\lambda \in \mathbb{N}$.

Further details on this technique can be found in [Lin16].

CHAPTER 3

General-Purpose Frameworks for Secure Multi-Party Computation

Despite a demonstrated demand for MPC technology, practical adoption has been limited. This can be attributed to many causes, but a common concern is the *efficiency* of the underlying protocols. General-purpose MPC protocols, capable of securely computing *any* function, have been known to the cryptographic community for 30 years [CCD88, GMW87, Yao82, Yao86]. Until recently such protocols were mainly of theoretical interest, and were considered too inefficient (from the standpoint of computation and communication complexity) to be useful in practice.

To address efficiency concerns, cryptographers have developed highly-optimized, special-purpose MPC protocols for a variety of use cases. Unfortunately, this mode of operation does not foster widespread deployment or adoption of MPC in the real world. Even if these custom-tailored MPC protocols are theoretically *efficient* enough for practical use, designing, analyzing and implementing a custom-tailored protocol from the ground up for each new application is not a scalable solution.

General-purpose MPC *frameworks*, could drastically reduce the burden of designing multiple custom protocols and could allow non-experts to quickly prototype and deploy secure computations. Frameworks can amortize the engineering effort devoted to making general-purpose MPC protocols practical and secure across all of the uses of such a system.

Many significant challenges arise when designing and building an MPC framework. In general, implementing any type of multi-round, distributed protocol robustly and efficiently is a major engineering challenge, but MPC presents additional requirements that makes it especially challenging to build correctly. For efficiency, both the front-end compiler and the cryptographic back-end need to be highly optimized. For usability, the compiler needs to be expressive, flexible, and intuitive for non-experts, and should abstract away many of the complexities of the underlying MPC protocol, including circuit-level optimizations (e.g. implementing floating-point operations as a Boolean circuit) and back-end protocol choice (e.g. selecting an optimal protocol for a particular computation). With today's frameworks, optimizing performance often still requires a fair degree of knowledge and effort on the part of the user.

Fairplay [MNPS04] was the first publicly available MPC framework. It translated code written in a high-level Secure Function Definition Language (SFDL) into a garbled circuit representation, which could then be (securely) evaluated by two parties. Fairplay was subsequently extended to allow for true multi-party computation in FairplayMP [BNP08], using a modified version of the BMR protocol [BMR90]. It was followed shortly by VIFF [GTK+, DGKN09] and SEPIA [BSMD10], which used the same basic architecture: they took programs written in fairly high-level languages, converted them to a circuit format, and executed the circuit using a secure computation protocol.

These early frameworks showed that general-purpose MPC was achievable, and, although their performance rendered them unsuitable for most real-world applications, they launched what is now a very active field of research in MPC framework design and implementation.

Thanks to these efforts, dramatic improvements in secure computation algorithms coupled with a steady increase in hardware performance have made MPC a viable solution to a large class of real-world problems. Modern MPC protocol implementations are fast enough to securely evaluate complex functions on moderately-large data sets, such as the numerous implementations of secure regression analyses with tens to hundreds of thousands of observations, and tens to hundreds of variables [BKLS14, CDNN15, GSB+17, KLSR09, NWI+13].

The rush of activity in this field can be difficult to navigate: dozens of new protocol implementations and supporting frameworks encompass a wide variety of architectures and features which influence their efficiency, usability and suitability for different tasks. The goal of this chapter is to provide a guide to the powerful new breed of MPC frameworks, and is primarily aimed at four distinct types of readers.

1. Developers looking to choose a framework with which to implement a specific secure computation

2. Theoretical cryptographers looking to understand state-of-the-art in practical, secure computation
3. Framework designers looking to understand the limitations of existing technology and identify new research directions
4. Managers and policy-makers looking to understand whether existing technology is mature enough for their needs

This chapter surveys the state-of-the-art MPC frameworks, evaluating them based on their general architecture, underlying technology, threat models, and expressiveness. Our comparison focuses on usability features rather than performance metrics, and we report on our experience with implementing three small test programs in each case. Casual readers may wish to skim Section 3.3, which discusses each framework in greater depth, and focus on the final discussion section, where we advocate for improved documentation and standardization and suggest future directions in compiler research.

Many of these frameworks are themselves research projects or works-in-progress: they have nontrivial build dependencies and complicated work flows. Indeed, implementing our simple example programs in each system required significant engineering effort: we estimate over 750 person-hours. To allow others to experiment more easily with these systems, we have created an on-line Github repository[1] with two artifacts: (1) a set of Docker containers, each of which provides a development environment configured with the required software infrastructure for each MPC framework, along with executable examples of our test cases, and (2) a wiki page that collects much of the evaluation presented here with additional documentation about each framework. These online resources have been updated with additional software frameworks that are not described in this chapter, including JIFF [LDAI+19, DAF], MP-SPDZ [Kel20], and MPyC [Sch]. It includes partial infrastructure for ABY3 [MR18] and FRESCO [Ale], and notes other frameworks that have yet to be added (EzPC [CGR+17], MOTION/HyCC [BDST20, BDK+18], and swanky [CMR]).

Related surveys. There is a small literature dedicated to surveying the state of MPC. Archer et al.'s survey [ABPP15] of secure computation tools across several paradigms, including garbled circuit schemes, defines a maturity taxonomy that aims to describe the practical readiness of several

[1] `https://github.com/MPC-SoK/frameworks`

schemes. Shan et al.'s survey [SRBW18] outlines different threat models and computation techniques for securely outsourcing many specific types of computation. The authors of Frigate [MGC$^+$16] include a short survey of existing MPC frameworks, which focuses on correctness and covers a slightly older body of work. The SSC protocol comparison tool [PGFW, PGFW14] allows users to find published protocols matching certain security or privacy criteria, but this tool classifies *theoretical* protocols rather than implementations and does not include protocols developed in the past few years. The awesome-mpc repository [Rot18] provides an up-to-date list of frameworks, back-ends and special-purpose protocols, with a short description of each.

To the best of our knowledge, these previous works did not actually install and experiment with each of the systems they surveyed, but drew their conclusions based on the descriptions of the systems in their published papers and documentation. Unfortunately, we have found that the features, functionality and syntax of the actual implementations do not always match those found in the documentation.

Alternatives to MPC. Fully homomorphic encryption [Gen09] provides an alternative method for securely computing functions encoded as arithmetic circuits. There are several libraries implementing fully homomorphic encryption including HELib [HS15, HS14], PALISADE [RR16] and SEAL [SEA20]. We do not include these in our comparisons; they are surveyed in [VJH20]. There are efforts to garble RAM-model programs [LO13, GHL$^+$14, GLO15], instead of circuits, but these have not been implemented. There is a great deal of software for special-purpose secure computation, such as private set intersection, and tailored to specific use cases, such as machine learning. We omit these from this survey.

3.1 Frameworks survey

We survey eleven general-purpose MPC compiler frameworks, all of which follow the same general approach: first, a *compiler* converts a program written in a specialized, high-level language to an intermediate representation (often a circuit). Then the circuit is passed as input to a *runtime*, which executes an MPC protocol and produces an output. We survey two compilers (Frigate and CBMC-GC) that do not have a runtime component.

Table 3.1 gives basic information about these frameworks, including protocol type, security

	Paper	Protocol family	Parties supported	Mixed-mode	Semi-honest	Malicious	Language docs	Online support	Example code	Example docs	Open source	Last major update
EMP-toolkit	[WMK16]	GC	2+	◐	●	●	○	○	●	○	●	*05/2019*
Obliv-C	[ZE15]	GC	2	●	●	○	●	○	●	◐	●	02/2019
ObliVM	[LWN$^+$15]	GC	2	●	●	○	○	○	●	○	●	02/2016
TinyGarble	[SHS$^+$15]	GC	2	○	●	○	○	○	◐	○	◐	10/2018
Wysteria	[RHH14]	MC	2+	●	●	○	◐	○	●	◐	●	10/2014
ABY	[DSZ15]	GC,MC	2	●	●	○	●	○	●	●	●	*05/2019*
SCALE-MAMBA	-	Hy	2+	◐	●	●	●	●	●	◐	●	*11/2020*
Sharemind	[BLW08]	Hy	3	●	●	○	●	●	●	●	◐	*03/2019*
PICCO	[ZSB13]	Hy	3+	●	●	○	●	○	◐	○	●	10/2017
Frigate	[MGC$^+$16]	-	2+	○	-	-	●	○	●	○	●	05/2016
CBMC-GC	[HFKV12]	-	2+	○	-	-	◐	○	●	○	●	04/2018

Table 3.1: A summary of defining features and documentation types. Partial support (◐) is explained in Section 3.2.1.

settings, availability, and some usability features such as documentation. We limit our scope to recent work: each framework has had a major update since 2014. We do not consider frameworks that are primarily protocol implementations: all the projects included have a developed front-end. We also do not consider standard libraries and APIs related to secure computation. In many cases, we only consider the latest work by a research group.

These frameworks are typically introduced in an academic *Paper*. Although many develop a custom or optimized protocol, we group them broadly into *Protocol Families* as described in Section 2.2: Garbled Circuits (GC), Multi-party Circuit protocols (MC), and Hybrid models (Hy). We also note the number of computation *Parties Supported* in a protocol evaluation.

We identify two main threat models as described in Section 2.1.3: *Semi-honest* and *Malicious* adversaries. In practice, security against a malicious adversary is implemented as a "malicious-with-abort" scheme, where the protocol aborts if malicious activity is detected; a malicious adversary cannot cause an honest party to receive an incorrect result, but may prevent them from receiving an output altogether. These descriptors do not apply to frameworks that do not execute a secure computation.

We note whether the framework allows *Mixed-Mode* computation: a way to execute both secure and insecure operations in a single program.

In all tables, partial support is indicated by a ◐ symbol; these limitations are explained in detail later in the text.

3.1.1 Engineering challenges

Many protocols use a circuit-based model to represent the function being computed. This has the advantage that circuit-based computations are input independent and thus the run-time of the protocol leaks nothing about the user inputs. However, using a circuit model introduces serious challenges and limitations that are present to some degree in all of the frameworks we tested.

Arithmetic circuits operate over a finite field whose size must be set in advance, and must be large enough to prevent overflow (which will vary by application). Arithmetic circuits do not natively support non-arithmetic operations like comparisons and equality checks.

Boolean circuits need to redefine basic operations for every bit width: supporting arithmetic on n-bit integers in such a protocol requires implementing n-bit addition and multiplication circuits. We found no standardization in this area, and most Boolean circuit compilers design and implement their own arithmetic operations. This leads to many restrictions on acceptable programs, and most Boolean-circuit-based frameworks do not support arbitrary bit-width operations.

Compiling high-level programs into circuits requires unrolling all loops and recursive calls. For privacy, the number of loop iterations and recursion depth cannot depend on private inputs. In some situations, static analysis techniques can infer loop termination conditions, but most compilers do not support such analysis, and instead force the programmer to manually define loop bounds at compile time. Few compilers support recursion.

Conditional operations on secret data can reveal which branch was chosen if the branches take different amounts of computation, so they are typically implemented as a multiplexer, where both branches are evaluated. Similarly, a simple array lookup on a private index must be expanded into a linear-size multiplexer circuit. Frigate, CBMC-GC, and PICCO implement private indexing in this way. Oblivious RAM [GO96] provides a method for making RAM access patterns data independent, but few frameworks have ORAM support, either natively (ObliVM and SCALE-MAMBA) or via a library (Obliv-C). Most languages do not even allow private array indexing syntactically: if `i` is a "secret" integer and `v` is a "secret" array, then `v[i]` is not valid syntax.

Balancing transparency and flexibility is a key challenge for the MPC compiler designer. MPC protocols often have very different performance characteristics than the corresponding insecure computation, and a compiler that completely hides these differences from the end-user (e.g. automatic multiplexing) in order to provide a more versatile, expressive high-level language can lead developers to write code that is not "MPC-friendly." Alternately, a framework that provides direct access to different back-end representations assumes a high degree of cryptographic expertise on the part of end-users but allows expert users to write highly optimized MPC protocols. It is possible to provide both expressiveness and protocol efficiency without requiring a high degree of cryptographic knowledge on the part of the developer by automatically selecting the optimal MPC protocol for different parts of the code. This type of automatic optimization is difficult, however, and the MPC compilers we analyzed do not attempt it. The EzPC project [CGR$^+$17], HyCC framework [BDK$^+$18] and ABY^3 project [MR18], all designed similarly to ABY (Section 3.3.6), attempt to automatically optimize the back-end representation among three protocols. At the time of this writing, none of these projects had code available and we have not included them in our tests.

3.2 Evaluation criteria

3.2.1 Usability

We consider the tools and documentation a developer needs to install, run, and write programs using the framework. Our findings are summarized in Table 3.1. We identify several types of valuable documentation. Thankfully, every framework we tested includes some form of basic installation documentation. *Language Doc*umentation gives an overview of the high-level language: a language architecture and design document, a start-up guide or tutorial, or a generated list of types and built-in functions. Some larger systems also have *Online Support*, such as an active mailing list or paid personnel who provide technical support[2]. Functional *Example Code* demonstrates end-to-end execution of a program within a framework and is often more up-to-date than general language

[2]Although we received support from academic authors, we do not count responsive authors as "online support;" this model is not scalable.

documentation (a ◐ indicates we needed additional files or tools to run examples). Explicit *Example Doc*umentation provides context for these programs, either in the comments of the code or a separate document (a ◐ indicates limited documentation; details in Section 3.3).

Most frameworks are *Open Source* under a standard GNU or BSD license (a ◐ indicates closed-source tools or code are required for full functionality). We record the date of the *Last Major Update* (as of this writing): either ongoing development or the latest tagged release. Dates in *italics* indicate the framework remains actively under development.

3.2.2 Sample programs

We implemented three sample programs to evaluate usability, expressiveness, architecture, and cryptographic design of the frameworks.

Multiply Three. This program takes integer input from three parties and computes the product. The simple function demonstrates the structure of each framework. The program tests whether the implementation supports three or more parties, or if there is a simple way to secret-share multiple inputs across the two computation servers. It also tests basic numeric capabilities of the framework: input and output of integers and simple computation on secure types.

Inner Product. The inner product takes the sum of the pairwise product of the elements of two vectors. It tests array-related functionality. Parties should be able to pass an array as input, store secret data within, and access and iterate over the contents. Some frameworks provide ways to parallelize operations over arrays, either through explicit syntactic support or through a parallel architecture device like SIMD gates.

Crosstabulation. The crosstabulation program calculates averages by category, where the category table and value table share a primary key but are owned by different parties. This tests framework expressiveness, including input, output, and modification of arrays and conditionals on secret data. In some cases, we tested whether user-defined data types (structs) are supported. We used a simple, brute-force algorithm, and typically returned a list of sums by category (rather than averages).

3.2.3 Functionality

We assess the expressive ability of the high-level language used to define secure functions. These criteria are summarized in Table 3.2.

	Data Types							Operators							Grammar		
	Boolean	Fixed int	Arbitrary int	Float	Array	Dynamic array	Struct	Logical	Comparisons	Addition	Multiplication	Division	Bit-shifts	Bitwise	Conditional	Array access	Pvt index
EMP-toolkit	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○
Obliv-C	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	Lib
ObliVM	○	●	●	●	◐	●	◐	●	●	●	●	●	●	●	●	●	ORAM
TinyGarble	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Wysteria	●	●	○	○	◐	-	●	○	●	●	●	◐	○	○	●	○	○
ABY	◐	●	○	◐	●	○	●	●	●	●	●	○	○	○	●	◐	○
SCALE-MAMBA	○	●	◐	●	●	○	◐	○	●	●	●	●	●	●	●	●	ORAM
Sharemind	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	○
PICCO	◐	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	Mux
Frigate	○	●	●	○	●	○	●	○	●	●	●	●	●	●	●	●	Mux
CBMC-GC	●	●	○	◐	●	○	●	●	●	●	●	●	●	●	●	●	Mux

Table 3.2: A summary of functionality and expressibility of each high-level language. Partial support (◐) is explained for individual frameworks in Section 3.3.

Data types. A fully-supported data type must have both public and secret forms, and the language should allow input and output of the type. These include *Boolean*s, signed or unsigned *Fixed*-length *Int*egers, and more complicated numerical types, such as *Arbitrary*-length *Int*egers and *Float*ing- or fixed-point numbers. Although libraries for these types can be built using fixed-length integer types, we only mark them supported if they are available by default. Combination types include *Array*s and *Dynamic Array*s, where the latter has a size not known at compile time, as well as *Struct*s, user-defined types that can hold other data types as sub-fields. These complex types are marked supported if they can contain secure data.

Operators. Supported operators can be applied to secret data types to get a secret result. We consider *Logical* operators on Booleans and *Comparisons* (equality and inequalities) between integers. We group *Addition* and subtraction as one category, and *Multiplication* and *Division* separately. We consider two bit-level operations: *Bit-shifts* on fixed-length integers and *Bitwise* operators.

				Model			I/O			
	Development language	AES -NI	Architecture	Arithmetic	Boolean	On-the-fly	Arbitrary format	Different input	Array output	Multiple output
EMP-toolkit	C++	●	Lib (C++)	○	●	●	●	●	◐	●
Obliv-C	OCaml, C	●	Ext (C)	○	●	●	●	●	●	●
ObliVM	Java	○	Ind	○	●	●	○	●	○	○
TinyGarble	C/C++	●	Ext (Verilog)	○	●	●	-	-	-	-
Wysteria	OCaml	-	Ind	○	●	-	○	○	-	◐
ABY	C++	●	Lib (C++)	●	●	○	●	●	●	●
SCALE-MAMBA	Python,C++	-	Ind	●	○	-	●	●	◐	●
Sharemind	C/C++	-	Ind	●	○	-	●	●	●	●
PICCO	C/C++	-	Ext (C)	●	○	-	○	●	●	●
Frigate	C++	-	Ind	○	●	○	-	●	●	◐
CBMC-GC	C++	-	Ext (C)	○	●	○	○	●	●	◐

Table 3.3: Details on architectural and implementation details. Partial support (◐) is explained in Section 3.2.4.

Grammar. *Conditional*s on a secret Boolean condition can be implemented either with if-statement syntax or a multiplexer operator, though we require an explicit language construct. We consider *Array Access* with a public index and the harder problem of array access with a *Private Index.* The latter can be implemented as a linear-time multiplexer (Mux), native ORAM support (ORAM) or a library for ORAM (Lib).

3.2.4 Implementation criteria

In this section we define architectural and cryptographic criteria, summarized in Table 3.3. We note the main *Development Language* for each framework. Garbled circuit protocols can significantly improve performance by using *AES-NI*, an extension to the x86 instruction set that speeds up AES encryption and decryption.

Architecture

We define three broad architecture categories. *Independent* (Ind) frameworks develop novel languages and compilers: from limited, domain-specific languages that interface with existing general-purpose languages to stand-alone environments. Some frameworks are *Extensions* (Ext) of an existing language. These may modify or extend existing compilers to add functionality or take a compiler intermediate state as input. *Library* (Lib) frameworks are fully implemented in an existing language. They generally define a secure type class and methods for circuit construction and protocol execution. Library and Extension types both depend on a host language, which we specify in the table.

Computation model

We consider whether the underlying computation model is over an *Arithmetic* field or is based on *Boolean* circuits. Garbled circuit implementations can generate circuits *On-the-Fly*, starting runtime execution before the circuit is fully generated. This can reduce resource consumption, allow dynamic array and loop bounds, and reduce overall program runtime.

I/O

Input is typically read from a file, but some frameworks allow *Arbitrary Format*ting, rather than a specific input format. (We've produced input generation scripts for our sample programs.) We note whether the framework supports *Different Input* types from each party. Frameworks should support *Array Output* (a ◐ indicates array elements must be returned one at a time) and *Multiple Output*, where a single party receives two or more output values in a single computation (a ◐ indicates multiple values must be wrapped in a struct).

In Section 3.3, we comment specifically on restrictions in the I/O file formats, including support for arbitrary-size integers. We recognize that many frameworks are produced in an academic setting that may not value "engineering problems" such as I/O, but we found that the significant usability impact of these capabilities make them worth discussing in this survey.

3.2.5 Performance

In this work, we focus on *usability* and do not benchmark framework performance (e.g. run-time, bandwidth, memory-usage, circuit depth). We believe a quantitative evaluation of our sample programs would not accurately represent the performance abilities of each framework. There are several reasons why theoretical efficiency metrics are not always applicable in practical MPC architectures, and we found that direct comparison between different models was often misleading. Circuit size and depth have different implications in garbled circuit and secret-sharing-based protocols, and many frameworks never generate a full circuit for comparison. Execution time varies based on the framework architecture, and preprocessing phases and other variations in execution architecture further complicate timing measurements.

Variations in protocol family and threat model mean that most frameworks are not directly comparable, and our choice of benchmarking function will have a major effect on the relative speed of the frameworks. Our sample programs are designed to reveal the expressive capabilities of a framework and do not necessarily represent a practical MPC use case. A stand-alone measurement for a single run of one of our programs would not take into account the context (typically part of a larger system) in which a secure computation may be evaluated in practice.

We do not wish to disservice incomparable frameworks by providing concrete numbers for impractical test cases. Although a worthwhile and practically useful endeavor, producing a realistic testing framework is beyond the scope of this project.

3.3 Frameworks

In this section, we discuss each framework in detail, elaborating on limitations noted in the tables and on the overall usability of each framework. We make recommendations on appropriate use for each framework. We emphasize that many of these frameworks are academic projects, and are therefore subject to the engineering constraints of such an endeavor. Even as we describe the limitations of these compilers, we wish to emphasize the significant contributions that each has made to the field.

3.3.1 EMP toolkit

EMP toolkit [WMK16] is an extensive set of MPC frameworks based on garbled circuits. The core toolkit includes an oblivious transfer library, secure type classes, and several custom protocol implementations. We tested two protocols: a semi-honest implementation of Yao's garbled circuit protocol, and a maliciously secure protocol with authenticated garbling [WRK17a]. The toolkit includes three maliciously secure protocols we did not study: a two-party computation that checks input validity [KMW16], a two-party computation library [WMK17], and a multi-party protocol [WRK17b].

Semi-honest. The implementation of Yao's garbled circuit protocol is a C library defining secure type classes and operations. We found it intuitive for non-expert C developers. The library structure allows developers to use C arrays and structs to hold secure values, and provides simple mixed-mode computation and can generate circuits on-the-fly.

The framework supports arbitrary-size integers and floating-point numbers. Arbitrarily large values can be initialized from a string and returned similarly; they are not restricted to C types. There is little explicit language documentation, but the code was relatively clear. The library can output a protocol-agnostic circuit, but this is not documented.

Malicious Authenticated Garbling. This library is primarily an implementation of a custom garbling protocol. We were able to run the included pre-compiled circuit examples and several of our own examples. However, supporting features are limited: functions must be encoded as a circuit prior to computation using the semi-honest library (thus, this version does not support mixed-mode computation), and I/O is encoded in Boolean arrays.

Recommendation: We recommend the EMP-toolkit semi-honest library for general use. The entire platform is well suited to academics looking to implement a novel circuit-based protocol due to the available circuit generation and cryptographic libraries, but we note that the end-to-end flow is not seamless.

3.3.2 Obliv-C

Obliv-C is an extension of C that executes a two-party garbled circuit protocol. The main language addition is an *obliv* qualifier, applied to C types and constructs. Typing rules enforce that *obliv* types remain secret unless explicitly revealed. Code within oblivious functions and conditionals cannot modify public data, except within a qualified ~*obliv* block, in which the code is always executed. These rules allow programmers to reason about data security and develop modular libraries.

The compiler combines these extended functionalities with supporting C code to produce an executable. The executable generates circuits on-the-fly. This allows circuit sizes to depend on values not known during compilation, but may result in under-optimized circuits.

We successfully used an Obliv-C ORAM library, Absentminded Crypto Kit[3], which implements several ORAM variations and other useful primitives [ZWR+16, Ds17].

[3]`https://bitbucket.org/jackdoerner/absentminded-crypto-kit`

Obliv-C extends C but many of the examples imply an independent-language architecture, separating Obliv-C code from C code. Example programs typically read, process, and output data in native C code, performing only the secure computation in Obliv-C code. However, this abstraction is not enforced: it is possible to perform I/O and call native C functions in Obliv-C files. While many of the examples implement a strict separation between supporting C code and secure Obliv-C code, example documentation uses a mixed paradigm.

Several groups have used Obliv-C to implement secure functionalities, including linear regression [GSB$^+$17], decentralized certificate authorities [JLE17], aggregated private machine-learning models [TJGE16], classification of encrypted emails [GFAW17] and stable matching [DEs16].

Recommendation: Obliv-C is a robust garbled circuit framework. We recommend it to developers for general use and to academics who wish to implement and optimize useful libraries such as ORAM.

3.3.3 ObliVM

ObliVM compiles a Java-like language called ObliVM-lang and executes a two-party garbled circuit protocol. It aims to provide a language intuitive to non-experts while implementing domain-specific programming abstractions for improved performance.

ObliVM-lang allows custom data types and type inference. It implements a built-in efficient ORAM scheme. ObliVM natively supports fixed-size integers, and includes a library for arbitrary sized integers.

However, documentation is limited, both for the language (we identified several undocumented reserved keywords), and for general usage. I/O is limited: it requires a non-human-readable format, and we did not find a method to return complex types (including structs and arrays) or more than 32 bits of information. We did not successfully implement the crosstabulation example.

Recommendation: Although ObliVM implements advanced cryptographic constructs, its usability for practical applications is significantly limited by its minimal documentation and restricted I/O functionality.

3.3.4 TinyGarble

TinyGarble [SHS$^+$15] repurposes hardware circuit generation tools to create optimized circuits appropriate for a garbled circuit protocol. It takes a three-step approach: first, it converts a function defined in Verilog to a *netlist* format. Then it converts the netlist format to a custom circuit description (SCD) and securely evaluates the Boolean circuit using a garbled circuit protocol.

We found that the first step of this process requires a closed-source logic synthesis tool (the Synopsys Design Compiler) that converts a Verilog file to the unstandardized netlist format. The authors reference an open-source tool, Yosys Open SYnthesis Suite, but we were unable to compile any examples (conversions from Yosys-produced netlist files to SCD failed). The source code for TinyGarble includes some pre-compiled netlist files. While we could see every step for these examples (Verilog source, netlist file, computation output), we were unable to compile examples end-to-end and thus do not make any claims about language functionality.

TinyGarble is preceded by JustGarble [BHKR13], a library for circuit garbling and evaluation. JustGarble does not include communication or circuit generation. The garbled circuit implementation in TinyGarble is a strict improvement over JustGarble, including recent protocol and circuit optimizations. A follow-up to TinyGarble [DDK+15] leverages hardware logic synthesis tools to optimze for GMW-style computations.

Recommendation: TinyGarble aims to leverage powerful circuit optimizers developed for producing hardware circuits. Unfortunately, from a usability standpoint, the lack of compatibility for Verilog compilers and lack of standards around netlist formats meant that we were unable to compile or run any new examples using the TinyGarble framework. We believe, however, that the MPC community could benefit greatly by leveraging the power of existing circuit optimizers.

3.3.5 Wysteria

Wysteria [RHH14] introduces a novel high-level *functional* programming language. It guarantees that a distributed secure computation produces the same output as a single trusted party. Wysteria supports an arbitrary number of computation parties, and the software contribution includes a front-end language, a type checker, and a run-time interpreter that executes a Boolean-circuit-based GMW protocol implementation [CHK+12].

Wysteria supports mixed-mode computation via a language construct called a *secure block*. A secure block is initialized with a set of parties and their inputs. All operations in the block's scope are compiled to a Boolean circuit and executed as a separate computation.

The Wysteria codebase has changed since the original publication of the paper, and the examples presented in the paper do not compile. However, they provide useful context for the architecture of a working program. The repository includes several example programs that run without errors, including a 6-party version of the millionaire's problem.

Wysteria includes a record type, which holds named values of other types and can be returned from a secure block. Although Wysteria includes working examples that input an array to a secure block, we were unable to replicate this for our inner product or crosstabulation programs. The language has support for iterating over arrays in secure blocks, and allows access to individual array elements outside of a secure block. We encountered other significant language limitations: Wysteria only supports division by 2 in secure blocks, and we did not find a way to use logical operators on Booleans.

The Wys* project [RSH17] built on the ideas of Wysteria, and attempted to create a fully-verified toolchain for secure computation based on the F* programming language. The F* language is undergoing rapid development, however, and we were not able to compile Wys*.

Recommendation: Wysteria's limited support for complex data types, current lack of development, and outdated back-end circuit parser, mean that it should not be used for developing complex or efficient protocols. On the other hand, Wysteria is the only compiler we examined that intends to provide a system for automatically verifying that the underlying multi-party computation has the same functionality as the monolithic program implemented by the developer, and the only compiler with a functional-style programming language. We recommend future compiler developers use Wysteria's type-based correctness and security guarantees as a model.

3.3.6 ABY

ABY [DSZ15] is a mixed-protocol two-party computation framework implemented as a C++ library. It aims to give developers fine-grained control over computation efficiency by providing a mechanism for mixing protocols. ABY switches between three protocols: The GMW-based *Arithmetic* protocol uses an additive sharing scheme with multiplicative triples on an arithmetic circuit. The protocol is based on those by [ABL$^+$04, KSS13, PBS12]. The other protocols use Boolean circuits: the *Boolean* protocol implements the original GMW protocol with an XOR-based sharing scheme, while the *Yao* protocol uses an optimized version of Yao's garbled circuit protocol.

ABY has a significant amount of documentation that provides a helpful framework for understanding the capabilities of the framework. This includes a slightly outdated developer guide, an extended README file, and a variety of commented examples.

Secure data is limited to unsigned C integer types: ABY does not support arbitrary-length integers or an explicit Boolean type, although it allows one-bit integers that function equivalently. It supports some floating-point operations and is actively developing this functionality. ABY can store secure data in a C struct, and supports both C++ arrays and SIMD constructions for efficient parallel operations. ABY provides functions for creating and populating SIMD "shares", but retrieving individual elements requires operating on the internal representation of secret data, which is not well-supported.

ABY has been used to implement several secure computation systems [ARS$^+$15, CDC$^+$16, CDC$^+$17, DHSS17, KLS$^+$17, PSWW18].

Recommendation: ABY provides a powerful, low-level cryptographic interface that gives the developer significant control over performance. ABY is targeted at users who are familiar with MPC protocols and the circuit model of computation. We recommend it to developers with sufficient cryptographic background.

3.3.7 SCALE-MAMBA

SCALE-MAMBA implements a maliciously secure two-phase hybrid protocol and supersedes the SPDZ framework. MAMBA is a Python-like language that compiles to a bytecode representation. SCALE implements a two-phase protocol which offloads all public-key operations to an offline pre-processing phase, generates three types of shared randomness to use during protocol execution, and executes an optimized hybrid protocol based on previous work [BDOZ11, DPSZ12, NNOB12].

SCALE-MAMBA has a significant amount of documentation, covering the differences from the previous SPDZ system, installation and runtime instructions, updated language documentation, and protocol primitives. The example programs are unit-style tests but are not explicitly documented. A community bulletin board [SPD18] hosts discussion and questions about the framework. The custom language runs a virtual machine that supports operations on both secure and public values, but is not particularly efficient on public values; we recommend using a separate program for any significant operations in the clear.

The SCALE-MAMBA framework allows developers to define their own I/O classes. This provides an extremely flexible interface. We did not implement a custom I/O class. The framework's secure channel setup requires users to produce a mini certificate authority in order to run a computation.

Running our sample programs with a simple full-threshold secret sharing scheme required significant memory resources. However, the system offers multiple, customizable options for a secret-sharing scheme, and includes programmatic tools for offline data generation in certain contexts. For testing purposes, SCALE includes an option to run with fake (insecure) offline data.

Integer size is determined by the field size, which must be chosen at compile time. Standard full-threshold sharing supports a modulus of up to 1024 bits. SCALE-MAMBA supports most bit-shift operations and includes Python-style tuples, which we consider to be a less powerful type of struct. Fixed-point numbers are fully supported, and floating-point numbers partially implemented. It has ORAM support, which we did not test.

Recommendation: We recommend SCALE-MAMBA for a variety of uses: it is flexible, supports an arbitrary number of parties and has strong security guarantees, though it may require significant computing resources.

3.3.8 Sharemind MPC

Sharemind [BLW08] is a secure data processing platform and a trademark of Cybernetica, a research-and-development-focused technology company based in Tallinn, Estonia. We used the Sharemind MPC platform, which securely executes a function written in the SecreC language. The framework executes a three-party hybrid protocol using an additive secret sharing scheme.

The Sharemind MPC platform explicitly defines three parties: *clients*, who input values; *servers*, who define and run the secure computation; and *outputs*, who receive the output of the computation. Server code is written in the SecreC language and executed using Sharemind's secure runtime, while client and output code uses a client library in a common programming language and is executed using standard compilers. We developed our sample programs using a C client library; the platform also provides libraries in Haskell, Java, and JavaScript.

Sharemind MPC implements a custom additive secret-sharing scheme over a fixed-size ring. These fixed-size integers have behavior consistent with traditional C integers, and the framework includes a floating point library. The protocol is written for exactly three servers, but there is support for arbitrarily many parties secret-sharing their inputs among the three computation servers into a database structure. Our samples passed all input values from a single client program. The SecreC language is expressive and well documented online[4]. The supporting client libraries are not as well documented.

The Sharemind MPC platform has several licensing options[5] through Cybernetica. We used the academic server. This license gave us access to the protocol implementation. The platform also includes an open-source simulator, which includes the SecreC language, its standard library, and an emulator for the secure computation. The emulator is available as a VM and as compilable source code; we successfully ran our examples on both versions. The emulator does not support client code and arguments must be passed on the command line.

As part of the academic license, we had access to several Sharemind employees who provided "reasonable assistance" throughout the development process, and we consider this to be an *online resource*.

Recommendation: The Sharemind MPC platform is suitable for a wide variety of purposes. We recommend it to companies looking to implement secure computation, particularly for large or complex functionalities, as well as to academics who require MPC as a tool for a project.

3.3.9 PICCO

PICCO [ZSB13, ZBA17] is a general-purpose compiler with a custom secret-sharing protocol. It includes three main software contributions: a compiler that translates an extension of C to a native C implementation of the secure computation; an I/O utility that produces and reconstructs secret-shared input and output; and a tool that initiates the computation. PICCO executes computations under a hybrid model, using an information-theoretic protocol for multiplication [GRR98] and custom primitives for other operations. It supports an arbitrary number of parties but requires an honest majority.

PICCO allows conditionals on private variables, but does not allow public variables to be assigned within the scope of such a conditional. It also allows indexing arrays at a private location, though this is implemented as a multiplexer, not an ORAM scheme. It supports pointers to private data and dynamic memory allocation using standard C-like syntax. The language supports single-bit integer

[4]`https://sharemind-sdk.github.io/stdlib/reference/index.html`
[5]`https://sharemind.cyber.ee/sharemind-mpc/`

types to approximate Booleans and while we ran provided examples, we were unable to successfully write our own program that uses them.

The language is well-documented in the paper. The code repository includes many examples in a C extension, but doesn't include examples of the additional files needed to compile and run a program end-to-end. The process to compile and execute a secure computation is lengthy but well-documented and requires multiple configuration files and explicit generation and reconstruction of secret-shared inputs and outputs.

Recommendation: PICCO is appropriate for developers or academics who require a true multi-party implementation. We found no correctness issues and it allows a great deal of flexibility in configuring the computation.

3.3.10 Frigate

Frigate [MGC$^+$16] compiles a novel C-like language to a custom Boolean circuit representation for any number of inputs. The framework emphasizes the use of good software engineering techniques, including an extensive testing suite and a focus on modularity and extensibility. The circuit format minimizes file size, and the framework includes an interpreter to efficiently interface between generated circuits and other applications.

Frigate produces a circuit, so all operations are secure by default. The type system is extremely simple, with only three native types: signed and unsigned integers and structs. While there is no explicit Boolean type, integers are of arbitrary size and the language defines comparison and bitwise operators, so it supports equivalent functionality. Global variables are not allowed. Frigate allows arrays but they must be contained within structs. The circuit compiler provides useful errors, and the source code includes a brief description of interpreter options and a language description.

One usability issue is that basic arithmetic operations are defined only for operands of the same type and size. This may increase circuit size for some applications. The framework does not include a simulator, so any correctness checking requires a separate back end. To test the circuits generated by Frigate, we wrote a tool that converts Frigate circuits into a format suitable for execution in an implementation of the BMR protocol [BLO16].

Recommendation: Frigate provides an expressive C-like language for fast circuit generation and is a good way to estimate the circuit size of a given computation. However, even with our conversion tools that connect Frigate's circuit form to useful back-ends, executing an end-to-end MPC computation requires relatively burdensome action from the user.

3.3.11 CBMC-GC

CBMC-GC [HFKV12, FHK$^+$14] produces Boolean circuits from a subset of ANSI-C. It is based on CMBC [CKL04], a bounded model checker that translates any C program into a Boolean constraint then adapts the output of this tool to produce an optimized circuit for an MPC computation. The compiler can optimize for minimal size or minimal depth circuits. It produces circuits for any number of input parties; we compiled and simulated sample programs with up to ten parties.

We did not find adequate documentation for the limitations of the adapted subset of ANSI-C that CMBC-GC compiles. For example, variable names for inputs to the main file must be prefixed by `INPUT_`. Arrays cannot be passed natively as arguments; they must be wrapped in a struct. Non-default integer types, such as specific-width integers, can be used but need to be explicitly included. We were unable to compile a program using C Boolean types. The framework includes a rudimentary set of floating-point operations, and allows conditionals on secret data. There are some configuration options, such as circuit optimization technique, depth to unroll loops, and a time limit on minimization.

CBMC-GC includes a tool for running circuits with ABY (Section 3.3.6). We were unable to run an example with this converter; it appears that the CBMC-GC code references a deprecated ABY API. CBMC-GC also includes a tool to output circuits in other formats, including the Simple Circuit Description (SCD) used by the TinyGarble compiler; Fairplay's Secure Hardware Definition Language (SHDL); and the Bristol circuit format [ST]. We tested the output of this tool with TinyGarble's compiler (Section 3.3.4), but were unable to run any examples; we weren't able to determine whether the errors were due to circuit generation by CBMC-GC or execution by TinyGarble.

Recommendation: CBMC-GC uses powerful tools to produce optimized circuits, but we were not able to successfully execute any of the circuits it produced.

3.4 Discussion

3.4.1 Leveraging Existing Compiler Research

Programming language research is a field dedicated to creating compilers but little MPC research leverages these techniques. Wysteria is a notable exception, but it has significant engineering gaps that make it unusable for practical computations. However, the MPC community would benefit if frameworks took a more principled approach to language design and verification.

One notable area for improvement is compiler correctness. We found that while the frameworks were generally successful in preventing security mistakes, many had correctness issues. Defining and implementing type rules that guarantee a correct output could reduce these issues, which were often silent failures.

3.4.2 Documentation

Universally, the biggest obstacle when using MPC frameworks was a lack of documentation. The community has put thousands of hours into producing the work presented herein, and even mediocre documentation makes these contributions significantly more accessible.

Documentation comes in many forms and having multiple types of documentation is helpful when using a complex software system such as these. Our evaluation criteria suggest several types we found particularly useful, and we encourage developers and researchers to produce multiple resources for system users.

In addition to static documentation provided by the authors, active online resources can be extremely valuable. These include archived correspondence, like an archived mailing list, Google group or issue tracker on GitHub. These resources can reduce the burden on researchers who may be asked the same (or similar) questions repeatedly via private correspondence. Example programs are also an important resource, and a repository where the community can archive simple example programs (e.g. like `http://www.texample.net`) would dramatically improve usability and utility of these systems.

3.4.3 Standardization and benchmarks

Many of these frameworks are designed around a particular feature, such as a type system or an optimization technique. Standardizing essential features common across frameworks allows researchers to concentrate their efforts on core features of their systems and provides a level of consistency for users. Standardization could also set a more consistent baseline for performance measurements. One potential issue is standardizing on a soon-to-be-obsolete technology. For example, while we could recommend a circuit format, this would not be useful for modern hybrid framework models that use a different intermediate representation.

Several projects are developing standardization in the field. SCAPI [Bar, EFLL12] defines a general API that provides a common interface for cryptographic building blocks and primitives commonly used in secure computation. It aims to provide a uniform, flexible, and efficient standard library for cryptographers to use in their MPC implementations and includes significant documentation. FRESCO [Ale] defines a set of Java APIs for function description and protocol definition and evaluation. As a demonstration, the project includes front-end code for several sample projects and a new implementation of the SPDZ protocol with MASCOT preprocessing. The SCALE-MAMBA systems use a set of bytecodes as an intermediate representation that have been reused in other projects, such as the Jana compiler [ABL^{+}18].

Benchmarking performance across frameworks presents a challenge due to the variety of dependencies on processing power, network bandwidth, network latency, computation structure, and framework architecture. Theoretical performance measures can be difficult to measure in practice and frameworks that excel in one benchmark environment may fare poorly in another. Nevertheless, benchmarks can provide insight into a framework's strengths and weaknesses, and do have value if they are not used as a sole measure of a framework's contribution. Recent work by Barak *et al.* [BHKL18] provides an approach for performance comparison between frameworks with compatible architectures. Keller [Kel20] builds off the sample programs in this work to benchmark the inner product operation across many frameworks.

We recommend that the community collectively develops a consistent set of problems and associated metrics that demonstrate the expressive capabilities of a framework and serve as a baseline for performance comparison. Standardized benchmarking has some known issues: certain metrics may not be relevant to every protocol; compilers may optimize for performance on benchmark problems rather than in the general case; and the performance measurement issues from Section 3.2.5 remain. We hope that careful design of benchmarking problems will mitigate these issues and provide a useful tool for practitioners in the future.

CHAPTER 4

Privacy Preserving Network Analytics

The 2008 financial crisis highlighted the fragility of existing financial networks and directly led to legislation, such as the Dodd-Frank Wall Street reform and consumer protection act [DF10], designed to identify and mitigate systemic risks. A core component of this legislation mandates that financial institutions, and banks in particular, meet capital requirements and engage in "stress-testing" against various hypothetical adverse scenarios. The effectiveness of such measures has understandably received considerable academic attention.[1]

Extant work commonly presumes that complete information about the network structure is available. This assumption is crucial because network-level dynamics depend on interactions between the institutions in the network, and emergent properties generally cannot be identified by examining each institution in isolation. In practice, however, complete information is *not* available to the institutions themselves. Each institution (presumably) knows its own assets, liabilities and interdependencies, but given these are commercially sensitive, they cannot be openly shared with the other—sometimes competing—institutions in the network.[2] What's more, security, liability and information leakage concerns can also distort information sharing incentives even in the presence of a "trusted" central agent.

For example, in 2015, the City of Boston and the Boston Women's Workforce Council (BWWC) launched an initiative to identify salary inequities. While data owners were willing to entrust salary and other sensitive data to a third party, "one of the major hurdles [...] was the unwillingness of any

[1]See, e.g., [AG98, AG00, Mor00, EN01, GK10, BEK$^+$11, GHM12, EGJ14, AOTS15, JP19].

[2]In practice, direct network information about the network may be so hard to come by that firms may have to resort to inferring network structures from coverage in the mainstream news [SZ19].

single entity (including a major local university, originally enlisted to perform the study) to assume the liability in case of leakage or loss of data entrusted to them." [BLV17]. [3] This fragmentation of knowledge presents a serious barrier to understanding basic properties of the network and can even make it impossible for individual members of the network to accurately carry out mandated stress tests, assess their own risk, or even compute their own market value. To the best of our knowledge, this tension between data privacy and regulatory oversight is highlighted in previous literature, but is not directly addressed.

In this chapter, we resolve this tension by leveraging cryptographic techniques from the multi-party computation (MPC) literature to compute privacy-preserving network analytics. We show how critical network-level statistics can be computed without any individual node revealing its private information to any third party, be it other nodes in the network, or even a central agent. As proof of concept, we deploy network analytics using MPC software tools using real and synthetic data. Below, we describe the problem and our contributions in more detail.

Stress-testing with limited information

In practice, many stress tests are performed individually by each firm (e.g. those mandated by the Dodd-Frank Act Stress Tests [DFA19]), but given the aforementioned informational limitations, these tests cannot adequately take account of network effects. This problem is highlighted in [JP19]:

> "...running stress tests for each bank separately overlooks a significant source of systemic risk. Indeed, without detailed information on the overall network, a bank-specific stress test does not capture the fact that a decrease in a bank's direct asset holdings is also likely to depress other banks' values, and hence also depress the value of its inter-bank assets."

For example, Figure 4.1 depicts a simple n-bank network where local stress testing reveals one failure, but in reality all but one of the institutions would fail. More specifically, assume bank 1 begins with reserve of 1 and all other banks have no reserves. The arcs between nodes represent liabilities between each bank. If bank 1's reserves drop by 10%, only bank 1 will fail its local stress test, whereas in reality, such a drop would cause $n-1$ cascading failures, and only bank n would

[3] Another reason institutions may be reluctant to share their data with a trusted party, is that it may still be at risk of subpoena [Eco14].

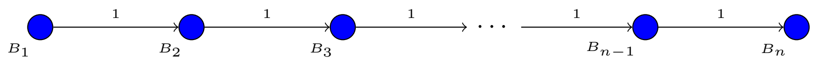


Figure 4.1: A n-bank example where local stress testing fails to identify systemic risk.

remain solvent given it has no liabilities.

Local stress testing can also fail to provide information about how to structure bailouts. Figure 4.2 gives an example of a simple 4-bank network (a scaled version of the one described in [JP19]) where local stress-testing gives insufficient information. Locally, banks 1 and 4 look similar in that they both have $4D$ of incoming debt, and $5D$ outgoing. Yet if both banks 1 and 4 fail (their reserves drop to 0) which bank should be bailed out? If we assume that bankrupt banks pay *none* of their debt,[4] then bailing out bank 1 by injecting D dollars saves banks 1, 2 and 3, whereas a bailout of D dollars to bank 4 saves *none* of the banks.

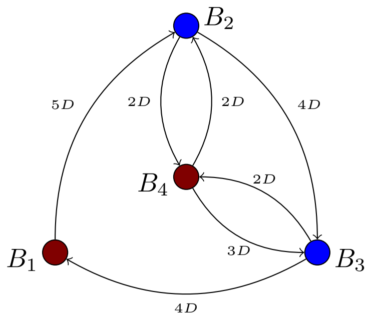


Figure 4.2: An example of a 4-bank network debt model adapted from [JP19]. Arrows point in the direction debt is owed.

Firm-level implications

The aforementioned limitations not only reduce the effectiveness of regulations imposed by a central entity, but can also have significant adverse effects on the individual institutions themselves. Consider that, depending on the network structure, each institution may not even be able to accurately assess its own market value without knowing the assets and liabilities of *all* other institutions! What's more, certain types of networks can have high *sensitivity*, which means that small imperfections in knowledge can lead to significant errors when assessing market values [FPR+18].

These examples show that even in the simplest, most idealized network settings, financial

[4]In the original debt model of [EN01], debts were paid proportionally. Other debt models (e.g. [GK10]) have assumed a zero recovery rate as in this example.

institutions cannot be expected to calculate (or even accurately estimate) their own market values. To make matters worse, the problem extends to most statistics of interest to the stakeholders. Complete, perfect knowledge of the network is also necessary to assess risk, or identify business strategies. Consider simple questions like: How many institutions would default if the underlying assets experienced a uniform drop of 10% in value? Or, how much would a given firm's market value increase if an asset class increased in value by 5%? These questions cannot be answered accurately by the institutions themselves without knowing the detailed portfolios of all the institutions in the network.

In brief, the privacy and security requirements of the individual institutions in the network prevents them from engaging in cooperative risk assessment and can be a serious practical barrier to institutions and central regulators alike.

Contributions

This chapter addresses some of the aforementioned issues by designing and implementing privacy-preserving MPC algorithms in two seminal network settings: the liabilities model of [EN01] (in Section 4.2) and the equities cross-holding model of [EGJ14] (in Section 4.3). We outline two contributions in particular: (i) The design of novel *data-oblivious* algorithms (see Definition 3) for computing market values in network settings under privacy preservation; (ii) a complete software implementation of our protocol, together with benchmarks of its computational and communication costs in real-world environments. We expand on each of these next.

(i) *Data-oblivious algorithms:* Data-oblivious algorithms guarantee that information isn't leaked by the algorithm operations themselves. Most existing MPC protocols focus on hiding the *data* in a computation, rather than the *operations* (see Section 4.1.2 for further discussion). Thus, before an algorithm can be implemented securely, its sequence of *operations* must be made data independent. Previous works have avoided this problem by focusing on simple algorithms (e.g. securely computing means and variances) where the sequence of arithmetic operations is fixed, and hence independent of the input data. In our setting, the algorithms for computing market clearing values in network settings are significantly more complex, and the algorithms developed in the extant literature (outside the context of secure computation) are *not* data-oblivious.

(ii) *Robust multi-party implementation:* The benefits of using MPC for privacy-preserving risk assessment is discussed in recent literature [AKL12, FKOS13, CK19]. These works, however, do not specifically focus on network settings, do not provide implementations, nor do they seek to design concrete protocols with realistic network data. Closer to our setting is [NPH14], who conduct a preliminary study showing the feasibility of using MPC to compute stress tests in financial networks. In contrast to this work, however, they do not seek to develop a globally secure algorithm nor benchmark it on real network data.[5] Our second main contribution—a robust multi-party implementation—shows that our data-oblivious algorithms are indeed practical.

The computational and communication costs of MPC protocols is typically a barrier to adoption. The exact cost of these protocols has a complex dependency on the underlying algorithms, the software implementation as well as the computing hardware and networking infrastructure, thus the best way to assess the practicality of an MPC algorithm is to actually implement and benchmark it. To this point, we implement and benchmark our algorithms using three different open-source software packages (each with their own advantages and disadvantages) and evaluate their suitability for practical implementation: SCALE-MAMBA [AKR$^+$19], MPyC [Sch] and EMP-Toolkit [WRK17b].

Cryptographic methods also have the potential to resolve some of the tension between privacy and regulatory oversight [FKOS13, AKL12]. There are two distinct privacy issues related to securely computing analytics on private data: *computing* the statistics and *revealing* the results. The first issue, privacy-preserving computation, is addressed by secure multi-party computation (MPC). The second issue, revealing the results without compromising privacy, is addressed through tools like differential privacy [Dwo11]. The work of [FKOS13] focuses more on differential privacy, whereas the work of [AKL12] focuses on secure computation, but neither work proposes concrete network analytics in a realistic network model, and neither includes any implementation.

Related practical applications. The past few years marked a dramatic increase in the number of *applications* and *implementations* of MPC protocols, and we highlight some of the recent applications

[5]To avoid the costs of doing a *global* secure computation, [NPH14], breaks the network until smaller groups of nodes, and performs secure computation within the groups. This leaves the system vulnerable if a small number of institutions *within a group* collude, and it leaks information about debts between the groups. In addition, the protocol leaks information about the whether debts exist between institutions (while hiding their magnitude).

of MPC towards privacy-preserving financial analytics here. For instance, continuing the example mentioned earlier, the city of Boston together with the Boston Women's Workforce Council used MPC to calculate aggregate payroll analytics across over 150 different companies in Boston, with the aim of studying gender-based discrepancies in employee compensation [LJDA$^+$18].

Protocols have also been developed for linking sales and purchase records in Estonia in order to identify potential cases of tax fraud without revealing the underlying sales and purchase data [BJSV15, BJSV16]. [SvA$^+$19] used homomorphic encryption, a different model of secure computation, to identify fraudulent transactions in banking networks.

This work contributes to this literature by providing a novel set of generalizable MPC algorithms for network applications and evaluating the feasibility of implementing these algorithms using real data. More broadly, our work contributes to a recent and growing literature studying the implications of new technologies in finance (FinTech). While the bulk of this literature focuses on blockchain/cryptocurrency and crowdfunding applications,[6] privacy-preserving technology is being increasingly adopted in practice, as the computational methods and algorithms underlying it continue to improve. This chapter, much like [AKL12], seeks to expand the boundaries of the extant fintech literature into this novel direction.

To summarize, the methods we develop can help resolve the tension between the privacy requirements of the individual institutions and the collective value of aggregate network statistics in virtually any type of network setting. Our work helps to bridge the gap between our extant theoretical models of financial networks—that generally assume complete information—and the real world—where information sharing is hindered by privacy concerns.

4.1 MPC and network algorithms

As previously discussed, secure multi-party computation is a cryptographic technique that, in theory, allows a group of data owners to securely compute any function of their private data, without revealing their underlying data to each other or to any outside party, and without the need for a central agent. This section addresses practical issues that arise when using MPC in a network setting. We describe the outsourced computation model, which reduces communication complexity but increases

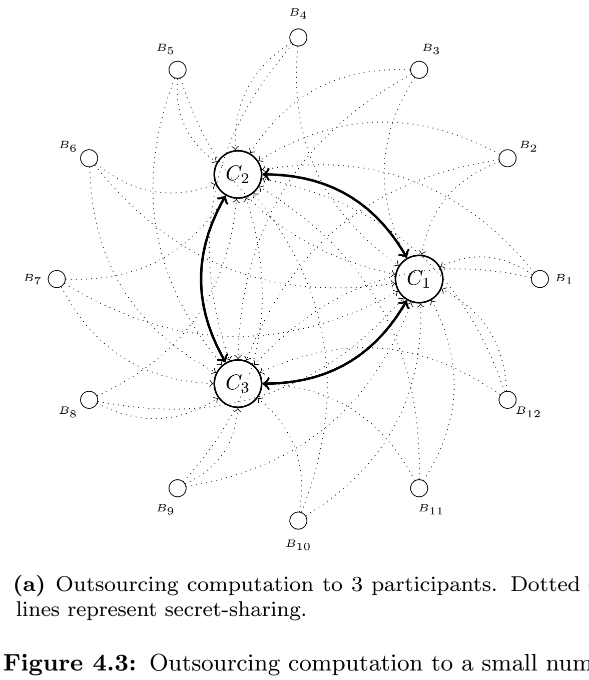
[6]For instance, see [BMT20a, TMB19, BMT20b, BBBC19, CS20, CL20, CTT$^+$20, CLW20, GTN20, HJS19, RS20, TF20]

the vulnerability of a computation to collusion. We also describe common implementation challenges, including the data-oblivious model.

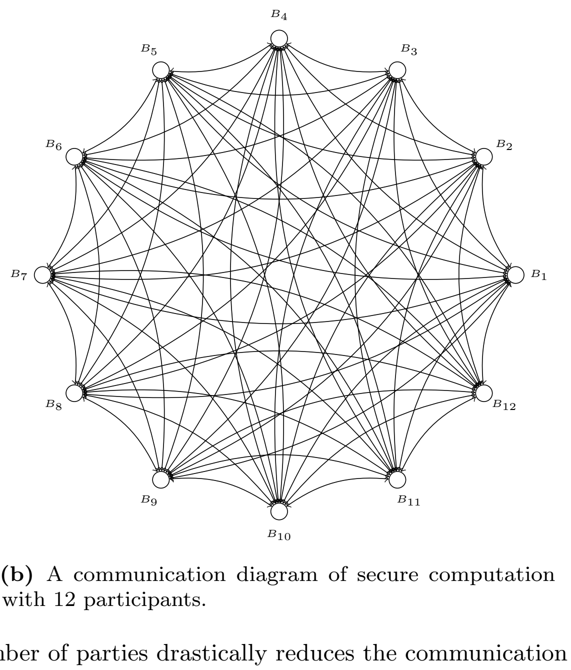
4.1.1 Outsourced computation

Secure computation protocols require repeated rounds of communication between *all* of the participants. Thus, if there are n participants in the protocol, the communication complexity grows as $O\left(n^2\right)$. To avoid this, it is common to use secret sharing to decouple the number of participants in the computation from the number of participants contributing (private) information.

For example, in a network with 1000 banks, the banks could choose 3 of the larger banks to be the "computation parties", and secret-share all private inputs to these three privileged banks. The three banks could then engage in a secure computation to compute the vector of market values. Since the secure computation only requires three participants (instead of 1000), the reduction in communication could speed up the overall computation. The drawback is that this type of outsourcing changes the security guarantees, and collusion between 2-out-of-3 of the computational parties could break the security (instead of collusion between 501 of the input parties). The different architectures are outlined in Figure 4.3. We will use this method in some of our implementations in Section 4.4.



(a) Outsourcing computation to 3 participants. Dotted lines represent secret-sharing.



(b) A communication diagram of secure computation with 12 participants.

Figure 4.3: Outsourcing computation to a small number of parties drastically reduces the communication complexity of the protocol but makes it more vulnerable to collusion.

4.1.2 Obstacles to privacy preservation

In principle, *any* function can be computed securely using one of the early MPC protocols (e.g. [GMW87, BGW88, CCD88]). Unfortunately, there are many obstacles to deploying MPC in *practice*.

Efficiency. Early MPC protocols were efficient in the sense that they ran in *polynomial time*, but the actual cost (in terms of computation and communication) was too high for practical applications. Decades of algorithmic improvements, coupled with a steady increase in raw computing power has made MPC practical for a variety of real-world problems. Nevertheless, efficiency remains a primary obstacle when developing and deploying novel MPC protocols. The practical efficiency of MPC protocols depends on a number of factors beyond the basic algorithms, including computing hardware, networking environment, software implementation, and the topology of the circuit being executed. This means that effectively measuring the efficiency of an MPC protocol requires actually implementing and benchmarking the protocol under real-world conditions.

Implementation. Despite progress in the last decade, implementing MPC protocols remains a significant engineering challenge. These are multi-participant networked protocols that require several rounds of communication interleaved with complex local cryptographic operations. These challenges are explored in greater detail in Chapter 3.

However, the network setting requires multi-party computation, which eliminates the many two-party frameworks. Only a few frameworks support computations with more than two participants, and we focus on three of the most recent works: EMP-toolkit, [WRK17b], SCALE-MAMBA [AKR$^+$19], and MPyC [Sch].

Designing data-oblivious algorithms. In general, MPC protocols are designed to execute algorithms that are *data-oblivious*, *i.e.*, algorithms whose control flow does not depend on their (private) inputs [MZ14]. Formally:

Definition 3 (Data-oblivious algorithms)**.** *An algorithm is called* data-oblivious *if the* control-flow *(*i.e.*, the sequence of computations and memory accesses made by the algorithm) is independent of the algorithm's inputs.*

For example, if an algorithm's runtime depends on the input data, executing the algorithm leaks information about its inputs, even if the internal state of the algorithm is hidden. Developing a compact, data-oblivious representation of a given functionality or algorithm is a fundamental challenge that must be overcome before executing a computation securely.

One method for ensuring that an algorithm is data-oblivious is to represent it as a *circuit*, either a boolean circuit (consisting of AND, XOR and NOT gates), or an arithmetic circuit (consisting of addition and multiplication gates over a finite field). Most secure computation frameworks (e.g. [Yao82, Yao86, GMW87, BGW88, CCD88]) take this approach, and these frameworks can only securely execute circuits.[7]

Although, in principle, any algorithm can be represented as a circuit, the circuit representation may be large, and finding the *minimum* circuit representation of an algorithm is difficult [HOS18]. The development of MPC compilers (described above) represent a crucial step toward making secure computation usable in practice. However, these compilers still require the user to provide a *data-oblivious* algorithm. For instance, many compilers do not allow conditionals on private data; instead they provide syntax for multiplexing. Consider the simple conditional program on the right, which can be rewritten without a conditional if x is a boolean:

```
if $x = 0$ then
    $y = z$
else
    $y = w$
end if
```

$$\iff \qquad y = x \cdot w + (1 - x) \cdot z$$

Comparisons can be handled similarly. If x and y are boolean values, then the comparison operation is equivalent to two boolean operations:

$$x < y \iff \neg x \wedge y.$$

Comparisons between integers can be handled bit-by-bit. Similarly, few compilers support "for" loops with *private* input bounds.

[7]There has been significant work developing MPC protocols for RAM-model computations, but these protocols are extremely complex, and are only useful in settings which are particularly ill-suited to circuit-model computations [LO13, LHS$^+$14, BCP15, GGMP16].

4.2 The Eisenberg & Noe network model

Before diving into the details of the debt and equity models in the following sections, we note that the underlying methods we develop could be applied to compute network statistics in essentially any network model. In this document, we apply the methods to the debt model by Eisenberg and Noe [EN01] and the equity model by Elliot, Golub, and Jackson [EGJ14], which generalize under the Jackson and Pernoud model [JP19]. Other applicable settings include time-based models that consider both short- and long-term investments [AG98, AG00, AOTS15], local contagion models [Mor00], and other debt and equity based models [Els11, GHM12].

4.2.1 The Eisenberg & Noe (2001) debt model

Consider a system with n financial institutions that have *reserves* and *debts*. Institution i's reserves are represented by e_i, and the debts (liabilities) are represented by L_{ij}, indicating a debt from institution i to institution j of L_{ij} dollars. Thus the entire network is a weighted, directed graph, where financial institutions are represented and debts are represented as (weighted, directed) edges between the institutions.

Define $\bar{p}_i$ to be the total liabilities of institution i, *i.e.*, $\bar{p}_i \overset{\text{def}}{=} \sum_j L_{ij}$, and the proportional liabilities

$$\Pi_{ij} = \begin{cases} \frac{L_{ij}}{\bar{p}_i} & \text{if } \bar{p}_i > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

In this proportional specification, failed or bankrupt institutions make partial payments on their outstanding debts, and the same proportion of every debt is assumed to be paid.

The total amount of money in the system is the sum of the reserves of all the stakeholders, and in this simple model, money is neither created nor destroyed. If every institution has enough reserves to pay all its debts, then each institution's market value can be computed locally by adding its reserves and incoming debt obligations and subtracting its outgoing debts. On the other hand, if some institutions fail (i.e., they do not have enough money to cover their debts) other institutions that may appear to be solvent (based on their local view) may actually be insolvent (if their incoming debts are not paid in full). This highlights the essential networked nature of the stress test – *institutions* *cannot effectively assess their risk based on knowledge of their debts alone.*

Formally, the action of institution i can be characterized by its total payment, p_i. We use $\mathbf{p} \in \mathbb{R}^n$ to denote the vector of *total* amounts paid by each institution. If $p_i < \sum_j L_{ij}$ then institution i is said to *fail* or *default*. Assuming that defaulting institutions must pay as much as they can, we have the following equations.

$$p_i = e_i + \sum_j p_j \Pi_{ji} \text{ for defaulting institutions} \tag{4.2}$$

$$p_i = \bar{p}_i \text{ for non-defaulting institutions.} \tag{4.3}$$

Computing the market-clearing vector absent privacy concerns

In this model, every institution will have a uniquely-defined equilibrium market value, based on its remaining reserves after all possible debts are settled. Assuming all information is public, [EN01] suggest a simple, iterative algorithm for calculating the market clearing vector.

Algorithm 1 Calculating the equilibrium payment vector in a network [EN01]

```
 1: Input: A network with assets $\mathbf{e} \in \mathbb{R}^n$, and debts $\mathbf{L} \in \mathbb{R}^{n\times n}$
 2: Initialize defaulter list, $D = \emptyset$
 3: Initialize payment vector $\mathbf{p} = \bar{p}$        ▷ Everyone attempts to pay full debts
 4: Compute proportional liabilities with Equation (4.1)
 5: Update the defaulter list assuming payments $\mathbf{p}$
 6: if Defaulter list grows then
 7:     Update $\mathbf{p}$ using Equations 4.2 and 4.3
 8:     Go To Line 5
 9: else
10:     return $\mathbf{p}$
11: end if
```

From here, a natural network-scale stress-test can be run by adjusting (decreasing) the underlying reserves held by different institutions and calculating the number of failures (or the financial shortfall) that arise in equilibrium. This type of stress testing, however, requires knowledge of the debt structure of all institutions in the network. Performing this type of stress test using standard tools would then require all institutions to share all the details of their reserves and liabilities with some centralized regulator or testing authority who could perform the test.

What's more, this model is very sensitive to small changes in the debt structure [FPR$^+$18], so high-fidelity data about the debt structure is required for effective stress testing. In particular, regulators who have only rough estimates of the inter-bank liabilities cannot effectively compute the equilibrium market values in the network.

4.2.2 Identifying data dependencies in Algorithm 1

The [EN01] Algorithm 1 was not designed with privacy in mind, and thus is not data-oblivious. For example, in line 6, the algorithm terminates when the defaulter list stabilizes. Thus the number of iterations of the algorithm depends on the (private) debt-holdings. More subtly, Line 8 traditionally requires a matrix inversion (an operation useful in many network settings), and most algorithms for matrix inversion (e.g. row-reduction) are *not* data-oblivious.

Overall, there are three main issues that we need to overcome in order to make the algorithm data oblivious.

1. **Line 5**: How can the defaulter list be updated when the current payment vector $\mathbf{p}$ must remain private? The current payout vector depends on the debts of all the participants in the network, so updating the defaulter list requires testing whether an institution fails under a *private* payment vector $\mathbf{p}$.

2. **Line 6**: A data-oblivious algorithm cannot change its control flow based on private information, so when calculating the clearing vector, the termination condition cannot depend on the set of failures.

3. **Line 8**: How can the current payout vector, $\mathbf{p}$, be updated obliviously? Updating the payout vector $\mathbf{p}$ requires solving Equation 4.8, which depends on the current payout vector, as well as the current set of failed institutions, both of which must remain private.

One implication of making an algorithm data-oblivious is that its running time becomes independent of the underlying data. For instance, when calculating the market values of a large network, if all the banks had zero liabilities and zero assets, a traditional algorithm might take a "shortcut" and return that all market values are zero. On the other hand, a data-oblivious algorithm *must* perform exactly the same computations when the inputs are all zero as it does on real-world inputs. One consequence of this is that all the running times we report would be the same whether we used real-world data, or simply ran the algorithms on an all-zero input. This is an instance of sacrificing efficiency for security – in the real-world, both software and hardware implement computational “shortcuts” that can drastically improve performance when the inputs satisfy certain criteria. Unfortunately, if these shortcuts cannot be applied to *all* inputs, the fact that a shortcut was taken leaks information about the inputs, and hence cannot be applied in a privacy-preserving manner. In other words, data-oblivious algorithms can never be faster than the “worst-case” running time of its non-oblivious counterpart.

4.2.3 A data-oblivious version of Algorithm 1

In this section, we describe the algorithms we develop for computing the defaulter list in a *data-oblivious* manner and explain how we can address the issues raised in Section 4.1.2. In general, such an analysis examines the *control flow* of the program, including loops with data-dependent bounds or early-termination conditions and conditionals (“if” statements) whose conditions depend on input values. To make an algorithm data-oblivious, we remove data-dependent optimizations, rewrite loops to use a fixed number of iterations, avoid conditionals, or rewrite them to evaluate both branches and multiplex the results. These changes can be inefficient compared to a data-dependent program: the upper bound for a loop might be large, and the complexity of a multiplexer can grow exponentially if the conditionals are nested. For example, in Gaussian Elimination, determining the next pivot location requires testing whether an element is zero.

Notably, we do not have to modify simple arithmetic operations (sums and products), since these directly translate to the circuit model.

The algorithms in this section do *not* provide security by themselves—they are only *data-oblivious*. In order to provide security, the operations required (additions, multiplications, etc.) at each step of the algorithms need to be executed securely e.g. using the open-source MPC protocols described in Section 4.1.

Recall the payment equations (4.2) and (4.3) from Section 4.2. These can be written more succinctly as

$$p_i = \min\left(\bar{p}_i, e_i + \sum_j p_j \Pi_{ji}\right). \tag{4.4}$$

For a given payment strategy, $\mathbf{p}$, we can define the defaulting matrix

$$\Lambda(\mathbf{p}) = \begin{cases} 1 & \text{if } i = j \text{ and } i \text{ defaults under } \mathbf{p} \\ \\ 0 & \text{otherwise.} \end{cases} \tag{4.5}$$

If complete information about the network (including all reserves and debts) were known, then the equilibrium values of each institution, the total number of failures and the total shortfall could be computed using Algorithm 1.

As noted in Section 4.1.2, the three major obstacles to making Algorithm 1 data-oblivious are: updating the current payout vector; updating the defaulter list; identifying termination conditions. Below, we outline the methodology we suggest to overcome these obstacles, culminating in Proposition 2 which shows that our solution efficiently computes an approximation to the true clearing vector and market valuations.

Payout vector. For a fixed defaulting set given by Λ, the equilibrium payouts are obtained by solving the linear equations

$$\mathbf{p} = \Lambda \left[\Pi^T \left(\Lambda \mathbf{p} + (I - \Lambda)\bar{p}\right) + \mathbf{e}\right] + (I - \Lambda)\bar{p}, \tag{4.6}$$

where Λ is the diagonal matrix with $\Lambda_{ii} = 1$ if and only if institution i has failed, Π is the $n \times n$ proportional liability matrix, $\bar{p}$ is the debt vector, and $\mathbf{e}$ is the vector of underlying assets. Given a fixed defaulting set, an equilibrium payout strategy is a solution to Equation (4.6). Rewriting this equation, we have

$$\mathbf{p} = \Lambda\Pi^T\Lambda\mathbf{p} + \Lambda\Pi^T(I - \Lambda)\bar{p} + \Lambda\mathbf{e} + (I - \Lambda)\bar{p}. \tag{4.7}$$

Defining

$$\begin{aligned} \mathbf{A} &\overset{\text{def}}{=} \Lambda\Pi^T\Lambda \\ \mathbf{b} &\overset{\text{def}}{=} \Lambda\Pi^T(I - \Lambda)\bar{p} + \Lambda\mathbf{e} + (I - \Lambda)\bar{p}, \end{aligned}$$

Equation (4.7) becomes

$$\mathbf{p} = \mathbf{A}\mathbf{p} + \mathbf{b}. \tag{4.8}$$

This makes $\mathbf{p}$ a *fixed point* of the transformation defined by $\mathbf{A}$ and $\mathbf{b}$: it maps onto itself. This has the solution

$$\mathbf{p} = (\mathbf{I} - \mathbf{A})^{-1}\,\mathbf{b}. \tag{4.9}$$

Traditional matrix inversion algorithms such as Gaussian Elimination are not data-oblivious, and thus cannot be implemented directly as a secure computation. Instead, Algorithm 2, below, outlines a simple, iterative method for (approximately) solving Equation (4.8). Subsequently, in Proposition 1, we show that Algorithm 2 will converge to the correct result given enough iterations or under special circumstances. We report on empirical convergence speeds in Section 4.4.2. Algorithm 2 is data-oblivious in the sense of Definition 3 because the number of rounds of the "for"-loop, k, does not depend on the inputs $\mathbf{A}, \mathbf{b}, \mathbf{p}^0$. The matrix and vector arithmetic is data-independent, since it derives entirely from basic arithmetic.

Algorithm 2 FindFix

```
 Input: $\mathbf{A}, \mathbf{b}, \mathbf{p}^0$
for $i = 1, \dots, k$ do
    $\mathbf{p}^i = \mathbf{A}\mathbf{p}^{i-1} + \mathbf{b}$
end for

return $\mathbf{p}^k$
```

Proposition 1 (FindFix Algorithm Properties). *(i) Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be an matrix with spectral radius $\rho \overset{def}{=} \rho(\mathbf{A})$. If $\rho < 1$ then $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ has a unique solution, $\mathbf{x}^*$, and Algorithm 2 will return an approximation, $\mathbf{y}$, such that as the number of iterations, $k \to \infty$, $\mathbf{y} \to \mathbf{x}^*$. In addition, if $\mathbf{p}^0 \geq \mathbf{x}^*$, then $\mathbf{p}^k \geq \mathbf{x}^*$ for all k. (ii) In an acyclic debt network with n nodes, Algorithm 2 (with $k = n$) solves Equation* (4.8) *with* no *error.*

Defaulter list. Next, to deal with Obstacle 1, we develop Algorithm 3, below, which takes a proportional liability matrix $\mathbf{\Pi}$, an asset vector $\mathbf{e}$, and a proposed payout vector $\mathbf{p}$, and calculates which institutions will fail under the proposed payout vector. If institution i fails, Λ_{ii} is set to 1. The loop in Algorithm 3 has a public number of iterations n, and the conditional can be evaluated using a multiplexer, as described in Section 4.1.2. Thus, the whole algorithm is entirely data-oblivious.

Algorithm 3 UpdateLambda

```
 Input: $\mathbf{\Pi}, \mathbf{e}, \mathbf{p}$
for $i = 1, \ldots, n$ do
    if $p_i > e_i + \sum_j p_j \Pi_{ji}$ then
        $\Lambda_{ii} = 1$
    else
        $\Lambda_{ii} = 0$
    end if
end for

return $\Lambda$
```

Market values (main algorithm). Putting these pieces together, we propose Algorithm 4 to compute the market values of the institutions in the debt network model obliviously. To overcome the early termination problem (Obstacle 2) we run the algorithm for n steps, independent of the inputs, and show in Proposition 2 that after n steps the defaulter list will *always* have stabilized.

Algorithm 4 A data-oblivious algorithm for calculating the market values in a debt network.

```
Input: $\mathbf{e} \in \mathbb{R}^n$, $\mathbf{\Pi} \in \mathbb{R}^{n \times n}$
Initialize $\Lambda = \mathbf{0} \in \mathbb{R}^{n \times n}$                    ▷ No defaults yet
Initialize $\mathbf{p} = \bar{p}$                                  ▷ Everyone attempts to pay full debts
for $i = 1, \ldots, n$ do
    $\mathbf{A} = \Lambda \Pi^T \Lambda$
    $\mathbf{b} = \Lambda \Pi^T (I - \Lambda)\bar{p} + \Lambda \mathbf{e} + (I - \Lambda)\bar{p}$
    $\mathbf{p} = \text{FindFix}(\mathbf{A}, \mathbf{b}, \mathbf{p})$                  ▷ Alg 2: Find equilibrium
    $\Lambda = \text{UpdateLambda}(\mathbf{\Pi}, \mathbf{e}, \mathbf{p})$            ▷ Alg 3: Update defaulter list
end for

for $i = 1, \ldots, n$ do                                ▷ Compute market value:
    $v_i = e_i + \sum_j p_j \Pi_{ji} - \bar{p}_i$                  ▷ assets + payments - liabilities
end for

return $\mathbf{v}$
```

Remark 1. *For financial stress testing, it may be desirable to calculate the number of defaulting institutions or the total shortfall of the system, rather than the market values of the institutions. Since the number of shortfalls is simply* $|\{v_i \mid v_i < 0\}|$*, and the total shortfall is* $\sum_{v_i<0} v_i$*, these statistics can easily be calculated with only slight modifications to Algorithm 4.*

Proposition 2. *(i) Algorithm 4 is data-oblivious and computes an approximation to the market values in an $n \times n$ debt network using only $(k+3)n^3 + 6n^2$ multiplications and n^2 comparisons. (ii) If there are no defaulters, Algorithm 4 will converge to the* exact *solution, when $k = 1$.*

Algorithm 4 has no conditionals and the two loops (Steps 4 and 11) both have fixed bounds (n). The remaining steps are either sub-algorithms that we have already demonstrated oblivious (FindFix and UpdateLambda) or are simple arithmetic (Steps 5, 6 and 12). Thus, the entire algorithm is data-oblivious.

Algorithm 4 requires $O\left(kn^3\right)$ (secure) multiplications, because it makes n calls to FindFix, which executes k $n \times n$ matrix multiplications (which takes $O\left(kn^2\right)$ operations using schoolbook matrix arithmetic). These multiplications dominate the cost of executing Algorithm 4 securely and motivate the development of a reduced-complexity FindFix algorithm, which we do in Section 4.4.3.

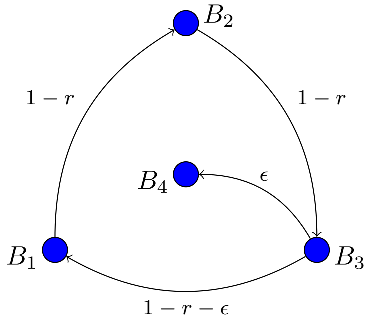
4.3 The equity cross-holdings model

In this section, we develop privacy-preserving algorithms in an alternative model of financial networks, in which institutions to hold shares of other institutions rather than debt contracts [EGJ14]. The algorithms rely on the same methods developed for the debt model in Section 4.2.3; the overall conclusion of this section is that the methods remain applicable in alternative network settings.

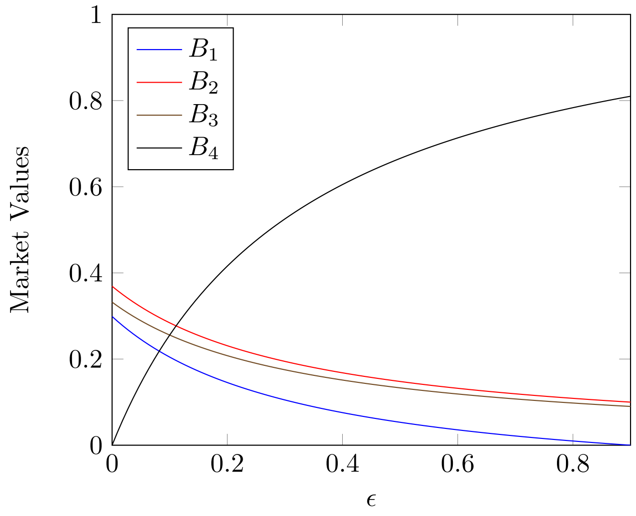
In this alternative model, each institution represents a node in the network and institutions are connected by proportional *cross-holdings*, which are represented as weighted, directed edges in the graph. An edge from institution i to institution j of weight p (with $0 \leq p \leq 1$) represents the statement that institution i owns a p-fraction of institution j. In addition to these proportional cross-holdings or shares, each bank can have some underlying assets, and these assets are the sole source of value in the network. These assets could represent tangible, physical assets or any investment the institution has made outside the network. This model can also be viewed as a flow, where at each time step money flows between institutions, and the total incoming flow to a node is divided among its outgoing edges according to their edge weights. With this intuition, the "market value" of a bank is the steady-state flow through that institution.

The possibility of cycles in the network (where institution a owns shares of b, who owns shares of c, who owns shares of a) means that the market value of institutions can be extremely sensitive to small changes in the cross-holdings [HK16]. In fact, if institutions can have arbitrarily small self-ownership, then an ϵ-change in cross-holdings can have arbitrarily large impact on market values. Figure 4.4a illustrates this in a simple network with 4 banks. The banks all have self-holdings ("reserves") of r, and the cross-holding values reported in the figure. Focusing on Bank 2, one can see that this bank would need to know the value of ϵ to be able to determine its own market value. If Bank 2 has an outside asset with value 1, and no other banks have any external assets, one can readily compute the vector of market values in closed form.[8]

In Figure 4.4b, we show the final market value of each bank, assuming each bank retains $r = 10\%$ self-holdings. The key point is that the value of Bank 2 drops significantly from .37 to .1 as ϵ goes from 0 to r, yet Bank 2 has no direct knowledge of ϵ.



(a) Four-bank network.



(b) Market values of the banks in Figure 4.4a.

The high sensitivity of market values to small changes in cross-holdings means that institutions must have a near perfect view of all other institutions' cross-holdings in order to calculate their own market value! In Section 4.3.1, we give Algorithm 5 for securely computing the approximate market

[8]Namely, the market value vector is given by

$$\mathbf{v} = \frac{1}{(\epsilon + r - 1)(1-r)^2 + 1} \begin{bmatrix} (\epsilon + r - 1)(r-1)r \\ r \\ r(1-r) \\ \epsilon(1-r) \end{bmatrix}. \tag{4.10}$$

The sum of all market values is 1 (which is the total value of all external assets).

values of each institution *without* requiring the institutions to share their cross-holdings.

More formally, we will use the notation C_{ij} to denote the fraction of institution j owned by institution i. We will use $n \times n$ matrix $\mathbf{C} = (C_{ij})$ to denote the cross-holdings within the network. As in [EGJ14], we define $C_{ii} = 0$. The matrix $\mathbf{C}$ corresponds to the weighted, directed graph on n vertices that has an edge from j to i with weight C_{ij} whenever $C_{ij} > 0$. With this notation, $\sum_i C_{ij}$ represents the proportion of j that is owned by other institutions. We assume that $\sum_i C_{ij} < 1$, *i.e.*, that each institution retains some proportion of self-ownership. We define $\mathbf{S}$ to be the diagonal matrix with $S_{jj} \overset{\text{def}}{=} 1 - \sum_i C_{ij}$ to be the fraction of self-ownership of institution j. We will use D_i to denote the value of the assets of institution i, and the vector $\mathbf{D}$ to denote the vector of asset values.

This type of model leads to two different notions of valuations of an institution, the *equity* valuation, and the *market* valuation [BBC89]. The equity valuation of institution i is denoted by

$$V_i = \underbrace{D_i}_{\text{Value of assets held by } i} + \underbrace{\sum_j C_{ij} V_j}_{\text{Values of institutions held by } i} \tag{4.11}$$

In matrix notation, this becomes

$$\begin{aligned} \mathbf{V} &= \mathbf{D} + \mathbf{C}\mathbf{V} \\ \mathbf{V} &= (\mathbf{I} - \mathbf{C})^{-1} \mathbf{D}. \end{aligned} \tag{4.12}$$

As noted in [EGJ14], since the columns of $\mathbf{C}$ sum to strictly less than one, the matrix $\mathbf{I} - \mathbf{C}$ is guaranteed to be invertible. (Actually, $\mathbf{I} - \mathbf{C}$ is an M-Matrix, see [Wil77, Joh82]).

It is relatively straightforward to see that the equity valuation significantly overcounts the underlying value of the system, since the value of each asset is counted towards its primary owner institution as well as the shareholders in that institution. Formally, this means $\|\mathbf{V}\|_1 \geq \|\mathbf{D}\|_1$. This property of the equity valuation is well-known [FP91, FHT94].

The *market* valuation eliminates this overcounting, by scaling each institution's equity value by its percentage of self-ownership. Thus the market valuation of institution i is

$$\mathbf{v} = \mathbf{S}\mathbf{V} = \mathbf{S}(\mathbf{I} - \mathbf{C})^{-1}\mathbf{D}. \tag{4.13}$$

4.3.1 Privacy-preserving equity model

As with the debt model, Equation (4.13) was not designed with privacy in mind. Happily, it does not introduce any new types of data dependencies not seen in the previous analysis. The only data dependency in the original algorithm is the matrix inversion step, since, as mentioned, traditional matrix inversion algorithms are not oblivious. Fortunately, we can use the same solution: an iterative method to find a fixed point, described in Algorithm 2. This leads to the following algorithm (Algorithm 5) for computing the market values in an equity network.

Algorithm 5 An iterative algorithm for calculating the market values in a equity cross-holdings network model.

```
1: Input: $n \times n$ cross-holdings matrix $\mathbf{C}$
2: Input: $n \times 1$ asset vector $\mathbf{D}$
3: Input: $n \times n$ diagonal self-holding matrix $\mathbf{S}$
4: $\mathbf{V} = \textsc{FindFix}(\mathbf{C}, \mathbf{D}, \mathbf{0})$
5: $\mathbf{v} = \mathbf{S}\mathbf{V}$
6:
7: return v
```

Proposition 3. *If* $\mathbf{C}, \mathbf{S}, \mathbf{D}$ *is an equity-network with mininum self-holdings* $s_{\min} \overset{def}{=} \min_i S_{ii} > 0$*, then Algorithm 5 gives an approximate solution* $\mathbf{v}$ *to the true market values* $\mathbf{v}_{\text{true}}$ *with*

$$\|\mathbf{v} - \mathbf{v}_{\text{true}}\|_1 < (1 - s_{\min})^{k+1}\mathbf{D}, \tag{4.14}$$

using only $kn^3 + n^2$ *additions and* $kn^3 + n^2 + n$ *multiplications.*

4.4 Implementations with real data

Secure computation protocols can be extremely resource intensive, with high processing and bandwidth costs. In fact, the costs (real or perceived) of running MPC protocols has been a significant barrier to adoption. To assess the feasibility of securely computing financial analytics, we implemented and benchmarked Algorithm 4 using three different software frameworks for MPC: SCALE-MAMBA [AKR$^+$19], EMP-toolkit [WMK16], and MPyC [Sch]. Each of these frameworks uses different cryptographic techniques and has different security guarantees which lead to different

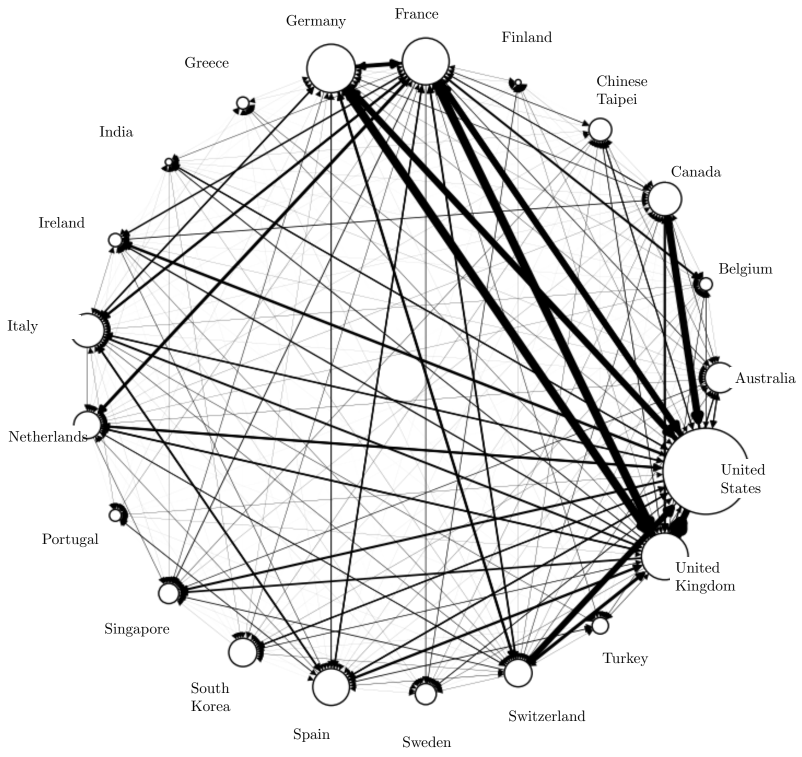


Figure 4.5: The full 21-country BIS dataset.

performance characteristics.

All the benchmarks reported in this section were collected on a single 15-core machine with 3.6GHz Intel Xeon Gold 5122 processors and 128GB of RAM. We ran all parties locally on this single machine which implies we had to simulate n instances of individual machines running the MPC protocols. While this limits the scale of the computations we are able to perform, it is important to note that in practice, the computational burden would be split across the parties of the network, each thus requiring a fraction of the computational power of our own setup.

4.4.1 Data sources

The Bank for International Settlements (BIS) is an international institution that collects and reports consolidated financial statistics on international claims and liabilities. Each quarter, a small set of (typically 25-30) countries report statistics relating to a larger set of several hundred countries.

We use these statistics to model a financial network in the debt model (Section 4.2.1). In this setting, network participants are countries, cross-holdings are international claims on other participants, and cash reserves are total domestic capital/equity.

In order to provide the complete liability network, we limit network participants to only the reporting countries who provide statistics on their total assets and equity. We used data for quarter 2 of 2018 but our data parsing tools can be used for any time period. This left us with 21 countries, as shown in Figure 4.5. Cross-country liabilities and represented by arrows, with darker arrows representing higher debts. The node size represents the country's underlying capital.

To model smaller networks, we worked with subsets of the BIS data set, and to model larger networks, we generated synthetic data. Note that a crucial feature of all secure computation algorithms is that the running times and memory usage depends only on the *size* of the network, and not the actual cross-holdings, since any dependence on cross-holdings would constitute a data leakage. Thus the estimates of running time and memory usage will be the same for *any* collection of real or synthetic data. In particular, this means our resource usage measurements are consistent across real and synthetic data as long as our generated data is the same size as the real data (*i.e.*, the values can be represented using 64-bit fixed point numbers, and the number of participants is the same).

Table 4.1 shows the full 21-country network we extracted from the BIS data set. To test on smaller networks, we considered the subnetworks obtained by the considering only the first t banks in the table for $t \leq 21$. The equilibrium market values are computed using the non privacy-preserving Algorithm 1. The privacy-preserving market values are computed using our data-oblivious Algorithm 4. There is only one defaulter in this network, so it is not surprising that the error term is zero (Proposition 2(ii) shows that if there are *no* defaulters the error will always be 0).

In addition to the BIS data set, in some of our implementations we also considered the 6-country network consisting of France, Germany, Greece, Italy, Portugal and Spain analyzed in [EGJ14],[9] visualized in Figure 4.6.

4.4.2 Correctness of the oblivious algorithms

Rate of convergence of Algorithm 2. The MPC algorithms we develop in Section 4.2.3 are error-free, in the sense that the privacy-preserving adjustments do not introduce any bias in the final

[9]Although [EGJ14] analyzed this as an *equity* network, the underlying data consisted of *debts*. In that work, they converted the underlying debt network to an equity network by assuming each country retained 2/3 self-holdings, and normalizing the outstanding debts.

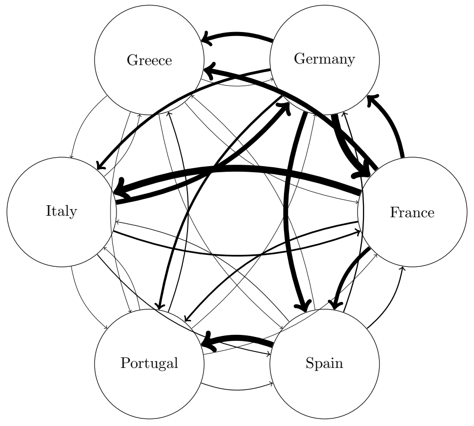


Figure 4.6: Six-country network [EGJ14].

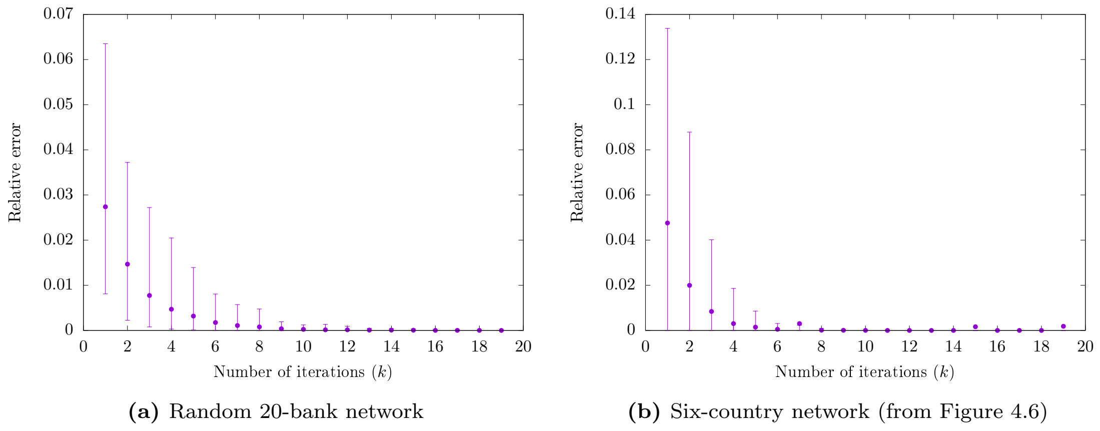


Figure 4.7: FindFix Algorithm 2 convergence over k.

output, with one exception: Algorithm 2, which replaces matrix inversion, is only asymptotically exact.

Figure 4.7 shows the algorithm's relatively quick rate of convergence in two different types of networks. Figure 4.7a considers a set of (synthetic) random 20-bank networks, and shows how the error in calculating market values decreases as a function of k, the number of iterations in Algorithm 2. The networks were generated with L_{ij} for $i \neq j$ uniformly distributed in $[0, 1]$. The relative error was calculated as $\frac{\|\mathbf{p}_{\text{approx}} - \mathbf{p}_{\text{true}}\|_1}{\|\mathbf{p}_{\text{true}}\|_1}$. The error bars show the 95% confidence interval based on 200 samples.

Figure 4.7b shows the accuracy of the oblivious FindFix algorithm on the real-world data set from [EGJ14] (as represented in Figure 4.6). In this 6-country network, all the countries are solvent, and in this particular case, the FindFix algorithm converges with *no* error (see Proposition 2(ii)).

Table 4.1: Equilibrium market values of the 21-country BIS data set. Values are in millions of USD.

		Exact	**Privacy-Preserving**	
Name	**Reserves**	**Market Value**	**Market Value**	**Error**
Australia	196,291	324,605	324,605	0
Belgium	35,992	110,776	110,776	0
Canada	242,895	802,207	802,207	0
Chinese Taipei	112,044	146,500	146,500	0
Finland	7,231	21,196	21,196	0
France	476,060	1,726,679	1,726,679	0
Germany	509,167	1,367,904	1,367,904	0
Greece	31,955	37,005	37,005	0
India	10,622	99,027	99,027	0
Ireland	38,679	38,483	38,483	0
Italy	247,149	529,981	529,981	0
Netherlands	160,377	376,396	376,396	0
Portugal	30,689	34,555	34,555	0
Singapore	88,725	79,733	79,733	0
South Korea	173,528	213,149	213,149	0
Spain	293,666	573,661	573,661	0
Sweden	96,427	157,070	157,070	0
Switzerland	165,399	1,002,046	1,002,046	0
Turkey	58,544	732,388	732,388	0
United Kingdom	473,852	2,010,311	2,010,311	0
United States	1,605,782	2,429,208	2,429,208	0

In this setting, we tested the stability of our algorithm by simulating stress tests, reducing each country's underlying assets by a random variable, and re-running Algorithm 4 while keeping the underlying network fixed. shows the relative error of the market-clearing vector returned by the oblivious algorithm (Algorithm 4) compared to the ground truth for increasing number of iterations, k. The bars represent a 95% confidence interval. In all tests, the inter-country debts remained fixed, and only the countries' self-holdings dropped.

In both cases, setting $k = 10$ effectively reduces the error to 0.

Four-bank example of Figure 4.4a

Figure 4.8 shows the banks' true market values as well as their approximate market values obtained from the privacy-preserving Algorithm 5, with $k = 10$. As in Figure 4.4b, $r = .1$ (each bank retains

10% self-ownership), and only Bank 2 possesses any outside assets (these are normalized to have value 1).

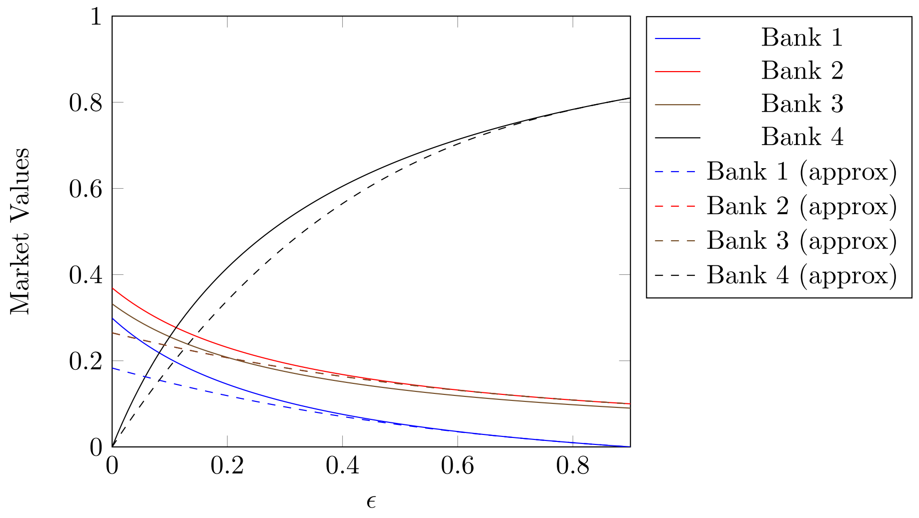


Figure 4.8: Four-bank Network of Figure 4.4a. The solid lines are the true market values, and the dotted lines represent the approximate market values calculated by the iterative algorithm (Algorithm 5).

Convergence of Algorithm 5

For a set of random 20-bank networks, Figure 4.9 shows how the error in calculating market values decreases as a function of k, the number of iterations in Algorithm 5. The networks were generated with C_{ij} for $i \neq j$ uniformly distributed in $[0, 1]$, S_{ii} uniformly distributed in $[.1, .5]$, then rescaled so that the columns of $\mathbf{C} + \mathbf{S}$ sum to 1. The relative error was calculated as $\frac{\|\mathbf{v}_{\text{approx}} - \mathbf{v}_{\text{true}}\|_1}{\|\mathbf{v}_{\text{true}}\|_1}$. The error bars show the 95% confidence interval. Setting $k = 10$ (as we did in the rest of our implementations) effectively reduces the error to 0.

4.4.3 Implementations and benchmarking

In Section 4.2.3 and Section 4.3.1, we showed how to compute network statistics in a data-oblivious manner, using only simple operations (addition, multiplication and comparisons). The three MPC frameworks we use provide *secure* (indistinguishable) executions of those basic operations. We implement Algorithms 4 and 5 in each of them.

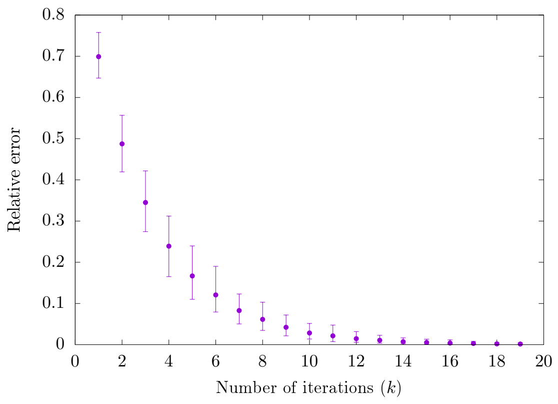


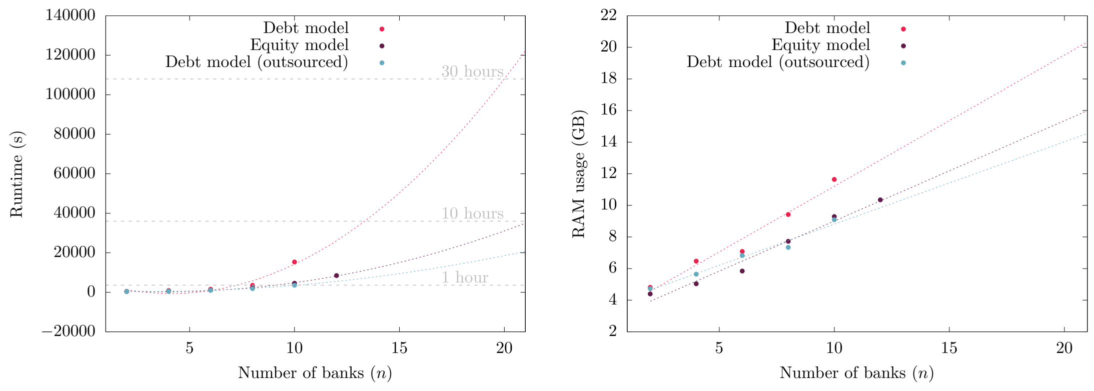
Figure 4.9: Market value error for varying numbers of iterations in Algorithm 5 for 20-bank networks

SCALE-MAMBA.

SCALE-MAMBA is an open-source software system developed at KU Leuven. It implements a custom multi-party computation algorithm based on linear-secret sharing [BDOZ11, DPSZ12, NNOB12] that is secure against a malicious adversary with an honest majority of participants. The system runs in three parts: the compilation phase consumes a user-described program to produce an executable format, the offline phase generates cryptographic material that is independent of the participants' input data Finally, the online phase executes the computation.

The offline phase is the most time- and resource-intensive. It generates and validates several forms of cryptographic material in the form of samples of correlated random variables. This correlated randomness is independent of the participants' private inputs, and can be generated and stored at any point prior to executing the protocol. These values are then used to mask or "blind" secret values during the execution of the protocol. This *pre-processing* data is discussed in more detail in Chapter 3. Since this offline phase is independent of the participants' data, it can be it can be run before the participants know their (private) inputs. For example, the offline phase could be run overnight while markets are closed, then when the markets open, the (faster) online phase could be executed using up-to-the-minute market data.

In Figure 4.10, we show total resources used *per party* to evaluate our network algorithms. These resources are for the cumulative computation, including both online and offline phases. The figure compares the debt model and the equity model with all parties participating, and the debt model computed in the “outsourced” setting described in Section 4.1.1, with three computation parties. The equity model is less resource-intensive than the debt model with all parties participating: it only requires one iteration of the iterative fixed-point algorithm (Algorithm 2), while the debt model requires n in the number of banks. The outsourced model is the most efficient; communication is a significant bottleneck, especially as the number of parties grows.



(a) Wall-clock time. **(b)** RAM usage.

Figure 4.10: Per-party resources required to run Algorithm 4 using the SCALE-MAMBA MPC framework. Given we chose to run all parties on a single machine, we benchmarked our implementation for smaller numbers of parties and extrapolated to determine resources required for a larger number of parties.

SCALE-MAMBA provides extremely strong security guarantees (security against malicious adversaries), however, the processor and memory requirements are significant. Our single machine could not handle the full 21-party BIS dataset due to memory limitations, but this should not be an issue in practice given that the computational burden would be split between the nodes in the network. We estimate that each machine would require around 18GB of RAM per party, which is highly feasible. Unfortunately, the running time is high – with an estimate of over 30 hours of computation to compute market values in a 21-bank network. Although this is, perhaps, acceptable in some situations (e.g. computing weekly stress tests), it does not seem likely to scale to much larger networks without significant algorithmic advances. The 3-party reduction affords meaningful efficiency gains (40% reduction in runtime), but is still within the same order of magnitude. The next implementation (using MPyC) *is* orders of magnitude faster, but uses a weaker security model.

MPyC.

This framework is secure against a semi-honest adversary when there is an honest majority. This is a weaker security setting than the other two frameworks and it does not require a pre-processing stage to generate cryptographic material. MPyC is implemented as a Python package and runs as interpreted language.

MPyC includes an implementation of a special-purpose secure protocol for solving linear equations of the form $\mathbf{Ax} = \mathbf{b}$ for an unknown vector $\mathbf{x}$ [BBSdV19]. The LinSol algorithm (Protocol 4 in [BBSdV19]) securely solves linear equations using integer operations only. We take advantage of this custom implementation by replacing the FindFix Algorithm 2, which uses a simple iterator to invert the input matrix, with Algorithm 6, which calls LinSol as a subroutine.

Algorithm 6 FindFix v.II

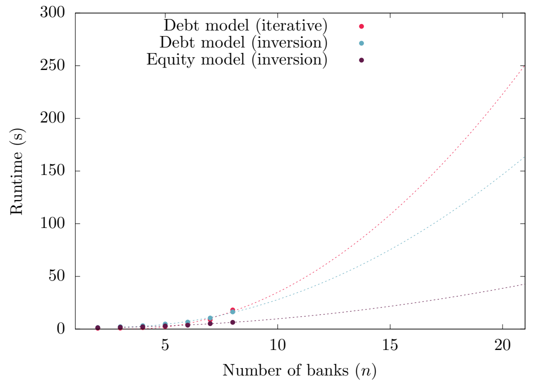
```
  Input: $\mathbf{A}, \mathbf{b}$
  $\bar{\mathbf{A}} = \mathbf{I} - \mathbf{A}$
  $(\text{adj } \mathbf{A})\mathbf{b}, \det \mathbf{A} = \textsc{LinSol}(\bar{\mathbf{A}}, \mathbf{b})$
  $\mathbf{p} = (\text{adj } \mathbf{A})\mathbf{b} / \det \mathbf{A}$

return p
```

LinSol$(\cdot)$ computes solutions directly, *without* resorting to iterative methods, but only works with *integer* inputs. All of our algorithms require real-valued inputs. To convert between real-valued (fixed-point) values and integer values requires, we follow the method proposed in [BBSdV19]. We choose a scaling factor α, left-shift[10] all values in $\mathbf{A}$ and $\mathbf{b}$ by α and truncate into an integer. If α is too small, information about the fractional part of the value is lost. If α is large, we must use more storage to store each integer, which hurts efficiency. In our implementation, we represent initial data as 64-bit fixed point numbers with a 32-bit fractional part and set parameter $\alpha = 15$.

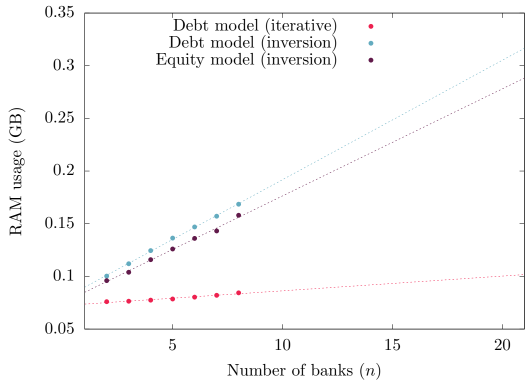
Although LinSol can operate efficiently on integer inputs, it adds an additional source of overhead cost by returning the adjugate matrix. To compute the final market values, we need to securely divide out the determinant from each value in this matrix. Division under MPC is slow and requires relatively large fixed-point integers to avoid truncation errors. In our implementation, we convert to 212-bit fixed point numbers in the equity model and 256-bit fixed point numbers in the debt model.

[10] *i.e.*, multiply by 2^α.

The data sizes we chose avoid truncation and overflow errors on the sample data with up to eight participants. We implemented both FindFix algorithms in MPyC and measured resource usage on the BIS data described in Section 4.4.1. Figure 4.11 compares the debt model using both the iterative (Algorithm 2) and inversion (Algorithm 6) methods, and the equity model using the inversion method. In the debt model, the iterative algorithm is slower with larger numbers of parties (Figure 4.11a), however, it also has a lower memory overhead (Figure 4.11b). As in SCALE-MAMBA, the equity model is faster than the debt model. Our implementation is one of the first to show the concrete efficiency gains from the special-purpose secure matrix inversion algorithm of [BBSdV19].



(a) Wall-clock time.



(b) RAM usage.

Figure 4.11: Total resources required to run Algorithms 4 and 5 using the MPyC framework.

Based on these measurements, the computation on the full 21-bank network is only expected to take about four minutes. This makes it extremely feasible for networks of this size, and much faster than the SCALE-MAMBA toolkit. However, there are several caveats to this extrapolation. First, the bitsize required to correctly represent values increases with the number of data points in the computation. If the value of a computation "overflows" its container, we get incorrect results, but increasing the bit size requires more resources (both time and RAM), especially for operations like divison and comparison.[11] The bit sizes used to benchmark up to 8 parties are too small to benchmark the full set of 21 parties, so the computation would likely take longer than 4 minutes. Second, we note that, with the given bit sizes, the matrix inversion FindFix algorithm is actually

[11]These operations require some "extra" bits in the operands or they will overflow. The amount of space required is not documented; we discovered this by trial and error. For comparisons, we converted the 64-bit fixed point operands to 128-bit integers, then converted the result back to fixed points. This was less overhead than comparing fixed points. For divison, we converted the operands to a larger fixed point: from 64 bits to 256.

slower than the iterative version for fewer than 8 parties, although the difference decreases with larger number of parties. An increase in bit sizes may add additional overhead and "reset" this advantage. To make this benchmark more consistent with a realistic setting, it may be valuable to produce an adaptive tool that selects an appropriate[12] bit size based on the input data, rather than selecting a fixed size and using it for all input.

Finally, MPyC only provides security against *passive* adversaries. By contrast, SCALE-MAMBA (and EMP-toolkit, discussed next) provide security against *malicious* adversaries, and supporting this more robust security model adds to the complexity of the underlying secure computation protocol.

EMP-toolkit.

EMP-toolkit includes implementations of several garbled-circuit-based MPC protocols as well as a general-purpose circuit compiler. To provide a point of comparison to SCALE-MAMBA, we used the multi-party protocol secure against a dishonest majority of malicious adversaries. Like SCALE-MAMBA, the algorithm runs in two phases: given a circuit describing the function, the parties run two preliminary protocols to generate cryptographic material. Then they run the secure computation protocol to evaluate the circuit.

EMP-toolkit works in the boolean circuit model, first converting the desired computation into a circuit consisting of AND, XOR and NOT gates, and then securely executing the circuit. Algorithm 4 requires matrix operations over the reals, and their corresponding boolean circuits are extremely complex, requiring hundreds of millions of gates. To illustrate the complexity of the secure computation algorithm we used EMP toolkit to generate *boolean circuits* computing Algorithms 4 and 5. The circuit sizes (as a function of the number of participants n) are in Figure 4.12. In contrast to the experimental results from SCALE-MAMBA and MPyC, the equity model produces much larger circuits. This is likely due to a discrepancy in bit sizes: the equity model uses 50-bit fixed point numbers (with a 16-bit fractional part), but the debt model uses only 32 bits (with an 8-bit fractional part).

[12] *i.e.*, the smallest sizes that don't overflow

Parties	Time (s)	RAM (gigabytes)
2	0.65	0.15 - 0.19
4	14.58	1.97 - 3.92

Table 4.2: Resource requirements for EMP-toolkit to run Algorithm 5. In our experiments, one party required a larger amount of RAM, while the rest used only the smaller amount.

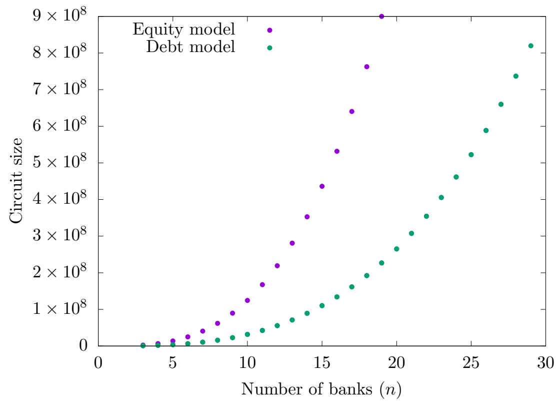


Figure 4.12: Circuit sizes for computing network statistics in the debt (Algorithm 4) and equity (Algorithm 5) models. The plot shows the number of boolean gates required to securely execute the algorithm with $k = 10$. These circuits are large, requiring over 800 million gates when $n = 30$.

The circuit size gives an indication of the complexity of the computation that is independent from the specific computing hardware and networking architecture. Unfortunately, we were unable, for the most part, to *execute* these circuits using EMP toolkit. Table 4.2 shows the runtime for Algorithm 5 with a small number of parties.

Previous work [KsS12] showed how to execute a garbled-circuit protocol in the malicious security model extremely efficiently. They report executing a circuit with 5.9 billion gates in 14,700 seconds. Extrapolating to our scenario (where our circuits have only 800 million gates), the computation time on their setup would only be around 20 seconds.

Summary.

Our experiments show that although the calculations are computationally intensive, they are *feasible* for small networks ($n \sim 20$) on commodity hardware. If performance were a consideration, real-world deployments could dramatically increase the run-times with more powerful hardware and a more highly optimized software implementation.

It is also interesting to note difference in runtimes between SCALE-MAMBA and MPyC. The two frameworks use extremely different MPC back-ends so it is hard to pinpoint a single cause of the performance differences. Nevertheless, it seems likely that providing security against malicious adversaries (as in SCALE-MAMBA) is a primary cause of its slower performance. If this is the case, it is an interesting question whether practitioners (e.g. banks or regulators) feel the need to provide security against malicious adversaries, or whether security against semi-honest adversaries is sufficient.

4.5 Discussion, limitations and conclusion

4.5.1 Feasibility and future work

Although prior work has suggested the use of MPC for privacy-preserving financial computations [AKL12, CK19], these works only considered generic protocols (e.g. sums, variances, moments, linear inference) rather than more complex protocols (e.g. market-values) from the literature. The one prior work that develops MPC for financial networks [NPH14] provides weaker security guarantees than ours, and does not analyze the convergence of their algorithms. This work is the first to provide strong security and convergence guarantees and to provide benchmarks on real-world data. To demonstrate feasibility, we have implemented two different market-value computations, one in the debt model, and one in the equity model, using three different secure multi-party computation frameworks.

Our implementations show that these secure computations are indeed feasible for realistic networks. In terms of efficiency, we claim that existing tools are sufficient for use in some applications, including computing market values for small networks. We estimate SCALE-MAMBA requires less than 20GB of RAM and around 30 hours per party to compute market values of a network with 21 participants. This is a relatively low cost for a small group of banks who wish to run monthly risk assessments amongst themselves. There are a variety of known optimizations that could improve our results as well, such as porting Algorithm 6 to SCALE-MAMBA or EMP-toolkit and carefully tuning software parameters and bit sizes to optimize the computation. At a higher level, using special-purpose MPC protocols designed for highly efficient operations over real-valued matrices could dramatically improve runtimes.

The software tools we use here were developed as academic projects and are not, generally speaking, production quality code. Despite this, they demonstrate the *feasibility* of this approach. In a real application setting, a coalition of financial institutions could fund the development an open-source, production-quality implementation. Coupling this with a moderate investment in hardware, we believe that these protocols could easily see significant performance improvements over our prototype implementations. We believe this is an investment worth making, as the development of such a tool would enable widespread adoption of secure computation.

Our work shows that key calculations in financial networks can be calculated in a privacy-preserving manner. Our work also highlights the key variables that affect the performance. Thus, any coalition of institutions considering this type of system needs to ask (1) How many institutions want to participate in the calculation (using secret-sharing, the number of institutions providing data can be larger than the number of institutions actively participating in the computation, see Section 4.1.1), (2) Do the institutions need security against malicious adversaries, or is semi-honest security sufficient? (3) How often do the calculations need to be run (e.g. is a running time of 6 hours acceptable)?

4.5.2 Limitations

Second-order inference: In this work, we focus on the question of securely computing market values, and financial stress-tests, without a trusted third party, and without revealing the stakeholders' underlying data. Like extant literature in this space, we do not address the scenario that the *result* of the calculation (e.g. the market values of the institutions, the number of institutions that fail, or the total financial shortfall) could reveal sensitive information about the underlying institutions. In principle, one could use secure computation to implement a differentially private stress-test to achieve both privacy against an adversary monitoring network traffic, and an adversary performing inference on the result [NPH14].

Garbage-in-garbage-out problem: Our algorithms do not ensure that the participants behave *truthfully.* In fact, MPC protocols in general can only enforce "truthful" behavior if such behavior can formally (mathematically) defined. For example, our protocol could be easily extended to enforce simple consistency checks (e.g., in the debt model if i reports a debt to j, then j should report the same debt from i), but without a mathematical definition of "truth" for each participants private inputs, the protocol cannot enforce that the participants behave truthfully. Of course, if participants provide incorrect inputs, the calculated market values will also be incorrect.

Scalability: Our implementations show that these types of computations can easily scale to dozens (or even hundreds) of participants. Unfortunately, the communication costs of the underlying secure computation algorithms scale *quadratically* with the number of participants, so it is unlikely that this architecture, in its current form, can scale to *thousands* of participants. To address settings with more participants, it is common to change the network architecture, separating the number of data owners (*i.e.*, the number of banks) from the number of computation parties (e.g. a subset of larger banks, or outside agencies). This allows us to increase the number of banks without increasing the number of computational parties. Figure 4.10 shows that running Algorithm 4 with three parties instead of 21 reduces the runtime by about 40%. Unfortunately, this reduces security in the sense that all privacy is lost if *all* the computation parties collude (even if the majority of the *input* parties do not collude).

CHAPTER 5

zkChannels: Fast, Unlinkable Payments for Any Blockchain using 2PC

Since the introduction of Bitcoin [Nak19], blockchain-based digital currencies have flourished and now process billions of dollars worth of transactions daily [Coi]. However, cryptocurrencies suffer from significant privacy, scalability, and latency limitations that reduce usability. In particular, (1) most blockchains do not provide robust anonymity properties, and (2) processing transactions is quite slow and often expensive.

Each of these problems has been addressed in isolation. Some cryptocurrencies integrate cryptographic anonymity features to prevent observers from tracking the flow of transactions, such as shielded transactions in Zcash [SCG$^+$14] and ring signatures in Monero [Mon]. Various off-chain solutions, including *payment channels*, have been developed to make processing transactions faster and cheaper [PD16]. These protocols work by moving the bulk of transactions off chain and relying on the chain to resolve disputes. In its simplest form, a payment channel allows two parties to escrow funds in an on-chain account and then make payments to each other off chain. An on-chain arbitration mechanism allows participants to close on their most recent balances. The most widely used of these off-chain solutions is the Lightning Network [PD16], which combines many payment channels into a larger *Payment Channel Network (PCN)* that allows two parties who do not have an established channel together to still engage in a mutual transaction.

Payment channels and PCNs are an active research area [SVR$^+$20], with many important, unresolved usability and privacy limitations. Most solutions involve channels that provide no cryptographically-enforced privacy guarantees to the participants, either with respect to one another, other parties in the larger PCN, or observers of the underlying blockchain.

Privacy-preserving payment channels, which Green and Miers introduce in the Bolt protocol [GM17], are an important step towards simultaneously improving scalability and achieving robust privacy. These channels enable anonymous, unlinkable payments between a *customer* and a *merchant*, a natural use case for point-to-point payments. In this context, anonymity and unlinkability mean that the merchant has no advantage in guessing which customer is making a given payment, or even that the source of any two payments is the same customer, unless the included purpose and amount of the payment reveals identifying information. This means the merchant cannot link a payment to a customer, nor can the merchant construct payment histories. As is standard in payment channels, the participants set up the channel by escrowing funds on a blockchain; after this initial, potentially non-anonymous phase, the customer can initiate payments to the merchant anonymously in an unlinkable fashion. Ideally, either party can initiate a channel closure, starting the process of allocating the escrowed funds according to the latest balances in the channel. Channel closures should not leak anything beyond the closing balances and should guarantee that an honest party cannot lose money.

In this work, we present, implement, and prove secure a novel privacy-preserving payment channels protocol, which we name *zkChannels*. zkChannels is a generalization of Bolt designed to integrate easily with existing cryptocurrencies, requiring that the underlying blockchain support only basic features that are already nearly ubiquitous. The heart of our protocol is a secure two-party computation (2PC) [Yao86, LP09a]. While secure 2PC protocols have been known for decades, they have only recently become efficient enough for use as a building block without compromising practicality [WRK17a]. zkChannels also introduces new techniques that improve the security and usability of privacy-preserving preserving payment channels that cannot be realized by 2PC alone. The resulting protocol is carefully designed to provide strong privacy and security, despite the impossibility of fairness for two-party protocols [Cle86].

The zkChannels protocol has several desirable properties compared to previous work: (1) Customer payments are fully unlinkable, both with respect to each other and to the channel, even in the presence of merchant aborts. A key limitation of Bolt is the failure to provide strong privacy against a merchant that aborts during a payment, instead relying heavily on funding the channel with a fully anonymous cryptocurrency, a feature that is not widely available. In zkChannels, the customer always has an unlinkable way to close the channel down. (2) Our protocol does not require the integration of any channel-aware logic into the blockchain or the deployment of new opcodes or primitives, instead relying on 2PC to support our on-chain arbitration process with existing blockchain functionalities. Bolt, in contrast, tightly binds its off-chain and on-chain components using shared cryptographic primitives, meaning the underlying blockchain must support specific cryptographic operations (none currently do). Their approach requires the introduction of new op-codes for blind signature verification and additional, channel-aware logic to enforce correct opening and closure. While a few blockchain developers have signaled a willingness to do this [ZIP, Tez], there appears to be no interest from the most active cryptocurrencies, *e.g.*, Bitcoin and Ethereum. (3) We provide a complete specification for the escrow and arbitration mechanisms of zkChannels, including how to guarantee the merchant the ability to close channels. In Bolt, escrow and closing mechanisms are left mostly unspecified: detailing, implementing, and auditing the full scope of changes that must be made to the underlying blockchain to support their approach is a significant engineering and community consensus effort [Zca]. Additionally, no solution for guaranteed merchant closures is provided in Bolt, which significantly impacts the incentive a merchant has to participate in a payment channel, as funds are locked in escrow until a customer chooses to close. (4) We specify how to integrate the purpose of a payment into zkChannels, answering the question of when a merchant can reasonably consider a payment complete, which Green and Miers [GM17] leave open.

Contributions. Our main contributions are as follows:

– We formally define zkChannels, the first fully privacy-preserving, unlinkable payment channel protocol suitable for deployment on Bitcoin and other cryptocurrencies without requiring modifications. We give a complete specification of our protocol, including how to integrate zkChannels with a payment network and codify our protocol's privacy guarantees and improvements over Bolt [GM17]. In particular, zkChannels provides the customer with a closing method that cannot be linked to their payments on the channel, even in the case of a malicious merchant abort, and uses a simple trick to provide the merchant the ability to force channel closure, even without knowing the current balance allocation. We also show how to integrate zkChannels payments with respect to the payment's purpose, allowing the merchant to safely provide the requested service, refund, or item.

– We define and prove zkChannels security in the context of an abstract, currency-controlling functionality we call an *arbiter*. This captures that zkChannels can work with many different types of payment networks. We provide a summary of our security proof in the standalone model. We emphasize that most UTXO cryptocurrencies[1] satisfy the requirements needed to realize our arbiter functionality, meaning that zkChannels can be instantiated on many blockchains. Our abstraction of the arbiter may be of independent interest and can serve as a useful tool for defining protocol extensions, such as state channels and modified closing approaches.

– We provide a complete, tested, end-to-end implementation of zkChannels for Bitcoin. This includes an efficient 2PC for both semihonest [Yao86] and malicious [WRK17a] security. Our circuit builds and signs actual Bitcoin transactions bit-by-bit. Our circuit has 10 million gates, including 2.5 million AND gates. Evaluating the semihonest version of this circuit as part of our payment protocol takes $\approx$ 355ms; the malicious version takes $\approx$ 10s. We believe this is one of the first substantial uses of generic 2PC frameworks with the potential to process real, sensitive information.

5.1 zkChannels overview and intuition

The zkChannels protocol allows a *customer* ($\mathcal{C}$) and a *merchant* ($\mathcal{M}$) to open a bidirectional payment channel with respect to a payment network capable of arbitration, which we refer to as the *arbiter* ($\mathcal{J}$). As is standard for payment channels, both parties may close the channel at any time and an honest party is guaranteed to be paid at least the balance they are owed. Our protocol, however, is asymmetric and private: the customer initiates all payments, each of which is anonymous and cannot be linked to any other payments.

Privacy preserving payment channels are composed of two primary parts, an off-network mechanism to allow payments and a set of on-network procedures that allow for escrow and arbitration during closure. To design a protocol that works for virtually any payment network, we design a payment subprotocol that integrates with a simple arbiter.[2]

[1] "Unspent transaction output" (UTXO) refers to the method of tracking accounts as sets of unspent coins.

[2] It may be possible to instantiate a more efficient payment subprotocol if the payment network supports advanced features, like account-based cryptocurrencies with smart contracts. In this work, we focus on the harder case, with a simpler UTXO Bitcoin-like network in mind.

At a high level, the arbiter allows funds to be kept in *accounts*. Accounts have *encumbrances* that specify how funds may be disbursed. Accounts may be created, modified, or closed through *transactions*. In the case of a cryptocurrency network, transactions that are *accepted* and/or *confirmed* are public; participants are responsible for monitoring the ledger for relevant transactions. Theoretically, an arbiter without a public ledger may be used, but requires direct communication between the arbiter and channel participants and is not the focus of this paper. The arbiter $\mathcal{J}$ is responsible for receiving, validating, and processing transactions. If the arbiter is a payment network such as a cryptocurrency, all zkChannels transactions must be well-formed according to the given payment network's rules; the instantiation must give an exact specification.

We formally define an arbiter in Section 5.2.2 and zkChannels in Section 5.3. In the next few subsections, we give the key intuitions behind our protocol.

5.1.1 Privacy properties

The merchant is at most pseudonymous and remains identifiable across all channels. The customer $\mathcal{C}$ is afforded privacy for individual payments, but the zkChannels protocol is agnostic as to whether or not the merchant learns $\mathcal{C}$'s identity during channel opening and, by consequence, closure. This does not affect $\mathcal{C}$'s ability to make payments anonymously *as long as they have an open payment channel with sufficient balance.* The anonymity set for a payment is the set of all customers with whom the given merchant has an open channel. We do not require the use of an anonymous cryptocurrency to achieve payment privacy.

The unlinkability property ensures that a merchant will not obtain any cryptographic evidence from the protocol that can tie a customer to a specific payment (or payments to each other). However, it does not eliminate inference or timing attacks by the merchant. For example, a merchant can track payment amounts and, upon channel closure, search for a subset of payments that add up to the final channel balances.[3]

The protocol includes optional auxiliary information as input and output to each payment. This can be used to facilitate the provision of services, *e.g.*, to provide the customer with a randomizable token that can be redeemed for a service. The protocol does not consider the extent to which auxiliary information can leak additional information.

[3]The subset-sum problem is NP-complete, but is likely feasible to compute in this setting.

The zkChannels protocol does not address networking privacy. Interactive subprotocols between the parties take place over *sessions*, in which the merchant is authenticated, but the customer is not. In practice, parties concerned about traffic analysis by third parties or the merchant can use an anonymizing networking tool such as Tor.

Use cases. zkChannels is suited to any use case with a traditional customer-merchant relationship. A simple example has a neighborhood coffee shop as the merchant. In this setting, zkChannels act as a pre-paid gift card. Many customers visit regularly for short periods of time; their payments are processed efficiently (*e.g.* faster than drinking an espresso) and do not leave the shop with a record of their visiting habits. The merchant saves money on individual transaction fees charged by a credit card processor or a direct on-chain cryptocurrency transaction.

A similar gift-card approach applies in many scenarios, such as large online retailers, VPN providers, or ISPs. These retailers can maintain their regulatory ("know your customer") requirements by verifying customer identity before establishing a channel, but zkChannels improves customer privacy across individual transactions.

Comparing to private cryptocurrencies. As discussed, zkChannels can layer privacy guarantees on top of a wide variety of existing, non-private payment networks. We compare the privacy and efficiency guarantees of zkChannels to those of cryptocurrencies that provide privacy by design.

Monero [Mon, NMRL16] provides unlinkability for all payments: observers cannot link payments to the sender or receiver, the receiver cannot identify the sender of a payment. The anonymity set includes all participants in the Monero network. ZCash [SCG$^+$14] offers two types of addresses: shielded and transparent. Transactions between shielded addresses maintain sender and receiver privacy, with the anonymity set as the set of shielded addresses. Monero and ZCash shielded transactions both hide the transaction amount. zkChannels provides sender unlinkability by default for all payments, although a customer can optionally reveal their identity via the auxiliary payment information. The anonymity set is the set of customers that maintain an open channel with the receiving merchant.

A simple efficiency measure is the minimum time for payment confirmation. The Kraken cryptocurrency exchange publishes the minimum number of confirmations required for a valid deposit [Kra21]. For Monero, this takes roughly 30 minutes; for ZCash, close to 60 minutes. This is comparable to non-private cryptocurrencies, such as Bitcoin's 40 minutes, but much slower than the benchmarks in Section 5.5: zkChannels takes less than 2 minutes to process a payment.

5.1.2 Channel opening

The goal of channel opening is for the customer and merchant to agree on initial channel parameters, safely escrow funds on network, and provide the customer the ability to start making unlinkable payments. To avoid the need for a special-purpose 2PC for opening, the parties pick channel parameters in the clear, including the customer's *initial state*, denoted by s_0. Since customer states in our protocol contain secrets that *should not be revealed* before making a payment, we have the customer make an initial zero payment, which gives the customer a fresh, unlinkable version of s_0, in order to bootstrap the channel.

Channel parameters consist of auxiliary information needed to form network transactions and the state s_0, which includes the starting customer balance, $B_0^{\mathcal{C}}$, and merchant balance, $B_0^{\mathcal{M}}$, a channel identifier, and two additional, randomly generated values, which we describe later. In order to escrow channel funds safely, the two parties first collaborate to create special transactions that allow each party to close; this prevents a malicious party from "locking" the other's funds indefinitely. With these transactions in hand, the parties post the escrow transaction, `escrow`, on network. Once the escrow transaction is confirmed, the merchant sends the customer a *payment tag* on the initial state, denoted by pt_0. This tag is a merchant authorization on the state s_0 under a well-known, longterm public key, *e.g.*, a signature, and can be used to make a payment.

5.1.3 Channel payments

From the customer's perspective, the payment channel consists of a sequence of *states*, denoted by $\{s_i\}_{i \in \mathbb{N}}$, each of which has a corresponding payment tag, pt_i, and a *closing authorization* that allows the customer to close on the balances embedded in a given state when presented to the arbiter $\mathcal{J}$.

To make a payment of (possibly negative) value ϵ, the customer $\mathcal{C}$ needs to privately "spend" a pair (s_i, pt_i), say representing a balance of $(B_i^{\mathcal{C}}, B_i^{\mathcal{M}})$, to get a new state and tag pair, (s_{i+1}, pt_{i+1}), that embeds and authorizes a balance of $(B_i^{\mathcal{C}} - \epsilon, B_i^{\mathcal{M}} + \epsilon)$. For privacy, the customer requires the pair (s_{i+1}, pt_{i+1}) be:

(1) not linkable to *any* previous payment session;[4] and

(2) usable in an unlinkable manner either to make another payment or to close the channel.

To preserve functionality, the merchant $\mathcal{M}$ requires that the following are true:

(1) The new state, s_{i+1}, is properly formed from the previous state and the payment amount.

(2) The balances satisfy $B_i^{\mathcal{C}} - \epsilon \geq 0$ and $B_i^{\mathcal{M}} + \epsilon \geq 0$.

(3) The customer's new closing authorization is with respect to the correctly updated state.

(4) The customer has never made a payment on state s_i before, preventing off-network "double spends".

(5) If the customer tries to close on the old state, s_i, then the merchant can use the on-network arbitration mechanism to dispute the dishonest transaction.

zkChannels uses a maliciously secure, generic 2PC that takes as input a pair (s_i, pt_i) and the merchant's secret key material, and computes all the outputs the customer needs at the end of the pay session: a payment tag pt_{i+1} on state s_{i+1} and a new set of closing authorization signatures. Producing the closing authorization signature(s) involves non-black box use of the arbiter $\mathcal{J}$'s authorization mechanism, *i.e.*, , the 2PC must run a signing algorithm supported by $\mathcal{J}$. This 2PC also performs correctness checks that ensure that the channel has sufficient funds for making the payment. This includes checking that the existing tag pt_i is valid and corresponds to the input state s_i, and that the new state s_{i+1} is correctly formed and embeds the correct balances.

Although a powerful primitive, 2PC alone cannot meet all the goals outlined above. Specifically, 2PC cannot prevent a malicious customer from reusing old state and cannot provide an unlinkable arbitration mechanism. To address these problems, we include additional secrets in each state: a *nonce*, denoted by n, and a *revocation pair*, denoted by (rl, rs).

Preventing reuse via nonces. To ensure a "spent" state cannot be reused, we include a nonce that is sent to the merchant in the clear at the beginning of the payment subprotocol. The merchant keeps a database of previously seen nonces and continues the pay session only if the revealed nonce is fresh. The 2PC checks that the input state does indeed contain the revealed nonce.

[4]zkChannels only guarantees that the customer's payments are unlinkable if the customer acts honestly. If the customer deviates from the protocol, it may be possible to link their payments.

Invalidating old states via revocation pairs. To disincentive the customer from closing the channel on an old state, we use revocation pairs to embed a punishment mechanism into the closing transactions. After making a payment, the customer invalidates the closing transactions for a given state s with an embedded revocation pair (rl, rs) by sending this pair to the merchant. If the customer later closes on s, the merchant can use the revocation secret rs to claim all the funds in the channel. As such, we require that it be infeasible to extract rs from the revocation lock rl. As with nonces, the 2PC checks that the revealed value rl is exactly the one stored in its input state. We formalize revocation pairs in Section 5.2 and instantiate them as hash locks for Bitcoin.

It may be tempting to use the same mechanism for preventing reuse and invalidating old state, as in Bolt [GM17]. However, this allows a malicious merchant to abort a payment session after learning the nonce and then wait for this value to appear in a closing transaction on the payment network. This means a customer must either forfeit their channel funds or link their on-network identity to this aborted payment session. When we separate these mechanisms, the nonce never appears on network and an honest customer never posts a previously revealed revocation lock to the network.

Composing the pay protocol. The payment protocol is an *exchange* of information: the customer receives a new state pair, (s_{i+1}, pt_{i+1}) and the merchant receives the nonce n_i and the pair (rl_i, rs_i) from the spent state s_i. This exchange ideally happens atomically. If only the customer $\mathcal{C}$ receives output, they can reuse s_i at will; if only the merchant receives output, they obtain a revocation secret for *all* closing transactions $\mathcal{C}$ can create. However, two-party protocols cannot achieve fairness [Cle86], so we cannot simply have the 2PC send the outputs to the customer and we must compromise. We focus on privacy and take a similar intuitive approach to Bolt [GM17]: the customer risks the value of the payment when they request a service, but can always close the channel unlinkably.

To accomplish this, we ensure that each participant receives their new information in an order that protects their interests: the customer will not reveal (rl_i, rs_i) until they have a way to unlinkably close on the new state, *i.e.*, , the closing authorization for s_{i+1}, and the merchant will not issue the tag pt_{i+1} until they have received (rl_i, rs_i). As such, the 2PC computes masked versions of the outputs, and then the parties do a step-by-step release of information during an *unmask* phase. If either player aborts during this phase, unlinkable closure is still possible. For a detailed description of the unmask phase, see Section 5.3.3.

5.1.4 Arbiter mechanics: escrow and closing

The zkChannels protocol aims to operate within a generic payment network, where each party has an account containing funds and can authorize spending those funds via *transactions*. However, the design space of blockchain-based cryptocurrencies is large and varied. Some payment networks can embed complex logic into transactions to define exactly when and how a payment should occur; others have far more limited expressive capabilities. This work focuses on UTXO networks, which tend to be less expressive.

To allow for these varied abilities, the zkChannels protocol interacts with a generic *arbiter* and uses a sequence of transactions to open and correctly close channels. This section defines the required capabilities the underlying network must satisfy in order to serve as a zkChannel arbiter, describes the transactions used for the escrow and close protocols, and briefly discusses other possible instantiations of the arbiter beyond the UTXO model.

Requirements for a UTXO arbiter. This work uses a generic arbiter $\mathcal{J}$ for a UTXO network. This design is motivated by Bitcoin, but is generic enough to apply to other, similar cryptocurrency networks.

A major design constraint is that such an arbiter is *zkChannel-agnostic*: transactions specified using $\mathcal{J}$ do not encode any zkChannel-specific logic and the arbiter cannot explicitly or implicitly identify these transactions as part of a zkChannel instance. Any party that can satisfy the encumbrances of the transaction can claim the funds. Thus, we define the three main requirements for a UTXO arbiter as types of encumbrances that it must support.

The first is signature-based verification. The transaction embeds a public key and the arbiter only releases funds to a party that can provide a signature that validates under the public key. zkChannels requires a 2-of-2 threshold signature scheme.[5] The second is support for revocation-based encumbrances. The transaction embeds a revocation lock rl; funds are released to a party that can provide a matching revocation secret rs. Finally, the arbiter must allow relative timelock encumbrances. These release funds to a party only after a certain amount of time (relative to the transaction's confirmation) has passed. The transaction embeds a time δ and a set of additional encumbrances.

[5] For Bitcoin, we instantiate this using 2-of-2 multi-sigs.

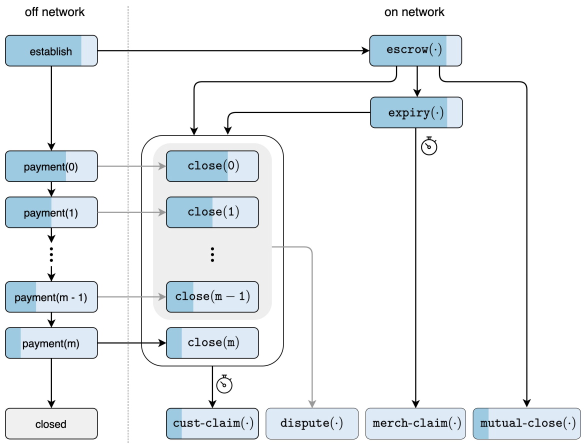


Figure 5.1: High level snapshot of the zkChannels lifecycle, with off-network subprotocols (left) and network transactions (right). Off network, the parties establish the channel and post `escrow` on network. A customer can initiate a sequence of payments off network. Channel closure can be either merchant-initiated, via `expiry`, or customer-initiated. The customer can post a transaction closing on any post-payment intermediate state, but should post `close`(n), which reflects the state after the final, n^{th} payment. If they do not, a disputing merchant can claim all channel funds with `dispute`. Dark and light blue shading in each node shows the customer and merchant balances at each step, respectively; this example shows only positive payments for clarity.

The arbiter must be able to combine and evaluate these encumbrances logically (that is, using *and* and *or* relations). Any arbiter that can authorize a transaction based on a combination of these elements is compatible with zkChannels. This includes many popular cryptocurrencies such as Litecoin, Bitcoin Cash, and Dogecoin.

Integration with zkChannels. The zkChannels escrow and closing sequences are captured in Figure 5.1. Channel participants collaborate to create and fund the channel with `escrow`, producing and authorizing initial closing transactions for both parties in the process. For each payment, the parties use 2PC to create and authorize updated closing transactions for the customer. The initial and subsequent closing transactions are authorized with signatures from both parties.

There are two primary types of closing sequences supported. In a *unilateral customer* flow, the customer first posts a customer closing transaction, `close`, which pays out the merchant balance and timelocks the claimed customer balance. This allows the merchant a chance to post `dispute` and claim this balance (using a revocation secret) if the posted state is outdated. In a *unilateral merchant* flow, the merchant first posts `expiry`, which puts a timelock on all the funds and allows the merchant to claim them all with `merch-claim` once the timelock expires. This time delay allows the customer an opportunity to post their own closing transaction with up-to-date balances, which is still subject to the same dispute mechanism as in the unilateral customer close flow. zkChannels also supports an efficient mutual close procedure when the customer and merchant are willing to cooperate.

Expanding to non-UTXO arbiters. This paper focuses primarily on zkChannels-agnostic, UTXO networks, but the zkChannels protocol is also compatible with more expressive account-based networks, such as Ethereum or Tezos. Such cryptocurrencies allow for a relatively natural realization of an account as a defined object that is acted upon by subsequent transactions, *i.e.*, , a smart contract. This design opens the possibility of *zkChannels-aware* closing methods, where the arbiter performs zkChannels-specific logic.

For example, Tezos accounts have memory that can be updated. A zkChannels contract can contain the channel balances, status (whether the channel is initialized, funded and open, or closing), and the revocation lock. The account defines *entrypoints*, which can update the account memory and pay out funds. Encumbrances can still depend on signatures, timelocks, and revocation pairs, as above, but the account itself can embed additional logic (such as updating balances and revocation locks) that simplifies the off-chain logic. For example, a Tezos account can verify a randomized Pointcheval-Sanders (PS) signature, which simplifies the pay protocol by replacing the "blind" signature under 2PC with a PS signature.

In practice, the capabilities of the arbiter span a spectrum, and explicit recognition of this fact is helpful in deciding on the specifics of an instantiation. It is possible to make efficiency improvements by shifting the implementation toward zkChannels-aware methods where possible. In future work, we aim to describe a generic, channel-aware arbiter functionality that can be instantiated by a smart contract, a series of UTXO transactions with additional non-arbiter logic, or some middle ground.

Another degree of freedom in our design is the *closing model*: at the extremes, we have a choice between *punishment*, in which a dishonest customer that posts an old view of the channel state to the network can lose all their funds to the merchant, and *reconciliation*, in which closing allows for on-network recovery of the most recent channel state. We focus on the former in this paper, as this is common in practice [PD16], but we observe that achieving a combined punishment and/or reconciliation model is an interesting question for future work.

5.2 Preliminaries

5.2.1 Cryptographic primitives

We establish notation and give definitions for the building blocks used in our protocol. We refer the reader to [KL14] for details on standard cryptographic primitives.

Notation. Let $y_1, \ldots, y_m = \mathsf{f}(x_1, \ldots, x_n; r_1, \ldots, r_s)$ denote an algorithm f taking as inputs the variables $(x_1, \ldots, x_n)$ and random coins $(r_1, \ldots, r_s)$ and generating as outputs the variables $(y_1, \ldots, y_m)$. When clear from context, we may omit some parameters, such as the public values or random coins.

Message Authentication Code (MAC). A MAC scheme Π_{MAC} is a tuple $(\mathsf{KeyGen}, \mathsf{Mac}, \mathsf{Verify})$, with the following interfaces:

– $\mathsf{KeyGen}(1^\lambda)$ outputs a key K with security parameter 1^λ;
– $\mathsf{Mac}(K, m)$ outputs a tag t on a message m; and
– $\mathsf{Verify}(K, m, t)$ outputs a boolean $b \in \{\mathsf{false}, \mathsf{true}\}$.

We assume standard existential unforgeability under adaptive chosen message attacks.

Digital Signatures. A digital signature scheme Π_{Sig} is a tuple $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ with the following interfaces:

– $\mathsf{Setup}(1^\lambda)$ outputs public parameters pp;
– $\mathsf{KeyGen}(\mathsf{pp}; r)$ outputs a keypair (sk, pk);
– $\mathsf{Sign}(sk, m; r)$ outputs a signature σ on a message m; and
– $\mathsf{Verify}(pk, m, \sigma)$ outputs a boolean $b \in \{\mathsf{false}, \mathsf{true}\}$.

We assume existential unforgeability under adaptive chosen message attacks.

Commitment Schemes. A commitment scheme Π_{Com} is a tuple $(\mathsf{Setup}, \mathsf{Com})$ with the following interfaces:

– $\mathsf{Setup}(1^\lambda)$ outputs public parameters pp; and
– $\mathsf{Com}(m; r)$ outputs a commitment $\mathsf{com}(m; r)$ on a message m and random coins r.

We assume Π_{Com} is hiding and binding.

Revocation Locks. We define a *revocation lock scheme*, $\Pi_{\mathsf{Rlock}} = (\mathsf{Gen}, \mathsf{Verify})$, as follows:

- $\mathsf{Gen}(1^\lambda)$ is a probabilistic algorithm that takes a security parameter 1^λ as input and outputs a *revocation lock pair* (rl, rs); and
- $\mathsf{Verify}(rl, rs)$ takes in a revocation lock rl and a revocation secret rs and outputs a boolean $b \in \{\mathsf{false}, \mathsf{true}\}$.

We require that these functions satisfy the following properties:

Correctness: For every λ and every pair $(rl, rs \leftarrow \mathsf{Gen}(1^\lambda)$, we have

$$\mathsf{Verify}(rl, rs) = \mathsf{true}.$$

Security: For all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr[\mathsf{Verify}(rl, rs) = \mathsf{true} \mid (rl, \cdot) \leftarrow \mathsf{Gen}(1^\lambda), rs \leftarrow \mathcal{A}(rl)] < \mathsf{negl}(\lambda).$$

Revocation locks are a formalization of the *hash lock* construction, discussed next.

Hash-based Constructions. In our implementation, we instantiate commitments and revocation locks with hash based primitives, relying on the collision resistance of some standard cryptographic hash function H. To commit to a message m, we sample randomness $r \xleftarrow{\$} \{0,1\}^\lambda$ and output the commitment $\mathsf{H}(r \| m)$. We use this hash-based commitment scheme to create revocation locks by setting $\Pi_{\mathsf{Rlock}}.\mathsf{Gen}(r) = (\mathsf{com}(r; \perp), r)$. This construction is a standard *hash lock* commonly used in blockchain applications.

5.2.2 Arbiter abstraction.

We describe transactions processed by the arbiter $\mathcal{J}$ as a pair (κ, T), where κ is a tuple of *instructions* specifying one or more accounts and the new encumbrances, and T is an *authorization set*: accompanying evidence that shows the terms of the instructions specifying disbursement have been met.

In a UTXO network, an account is realized through a sequence of ephemeral objects, called *addresses*, that are acted upon by a corresponding sequence of transactions. For this reason, we sometimes abuse terminology and refer to an account by the most recent transaction that places encumbrances on the funds in question. We say an account is *current* if we are referencing the most recent set of encumbrances on the given funds. That is, we define accounts via a one-to-one mapping with (funds, encumbrance) pairs; transactions can split, combine, create, or remove accounts.

The set of *well-formed* transactions depends on the formal language of the chosen arbiter, but must be precise in its specification of which funds are being acted upon and how. In a UTXO network, transactions specify *sending* and *receiving* accounts, together with the associated encumbrances on those receiving accounts, and provide evidence meant to satisfy the encumbrances on the sending accounts. In order to be *legitimate*, transactions must be well-formed, the instructions must specify a current account(s), and the authorization set and transaction context must be deemed satisfactory by $\mathcal{J}$. That is, the transaction must actually satisfy any encumbrances placed on the specified account(s), which may include restrictions on the type of allowable encumbrances for receiving accounts.[6] We assume transactions can be verified as legitimate by any entity, in keeping with the public nature of blockchain ledgers.

Key assumptions and notation

The arbiter $\mathcal{J}$ must implement a threshold signature scheme Π_{Sig} and a revocation lock scheme Π_{Hlock}.[7] Further, transactions must have canonical identifiers, called *transaction IDs*, defined by a function $f_{\mathsf{ID}} \colon \mathtt{tx} \mapsto \mathtt{txid}$. That is, any two transactions that have the same *effect* (*i.e.*, , instructions) should have the same transaction ID, independent of any possible differences in the transactions' authorization sets.[8] When we wish to make this explicit, we write $\mathcal{J}^{(\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Hlock}}, f_{\mathsf{ID}})}$.

As mentioned, the arbiter must also have a method of specifying timeouts. That is, $\mathcal{J}$ allows for encumbrances of the form "After time δ has passed, a transaction may specify the transfer of these funds provided a sufficient authorization set."

We name and parametrize transactions with zkChannel-specific vocabulary and variables (these names and parameters are not provided directly to $\mathcal{J}$). Notationally, we describe a given transaction $\mathtt{tx}$ as a pair of instructions κ and an authorization set $T = \{t_0, \dots, t_\ell\}$. We write $\mathtt{tx}(\mathit{params})$ to refer only to the instructions κ and $\mathtt{tx}(\mathit{params}; t_0, \dots, t_\ell)$ to denote the complete, authorized transaction.

[6]More formally, transactions must adhere to the semantics of the arbiter's language and the formulas used to define encumbrances must be valid given the current state of the network.

[7]Technically, a threshold signature scheme suffices for revocation locks. On a UTXO network, we can use multi-sigs instead of threshold signatures.

[8]Canonical transaction IDs allow us to unambiguously reference even as-yet unconfirmed transactions, allowing for unconfirmed dependency chains; account-based smart contracts obviate this requirement.

At a high level, we can think of an *account holder* as an entity that has the necessary information to create an authorization set spending from the account in question. We denote the collection of accounts for which a given entity $\mathcal{P}$ is an account holder by $\mathcal{J}[\mathcal{P}]$, and the amount of funds $\mathcal{P}$ controls by $\|\mathcal{J}[\mathcal{P}]\|$.

5.3 zkChannels protocol

Figures 5.2, 5.3, 5.4, 5.5, and 5.6 describe the zkChannels protocol in detail. We now define and contextualize the parameters and variables used therein. We describe transactions, closing authorizations, and the escrow process with respect to a UTXO arbiter, but this generalizes for a non-UTXO arbiter.

5.3.1 Setup

– $\mathsf{Setup}(1^\lambda, \mathcal{J}^{(\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Rlock}}, f_{\mathsf{ID}})})$: On inputs of security parameter 1^λ and arbiter $\mathcal{J}^{(\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Rlock}}, f_{\mathsf{ID}})}$, outputs pp, which consists of (1) public parameters for a signature scheme Π_{Sig}, hash lock scheme Π_{Rlock}, and transaction identity function f_{ID} that are supported by the arbiter $\mathcal{J}$; and (2) public parameters for a message authentication code scheme Π_{MAC}, and a commitment scheme Π_{Com}.

– $\mathsf{Init}_{\mathcal{M}}(\mathsf{pp})$: On input pp, initializes $\mathsf{S}_{\mathsf{unlink}}$ and $\mathsf{S}_{\mathsf{spent}}$ to $\emptyset$ and outputs (1) the *external (on-network) merchant public key set* $\overline{PK}_{\mathcal{M}}$ and corresponding secret keys: $\overline{PK}_{\mathcal{M}} = (pk_{\mathcal{M}}, pk_{\mathcal{M}}^{\mathsf{claim}}, pk_{\mathcal{M}}^{\mathsf{disp}})$, $(sk_{\mathcal{M}}, sk_{\mathcal{M}}^{\mathsf{claim}}, sk_{\mathcal{M}}^{\mathsf{disp}})$, each generated by running $\Pi_{\mathsf{Sig}}.\mathsf{KeyGen}$; (2) a secret key $K \leftarrow \Pi_{\mathsf{MAC}}.\mathsf{KeyGen}$; (3) randomness r selected uniformly at random as required for Π_{Com}; and (4) a public commitment $\mathsf{com}_{\mathcal{M}} = \Pi_{\mathsf{Com}}.\mathsf{Com}(K; r)$ to ensure the merchant always uses the same MAC key.

To initialize zkChannels systemwide parameters, run $\mathsf{Setup}(1^\lambda, \mathcal{J}^{(\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Rlock}}, f_{\mathsf{ID}})})$. The merchant $\mathcal{M}$ runs $\mathsf{Init}_{\mathcal{M}}(\mathsf{pp})$ to prepare for opening payment channels with customers. The merchant uses the same longterm key material $\mathsf{com}_{\mathcal{M}}$ and $\overline{PK}_{\mathcal{M}}$ for all their channels.

5.3.2 Channel establishment

The $\mathsf{Establish}$ protocol consists of three subprotocols: $\mathsf{Initialize}$ (Figure 5.2), $\mathsf{Activate}$, and Unlink (both illustrated in Figure 5.3). In $\mathsf{Initialize}$, the customer and merchant exchange information needed to set initial channel parameters and form channel transactions. The customer provides their freshly

Customer($\mathcal{J}[\mathcal{C}]$) $\quad$ $\mathsf{Establish}(\mathsf{pp}, \mathcal{J}, \overline{PK}_{\mathcal{M}}, \mathsf{com}_{\mathcal{M}})$ $\quad$ Merchant($sk_{\mathcal{M}}, (K, r), \mathcal{J}[\mathcal{M}], \mathsf{S}_{\mathsf{unlink}}, \mathsf{S}_{\mathsf{spent}}$)

Initialize subprotocol:

Choose balance $B_0^{\mathcal{C}} \geq 0$ and generate

- $(pk_{\mathcal{C}}, sk_{\mathcal{C}}) \leftarrow \Pi_{\mathsf{Sig}}.\mathsf{KeyGen}(\mathsf{pp})$;
- $(rl_0, rs_0) \leftarrow \Pi_{\mathsf{Rlock}}.\mathsf{Gen}()$; and
- $n_0 \xleftarrow{\$} \{0,1\}^{\lambda}$.

Generate customer-side auxiliary information, $aux_{\mathcal{J}[\mathcal{C}]}$, and set

- $s_0 = (\perp, n_0, rl_0, B_0^{\mathcal{C}}, \perp)$ and
- $\bar{s}_0 = (\perp, rl_0, B_0^{\mathcal{C}}, \perp)$.

$$\xrightarrow{pk_{\mathcal{C}},\ rl_0,\ n_0,\ B_0^{\mathcal{C}},\ aux_{\mathcal{J}[\mathcal{C}]}}$$

Check that $pk_{\mathcal{C}}$, rl_0, n_0, $B_0^{\mathcal{C}}$, and $aux_{\mathcal{J}[\mathcal{C}]}$ are well-formed. If the channel will not be accepted, send reject message and abort. Otherwise, choose $B_0^{\mathcal{M}} \geq 0$ and continue.

Generate merchant-side auxiliary information $aux_{\mathcal{J}[\mathcal{M}]}$. Set

- $\mathtt{arg}_{\mathsf{esc}} = pk_{\mathcal{C}}, aux_{\mathcal{J}[\mathcal{C}]}, pk_{\mathcal{M}}, aux_{\mathcal{J}[\mathcal{M}]}, B_0^{\mathcal{C}}, B_0^{\mathcal{M}}$
- $\mathtt{txid}_{\mathsf{esc}} = f_{\mathsf{ID}}(\mathtt{escrow}(\mathtt{arg}_{\mathsf{esc}}))$;
- $\mathtt{txid}_{\mathsf{exp}} = f_{\mathsf{ID}}(\mathtt{expiry}(\overline{PK}_{\mathcal{M}}, pk_{\mathcal{C}}))$;
- $cid = \mathtt{txid}_{\mathsf{esc}} \| \mathtt{txid}_{\mathsf{exp}}$;
- $s_0 = (cid, n_0, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$; and
- $\bar{s}_0 = (cid, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$.

Compute $\sigma_0 = (\sigma_0^{\mathsf{um}}, \sigma_0^{\mathsf{uc}})$, where

$$\sigma_0^{\ell} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{M}}, \mathtt{close}(\overline{PK}_{\mathcal{M}}, \bar{s}_0, \ell))$$

$$\xleftarrow{(\mathsf{accept}, B_0^{\mathcal{M}}, aux_{\mathcal{J}[\mathcal{M}]}, \sigma_0) \text{ or } \mathsf{reject}}$$

Abort if reject, $B_0^{\mathcal{M}} < 0$, or σ_0 invalid.

Set

- $\mathtt{arg}_{\mathsf{esc}} = pk_{\mathcal{C}}, aux_{\mathcal{J}[\mathcal{C}]}, pk_{\mathcal{M}}, aux_{\mathcal{J}[\mathcal{M}]}, B_0^{\mathcal{C}}, B_0^{\mathcal{M}}$
- $\mathtt{txid}_{\mathsf{esc}} = f_{\mathsf{ID}}(\mathtt{escrow}(\mathtt{arg}_{\mathsf{esc}}))$;
- $\mathtt{txid}_{\mathsf{exp}} = f_{\mathsf{ID}}(\mathtt{expiry}(\overline{PK}_{\mathcal{M}}, pk_{\mathcal{C}}))$;
- $cid = \mathtt{txid}_{\mathsf{esc}} \| \mathtt{txid}_{\mathsf{exp}}$;
- $s_0 = (cid, n_0, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$; and
- $\bar{s}_0 = (cid, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$.

Compute the customers's authorization set for $\mathtt{escrow}$, denoted by $\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{C}]}$, using $\mathcal{J}[\mathcal{C}]$, and compute

$$\sigma_{\mathsf{exp}}^{\mathcal{C}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{C}}, \mathtt{expiry}(\overline{PK}_{\mathcal{M}}, pk_{\mathcal{C}})).$$

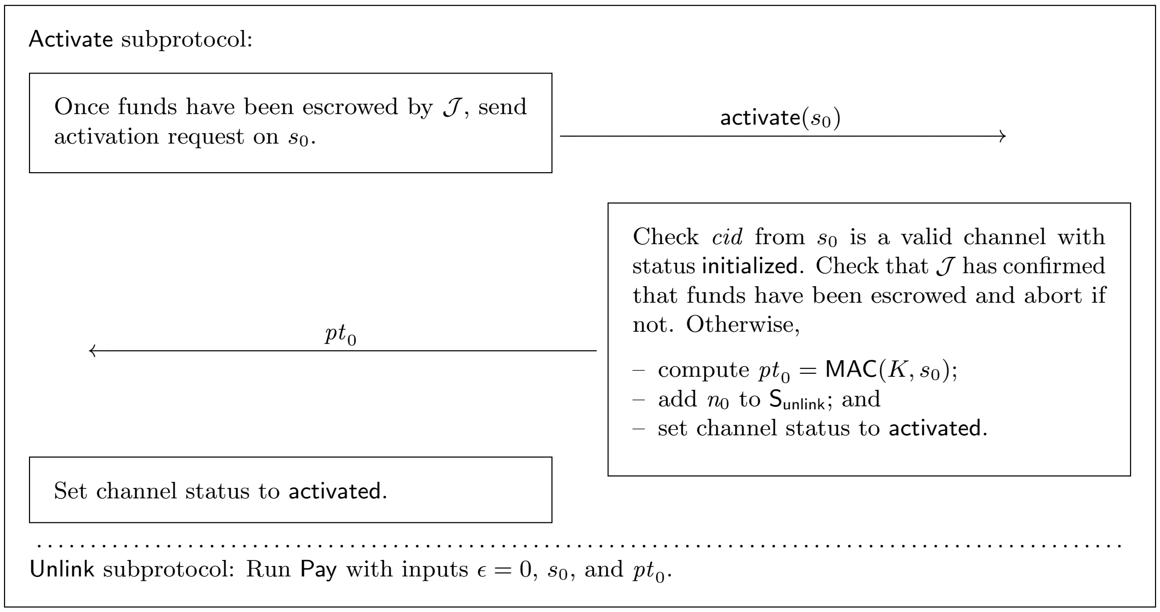
Set status to initialized.

$$\xrightarrow{\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{C}]},\ \sigma_{\mathsf{exp}}^{\mathcal{C}}}$$

Abort if $\{\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{C}]}, \sigma_{\mathsf{exp}}^{\mathcal{C}}\}$ invalid. Otherwise, set status to initialized.

If $B_0^{\mathcal{M}} = 0$, send $\mathtt{escrow}$ instructions and authorization set $\{\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{C}]}\}$ to $\mathcal{J}$. Otherwise, use $\mathcal{J}[\mathcal{M}]$ to compute the merchant's authorization set for $\mathtt{escrow}$, denoted by $\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{M}]}$, and send $\mathtt{escrow}$ instructions and authorization set $\{\sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{C}]}, \sigma_{\mathsf{esc}}^{\mathcal{J}[\mathcal{M}]}\}$ to $\mathcal{J}$.

Figure 5.2: 2PC Establish protocol with UTXO-based arbiter: Initialize


Activate subprotocol:

Once funds have been escrowed by $\mathcal{J}$, send activation request on s_0.

$\xrightarrow{\mathsf{activate}(s_0)}$

Check cid from s_0 is a valid channel with status initialized. Check that $\mathcal{J}$ has confirmed that funds have been escrowed and abort if not. Otherwise,

– compute $pt_0 = \mathsf{MAC}(K, s_0)$;
– add n_0 to $\mathsf{S}_{\mathsf{unlink}}$; and
– set channel status to activated.

$\xleftarrow{pt_0}$

Set channel status to activated.

Unlink subprotocol: Run Pay with inputs $\epsilon = 0$, s_0, and pt_0.

Figure 5.3: 2PC Establish protocol with UTXO-based arbiter: Activate and Unlink.

generated channel public key, $pk_{\mathcal{C}}$, and their initial revocation lock, rl_0. Each party $P \in \{\mathcal{C}, \mathcal{M}\}$ provides B_0^P, the amount of funds they wish to contribute to the channel, together with any auxiliary information, $aux_{\mathcal{J}[P]}$, needed by the other party in order to:

1. Form $\texttt{escrow}(pk_{\mathcal{C}}, aux_{\mathcal{J}[\mathcal{C}]}, pk_{\mathcal{M}}, aux_{\mathcal{J}[\mathcal{M}]}, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$. The escrow transaction instructions specify sending accounts for customer and merchant initial funds and encumber the channel funds with a 2-of-2 multisig requirement with respect to $pk_{\mathcal{M}}$ and $pk_{\mathcal{C}}$.

2. Form $\texttt{expiry}(\overline{PK}_{\mathcal{M}}, pk_{\mathcal{C}})$. The expiry transaction instructions specify how the merchant initiates channel closure.

3. Set the channel identifier, cid, based on the above instructions, the initial state $s_0 = (cid, n_0, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$, and the associated *closing state*, $\bar{s}_0 = (cid, rl_0, B_0^{\mathcal{C}}, B_0^{\mathcal{M}})$.[9]

4. Form $\{\texttt{close}(\overline{PK}_{\mathcal{M}}, \bar{s}_0, \ell)\}_{\ell \in \{\mathsf{um},\mathsf{uc}\}}$. The customer closing transaction instructions specify how the customer closes on the initial state during a unilateral merchant (um) and a unilateral customer (uc) close.

[9]For each customer state s, we have a special closing state, $\bar{s}$, in order to ensure the customer has an unlinkable way to close at all times.

Further details on the transactions `expiry` and `close` and their roles in closing are discussed in Section 5.3.4.

After receiving this information, the merchant authorizes the customer closing transactions by signing the instructions under $sk_{\mathcal{M}}$ and sending the resulting signature set, denoted by σ_0, to the customer. If σ_0 contains valid signatures on the expected transactions, the customer can safely authorize the escrow and expiry transactions, as they now have a way to close the channel. The customer creates (1) customer authorization, $\sigma^{\mathcal{C}}_{\mathsf{esc}}$, on the escrow transaction instructions;[10] and (2) customer authorization, $\sigma^{\mathcal{C}}_{\mathsf{exp}}$, on the expiry transaction instructions; this consists of a signature under $sk_{\mathcal{C}}$ in a UTXO network. These are sent to the merchant. If the authorization provided by the customer is valid, the merchant can safely send the escrow transaction to $\mathcal{J}$, as they now have a way to close the channel.[11]

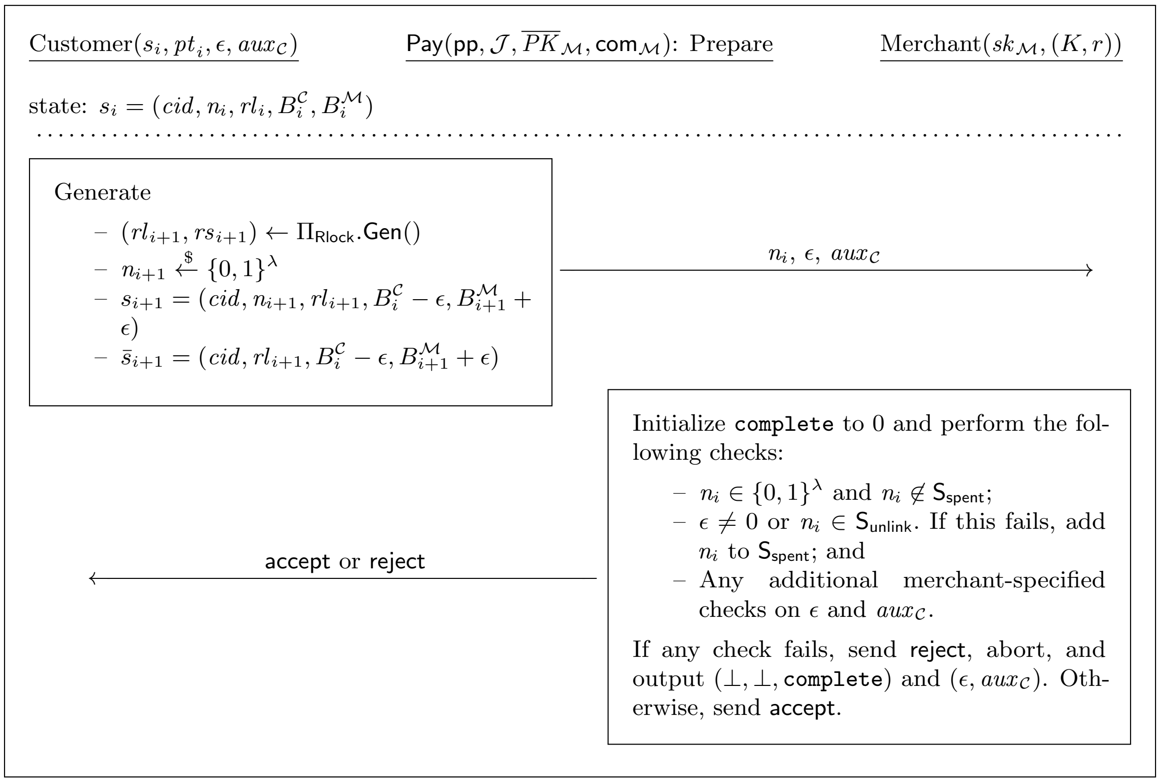
In the $\mathsf{Activate}$ subprotocol, once the escrow transaction is confirmed, the customer sends an activation request on the initial state s_0 and receives back a payment tag pt_0 (computed by applying a MAC to the state) from the merchant. We use a MAC here as an efficiently computable, designated verifier signature. This mechanism could be replaced with a full signature scheme, but doing so significantly increases the computational cost of the 2PC executed during Pay. The public commitment $\mathsf{com}_{\mathcal{M}} = \Pi_{\mathsf{Com}}.\mathsf{Com}(K; r)$ acts as a public key here, allowing the customers to enforce that the merchant always uses the same MAC key for each channel.

However, a payment made with this tag pt_0 is linkable to the customer: the merchant knows the contents of s_0 and thus n_0 and rl_0, which will be revealed during a payment on pt_0. To avoid this, we have the customer exchange their payment tag for a fresh, unlinkable tag on a new initial state $s'_0 = (cid, n'_0, rl'_0, B^{\mathcal{C}}_0, B^{\mathcal{M}}_0)$. As we see in the next section, we already have the machinery to do this in Pay, and so we have the customer make a zero payment in the *Unlink* subprotocol. To do so, the merchant stores the nonce n_0 in a special set $\mathsf{S}_{\mathsf{unlink}}$. When a payment request for amount zero is received, the merchant confirms that the nonce in the payment request is stored in $\mathsf{S}_{\mathsf{unlink}}$ and allows the payment to proceed.[12] Pay provides the customer with a new payment tag pt'_0 that is *unknown* to the merchant, revokes s_0, and provides fresh closing signatures σ'_0 on the initial balance allocation.

[10]The exact specification of the customer-side authorization depends on how the $B^{\mathcal{C}}_0$ funds are encumbered with $\mathcal{J}$ prior to opening a zkChannels.

[11]In the case of channel that is funded solely by the customer, the merchant need not provide any authorization to complete the escrow transaction.

[12]We disallow zero payments otherwise as a DoS mitigation.


Customer($s_i, pt_i, \epsilon, aux_{\mathcal{C}}$) Pay(pp, $\mathcal{J}$, $\overline{PK}_{\mathcal{M}}$, com$_{\mathcal{M}}$): Prepare Merchant($sk_{\mathcal{M}}$, (K, r))

state: $s_i = (cid, n_i, rl_i, B_i^{\mathcal{C}}, B_i^{\mathcal{M}})$

Generate

- $(rl_{i+1}, rs_{i+1}) \leftarrow \Pi_{\mathsf{Rlock}}.\mathsf{Gen}()$
- $n_{i+1} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$
- $s_{i+1} = (cid, n_{i+1}, rl_{i+1}, B_i^{\mathcal{C}} - \epsilon, B_{i+1}^{\mathcal{M}} + \epsilon)$
- $\bar{s}_{i+1} = (cid, rl_{i+1}, B_i^{\mathcal{C}} - \epsilon, B_{i+1}^{\mathcal{M}} + \epsilon)$

$\xrightarrow{n_i,\ \epsilon,\ aux_{\mathcal{C}}}$

Initialize `complete` to 0 and perform the following checks:

- $n_i \in \{0,1\}^{\lambda}$ and $n_i \notin \mathsf{S}_{\mathsf{spent}}$;
- $\epsilon \neq 0$ or $n_i \in \mathsf{S}_{\mathsf{unlink}}$. If this fails, add n_i to $\mathsf{S}_{\mathsf{spent}}$; and
- Any additional merchant-specified checks on ϵ and $aux_{\mathcal{C}}$.

If any check fails, send reject, abort, and output $(\bot, \bot, \texttt{complete})$ and $(\epsilon, aux_{\mathcal{C}})$. Otherwise, send accept.

$\xleftarrow{\text{accept or reject}}$

Figure 5.4: Pay Protocol Phase 1: Prepare

5.3.3 Channel payments

Pay has three phases: the *prepare* phase, the *update state*, and the *unmask* phase, shown in Figures 5.4, 5.5, and 5.6, respectively.

During the prepare phase, the customer sends the payment amount, ϵ, to the merchant, along with the nonce contained in their most recent state. The customer also sends auxiliary information $aux_{\mathcal{C}}$ associated with the service or good that the customer is purchasing. A natural use case is the purchase of an anonymous credential, say for a subscription to a service; in this case, $aux_{\mathcal{C}}$ details the information that should be included in the credential provided by the merchant at the end of Pay. Another example is for enabling *refunds*, *i.e.*, , if $\epsilon < 0$, the auxiliary data could include justification for the negative payment. The merchant consults their internal state to determine whether to accept the payment, including checking that the nonce is fresh.

The update state phase begins with the parties exchanging hiding and binding commitments on

Customer($s_i, pt_i, \epsilon, aux_{\mathcal{C}}$) | $\mathsf{Pay}(\mathsf{pp}, \mathcal{J}, \overline{PK}_{\mathcal{M}}, \mathsf{com}_{\mathcal{M}})$: Update State | Merchant($sk_{\mathcal{M}}, (K, r)$)

$s_i = (cid_i, n_i, rl_i, B_i^{\mathcal{C}}, B_i^{\mathcal{M}})$
$s_{i+1} = (cid_{i+1}, n_{i+1}, rl_{i+1}, B_{i+1}^{\mathcal{C}}, B_{i+1}^{\mathcal{M}})$
$\bar{s}_{i+1} = (cid_{i+1}, rl_{i+1}, B_{i+1}^{\mathcal{C}}, B_{i+1}^{\mathcal{M}})$

Sample $\rho_i \xleftarrow{\$} \{0,1\}^{\lambda}$ and compute $\mathsf{RLcom} = \Pi_{\mathsf{Com}}.\mathsf{Com}(rl_i; \rho_i)$

$\xrightarrow{\mathsf{RLcom}}$

Generate keys $K_1 \xleftarrow{\$} \{0,1\}^{2\lambda}, K_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$, and sample randomness $r_1 \xleftarrow{\$} \{0,1\}^{\lambda}$. Then form $\mathsf{PTcom} = \Pi_{\mathsf{Com}}.\mathsf{Com}(K_1; r_1)$.

$\xleftarrow{\mathsf{PTcom}}$

$\mathsf{UpdateState}\{\mathcal{C}, \mathcal{M}\}$ 2PC

Public Inputs:

$$\overline{PK}_{\mathcal{M}},\ \mathsf{com}_{\mathcal{M}},\ \epsilon,\ n_i,\ \mathsf{PTcom},\ \mathsf{RLcom}$$

The 2PC checks the following, outputing $\perp$ if any do not pass:

1. $cid_i = cid_{i+1}$, $B_{i+1}^{\mathcal{C}} = B_i^{\mathcal{C}} - \epsilon$, and $B_{i+1}^{\mathcal{M}} = B_i^{\mathcal{M}} + \epsilon$;
2. $B_{i+1}^{\mathcal{C}} \geq 0$ and $B_{i+1}^{\mathcal{M}} \geq 0$;
3. Public input $n_i \in s_i$;
4. $\Pi_{\mathsf{MAC}}.\mathsf{Verify}(K, s_i, pt_i) = 1$;
5. $\mathsf{RLcom} = \Pi_{\mathsf{Com}}.\mathsf{Com}(rl_i, \rho_i)$ for $rl_i \in s_i$;
6. $\mathsf{PTcom} = \Pi_{\mathsf{Com}}.\mathsf{Com}(K_1, r_1)$; and
7. $\mathsf{com}_{\mathcal{M}} = \Pi_{\mathsf{Com}}.\mathsf{Com}(K, r)$.

The 2PC computes:

1. $pt_{i+1} = \Pi_{\mathsf{MAC}}.\mathsf{Mac}(K, s_{i+1})$;
2. $\sigma_{i+1} = (\sigma_{i+1}^{\mathsf{um}}, \sigma_{i+1}^{\mathsf{uc}})$, where for $\ell \in \{\mathsf{um}, \mathsf{uc}\}$,

$$\sigma_{i+1}^{\ell} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{M}}, \mathtt{close}(\overline{PK}_{\mathcal{M}}, \bar{s}_{i+1}, \ell));$$

3. $c_1 = K_1 \oplus pt_{i+1}$; and
4. $c_2 = K_2 \oplus \sigma_{i+1}$.

customer secrets:

$\xrightarrow{pt_i,\ s_i,\ \rho_i,\ s_{i+1}}$

$\xleftarrow[\text{correctness of checks}]{c_1,\ c_2}$

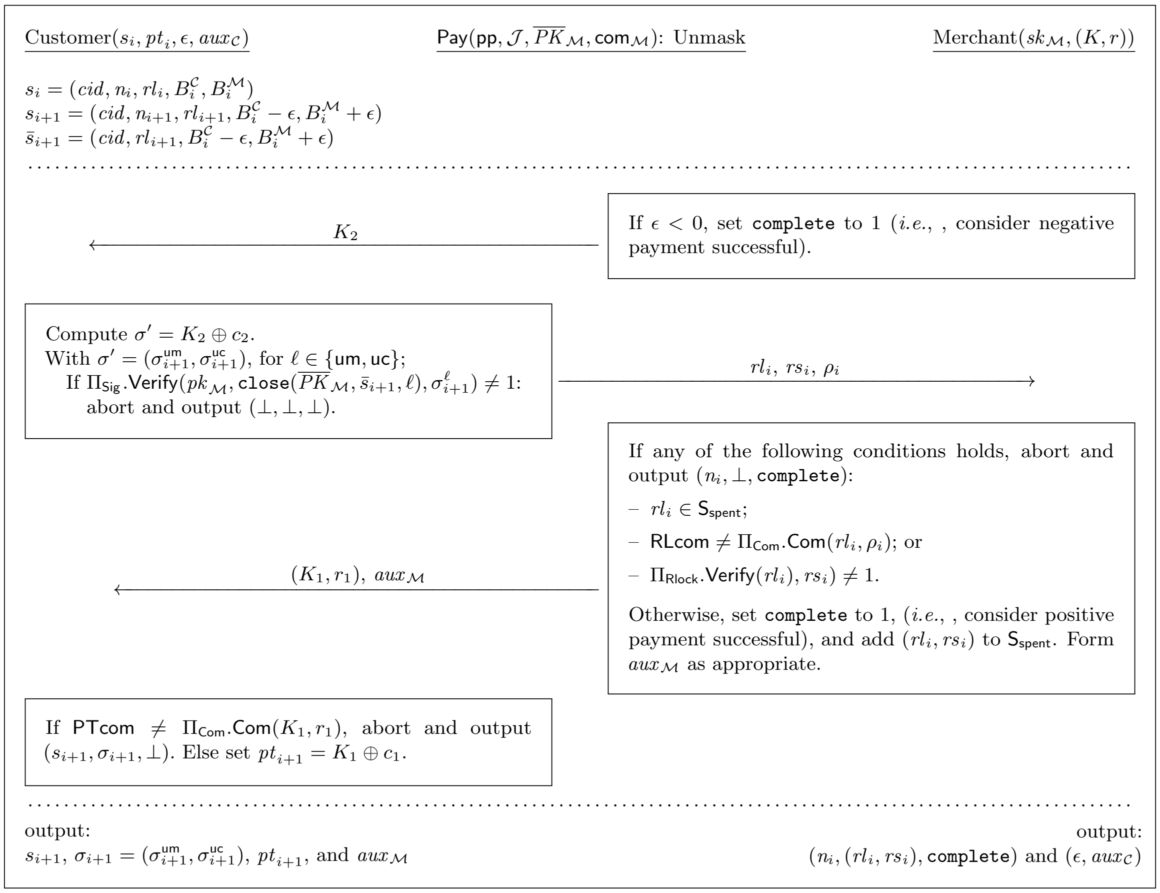
merchant secrets:

$\xleftarrow{sk_{\mathcal{M}},\ (K; r),\ (K_1, r_1),\ K_2}$

$\xrightarrow[\text{correctness of checks}]{success \in \{0,1\}}$

If UpdateState fails from perspective of merchant, abort and output $(\perp, \perp, \mathtt{complete})$. Otherwise, add n_i to $\mathsf{S}_{\mathsf{spent}}$ and continue.

Figure 5.5: Pay Protocol Phase 2: Update State



Customer($s_i, pt_i, \epsilon, aux_{\mathcal{C}}$) — $\mathsf{Pay}(\mathsf{pp}, \mathcal{J}, \overline{PK}_{\mathcal{M}}, \mathsf{com}_{\mathcal{M}})$: Unmask — Merchant($sk_{\mathcal{M}}, (K, r)$)

$s_i = (cid, n_i, rl_i, B_i^{\mathcal{C}}, B_i^{\mathcal{M}})$
$s_{i+1} = (cid, n_{i+1}, rl_{i+1}, B_i^{\mathcal{C}} - \epsilon, B_i^{\mathcal{M}} + \epsilon)$
$\bar{s}_{i+1} = (cid, rl_{i+1}, B_i^{\mathcal{C}} - \epsilon, B_i^{\mathcal{M}} + \epsilon)$

Merchant: If $\epsilon < 0$, set `complete` to 1 (*i.e.*, , consider negative payment successful).

Merchant → Customer: K_2

Customer: Compute $\sigma' = K_2 \oplus c_2$.
With $\sigma' = (\sigma_{i+1}^{\mathsf{um}}, \sigma_{i+1}^{\mathsf{uc}})$, for $\ell \in \{\mathsf{um}, \mathsf{uc}\}$;
  If $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_{\mathcal{M}}, \mathtt{close}(\overline{PK}_{\mathcal{M}}, \bar{s}_{i+1}, \ell), \sigma_{i+1}^{\ell}) \neq 1$:
    abort and output $(\perp, \perp, \perp)$.

Customer → Merchant: rl_i, rs_i, ρ_i

Merchant: If any of the following conditions holds, abort and output $(n_i, \perp, \mathtt{complete})$:
– $rl_i \in \mathsf{S}_{\mathsf{spent}}$;
– $\mathsf{RLcom} \neq \Pi_{\mathsf{Com}}.\mathsf{Com}(rl_i, \rho_i)$; or
– $\Pi_{\mathsf{Rlock}}.\mathsf{Verify}(rl_i), rs_i) \neq 1$.

Otherwise, set `complete` to 1, (*i.e.*, , consider positive payment successful), and add (rl_i, rs_i) to $\mathsf{S}_{\mathsf{spent}}$. Form $aux_{\mathcal{M}}$ as appropriate.

Merchant → Customer: (K_1, r_1), $aux_{\mathcal{M}}$

Customer: If $\mathsf{PTcom} \neq \Pi_{\mathsf{Com}}.\mathsf{Com}(K_1, r_1)$, abort and output $(s_{i+1}, \sigma_{i+1}, \perp)$. Else set $pt_{i+1} = K_1 \oplus c_1$.

Customer output:
s_{i+1}, $\sigma_{i+1} = (\sigma_{i+1}^{\mathsf{um}}, \sigma_{i+1}^{\mathsf{uc}})$, pt_{i+1}, and $aux_{\mathcal{M}}$

Merchant output:
$(n_i, (rl_i, rs_i), \mathtt{complete})$ and $(\epsilon, aux_{\mathcal{C}})$

Figure 5.6: Pay Protocol Part 3: Unmask

values used inside the 2PC and revealed after it completes; in this way, we create a binding between the actual values used in the 2PC and the unmask phase. The customer first sends a commitment RLcom to the revocation lock rl_i, thereby ensuring that the revocation lock revealed during unmask *must* be the one embedded in the provided state, s_i, while maintaining unlinkability against merchant aborts. The merchant then sends a commitment PTcom to a random key K_1 that will be used to mask the payment tag pt_{i+1}. Because the customer does not have the key K to the MAC scheme, they cannot independently verify if the MAC revealed during the unmask phase is honestly computed; committing to K_1 makes it impossible for the merchant to manipulate this output of the 2PC.

The parties then execute a maliciously secure 2PC protocol. This computation checks the correctness of each party's inputs with respect to the publicly known values, outputing $\perp$ if any of the checks fails. The 2PC computes a new payment tag, pt_{i+1}, and set of closing signatures. These values are masked with the new merchant-selected keys and output to the customer.

If the 2PC successfully completes, the parties alternate unmasking the values used and computed in the 2PC; message order compensates for the inability of 2PC to guarantee fairness. First, the merchant reveals the signatures on the new closing transactions. At this point, if the payment amount is negative, the merchant should be satisfied that the refund is complete. Once the customer is convinced that they can close on the new balances, they send the revocation pair to the merchant, invalidating the old closing transactions. Once the merchant has a valid revocation pair for the old state, they unmask the new payment tag, pt_{i+1}, allowing the customer to continue to make payments. Finally, the merchant sends the information for the service that the customer purchased, $aux_{\mathcal{M}}$.

5.3.4 Channel closing

Three types of closing are supported. We list them in order of desirability: from least to most expensive in terms of number of transactions. Closing with a UTXO arbiter requires parties to watch the network for relevant transactions. The close protocols are also formalized in Figure 5.7.

Mutual close. This subprotocol may be initiated by the customer or, if the merchant has a method to directly contact the customer, the merchant. The parties collaborate to produce a transaction $\texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}}; \sigma_{\mathcal{C}}, \sigma_{\mathcal{M}})$ that sends $B^{\mathcal{C}}$ funds to a customer account and $B^{\mathcal{M}}$ funds to a merchant account. The authorization set consists of two signatures on the instructions, $\sigma_{\mathcal{C}}$ and $\sigma_{\mathcal{M}}$, formed under $sk_{\mathcal{C}}$ and $sk_{\mathcal{M}}$, respectively. To prove correctness of the final fund allocation, the customer sends the most recent state and payment tag pair, (s, pt), to the merchant, along with $\texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}}; \sigma_{\mathcal{C}}, \cdot)$. The merchant checks the freshness of the included nonce, the validity of pt, that the transaction is well-formed, and that $\sigma_{\mathcal{C}}$ verifies. If these checks pass, the merchant adds the nonce to $\mathsf{S}_{\mathsf{spent}}$, signs the transaction instructions under $pk_{\mathcal{M}}$, and sends the completed transaction to $\mathcal{J}$ to immediately distribute the channel funds.

Unilateral customer close. The customer posts a closing transaction, $\texttt{close}(\overline{PK}_{\mathcal{M}}, \bar{s}, \mathsf{uc})$, on closing state $\bar{s} = (cid, rl, B^{\mathcal{C}}, B^{\mathcal{M}})$. This transaction sends $B^{\mathcal{M}}$ channel funds to an account encumbered under $pk^{\mathsf{claim}}_{\mathcal{M}}$, reveals rl, and timelocks $B^{\mathcal{C}}$ funds. If the customer posts an old state, the merchant can recognize rl and use the corresponding rs and $sk^{\mathsf{disp}}_{\mathcal{M}}$ to produce a dispute transaction, $\texttt{dispute}(cid, rl; rs, \sigma)$, which sends all funds to an account controlled by the merchant. Otherwise,

Close procedures

These are parameterized by the UTXO arbiter $\mathcal{J}$.

Customer-specific closing procedures:

```
on $(\mathsf{Close}, cid, type)$ from self :
  // initiate customer close procedure
  if $(type \notin \{\mathsf{um}, \mathsf{uc}, \mathsf{mutual}\})$ return
  Retrieve most recent state $s$ from memory
    where $s = (cid, n, rl, B^{\mathcal{C}}, B^{\mathcal{M}})$
  if $(type = \mathsf{mutual})\{$
    if $(chan\text{-}status \neq \textsc{open})$ return
    Retrieve payment tag $pt$ from memory
    Compute $\sigma_{\mathcal{C}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{C}}, \texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}}))$
    Send $(\mathsf{mutual\text{-}close}, s, pt, \sigma_{\mathcal{C}})$ to $\mathcal{M}$
  $\}$
  if $(type \in \{\mathsf{um}, \mathsf{uc}\}\{$
    if $(type = \mathsf{uc}$ and $chan\text{-}status \neq \textsc{open})$ return
    if $(type = \mathsf{um}$ and $chan\text{-}status \neq \textsc{pending-close})$ return
    Retrieve $\sigma^{type}$ and $aux_{cid}$ from memory
    Set $\mathsf{instr} \leftarrow \texttt{close}(\overline{PK}_{\mathcal{M}}, (cid, rl, B^{\mathcal{C}}, B^{\mathcal{M}})$
    Compute $\sigma_{\mathcal{C}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{C}}, \mathsf{instr}, type))$
    Send $\texttt{close}(\overline{PK}_{\mathcal{M}}, (cid, rl, B^{\mathcal{C}}, B^{\mathcal{M}}), type; \sigma_{\mathcal{C}}, \sigma^{type})$ to $\mathcal{J}$
    Set $chan\text{-}status = \textsc{pending-close}$
  $\}$

on $\texttt{expiry}(cid)$ from $\mathcal{J}$ :
  // procedure to respond to unilateral merchant close
  Send $(\mathsf{Close}, cid, \mathsf{um})$ to self

on $(\texttt{close}(\overline{PK}_{\mathcal{M}}, (cid, rl, B^{\mathcal{C}}, B^{\mathcal{M}}), type), time)$ from $\mathcal{J}$ :
  // Finish processing close
  Compute $\sigma_{\mathcal{C}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk^{\mathsf{claim}}_{\mathcal{C}}, \texttt{claim}(cid, \mathcal{C}))$
  At time $time + \delta'$, send $\texttt{claim}(cid, \mathcal{C}; \sigma_{\mathcal{C}})$ to $\mathcal{J}$
```

Shared closing procedures:

```
on $(\texttt{claim}(cid, \mathcal{P})$ or
    $\texttt{dispute}(cid, rl)$ or
    $\texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}}))$ from $\mathcal{J}$ :
  Set $chan\text{-}status = \textsc{closed}$
```

Merchant-specific closing procedures:

```
on $(\mathsf{Close}, cid)$ from self :
  // initiate unilateral merchant close procedure
  if $(chan\text{-}status \neq \textsc{open})$ return
  Retrieve $\sigma^{\mathcal{C}}_{\mathsf{exp}}$ and $aux_{cid}$ from memory
  Compute $\sigma_{\mathcal{M}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{M}}, \texttt{expiry}(cid))$
  Send $\texttt{expiry}(cid; \sigma^{\mathcal{C}}_{\mathsf{exp}}, \sigma_{\mathcal{M}})$ to $\mathcal{J}$
  Set $chan\text{-}status = \textsc{pending-close}$

on $(\texttt{expiry}(cid), time)$ from $\mathcal{J}$ :
  // process to finish unilateral merchant close
  Compute $\sigma_{\mathcal{M}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk^{\mathsf{claim}}_{\mathcal{M}}, \texttt{claim}(cid, \mathcal{M}))$
  At time $time + \delta$, send $\texttt{claim}(cid, \mathcal{M}; \sigma_{\mathcal{M}})$ to $\mathcal{J}$

on $\texttt{close}(\overline{PK}_{\mathcal{M}}, (cid, rl, B^{\mathcal{C}}, B^{\mathcal{M}}, type))$ from $\mathcal{J}$ :
  if $rl \in \mathsf{S}_{\mathsf{spent}}\{$
    Retrieve entry $(rl, rs)$ from $\mathsf{S}_{\mathsf{spent}}$
    Compute $\sigma_{\mathcal{M}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk^{\mathsf{disp}}_{\mathcal{M}}, \texttt{dispute}(cid, rl))$
    Send $\texttt{dispute}(cid, rl; rs, \sigma_{\mathcal{M}})$ to $\mathcal{J}$
  $\}$ else $\{$
    Add $rl$ to $\mathsf{S}_{\mathsf{spent}}$
    Set $chan\text{-}status = \textsc{closed}$
  $\}$

on $(\mathsf{mutual\text{-}close}, (cid, n, rl, B^{\mathcal{C}}, B^{\mathcal{M}}), pt, \sigma_{\mathcal{C}})$ from $\mathcal{C}_i$ :
  Retrieve $pk_{\mathcal{C}}$ for $cid$
  if $\Pi_{\mathsf{MAC}}.\mathsf{Verify}(K, s, pt) \neq 1$ return
  Set $\mathsf{instr} \leftarrow \texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}})$
  if $\Pi_{\mathsf{Sig}}.\mathsf{Verify}(pk_{\mathcal{C}}, \mathsf{instr}\sigma_{\mathcal{C}}) \neq 1$ return
  Compute $\sigma_{\mathcal{M}} = \Pi_{\mathsf{Sig}}.\mathsf{Sign}(sk_{\mathcal{M}}, \mathsf{instr})$
  Send $\texttt{mutual-close}(cid, B^{\mathcal{C}}, B^{\mathcal{M}}; \sigma_{\mathcal{C}}, \sigma_{\mathcal{M}}))$ to $\mathcal{J}$
  Set $chan\text{-}status = \textsc{pending-close}$
```

Figure 5.7: On-network close procedures for parties closing a given zkChannel, interacting directly and via the arbiter.

the customer can claim their funds with a signature under $pk_{\mathcal{C}}^{\mathsf{claim}}$ once the timelock elapses.

Unilateral merchant close. The merchant posts $\texttt{expiry}(\overline{PK}_{\mathcal{M}}, pk_{\mathcal{C}}; \sigma_{\mathcal{C}}, \sigma_{\mathcal{M}})$. This transaction spends from the escrow account and timelocks the channel funds. The funds may be spent before the timelock period passes as long as signatures under both $pk_{\mathcal{M}}$ and $pk_{\mathcal{C}}$ are presented, or after with a signature under $pk_{\mathcal{M}}^{\mathsf{claim}}$. This allows the customer time to post a closing transaction representing the most recent state, proceeding as in the unilateral customer close flow.

5.3.5 Security proof

Our high level goal is to show that our protocol allows unlinkable payments and ensures an honest player can eventually extract the money they are due from the escrowed funds of a channel. Creating a sufficiently precise formalism for this notion of security is tricky, given the complexity of the protocol and the many, interwoven sub-properties. For instance, we must simultaneously reason about how parties can deviate from the payment protocol and how that impacts both unlinkability and arbiter interactions. As such, we use a simulation-based security proof to argue the security of our construction, which argues each of these properties simultaneously. This is a departure from the work of Green and Miers [GM17], which argues the security of their protocol using multiple game-based definitions.

In the simulation-based security paradigm, a protocol designer shows that their protocol *realizes* some set of properties, formalized into an ideal functionality. The ideal functionality is a stateful entity composed of a series of callable interfaces, each of which consumes inputs, (possibly) updates internal state, and (optionally) produces output to specific parties. Intuitively, a simulation-based security proof shows that the protocol provides at least as much "security" as if the protocol participants instead interact with a trusted entity running the ideal functionality. Note that security in this context is defined implicitly by the behavior of ideal functionality. As there is no prior ideal functionality for unlinkable payment channels, the first step in our proof is to give this formal description. We briefly explain the intuition behind our definitions here. Details of the proof are omitted from this dissertation because they are primarily contributed by other authors.

Ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$. Our ideal functionality, $\mathcal{F}_{\mathsf{zkChannels}}$, captures both the flow of an unlinkable payment protocol, and the security properties it must satisfy when run between an

Ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$

Upon initialization, for each party $P \in \{\mathcal{M}, \mathcal{C}_0, \dots, \mathcal{C}_n\}$, $\mathcal{S}$ set $\mathbb{B}[P]$ to any value ≥ 0 and it is sent to P. Additionally, initialize a monotonically increasing counter $\mathsf{CurrentTime} = 0$. We allow the adversary to arbitrarily increase the counter.

Time:

```
on $(\mathsf{get\text{-}time})$ from $\mathcal{P}$ :
  Send $(\mathsf{CurrentTime})$ to $\mathcal{P}$

on $(\mathsf{increment\text{-}time})$ from $\mathcal{P}$ :
  Set $\mathsf{CurrentTime} = \mathsf{CurrentTime} + 1$
```

Establish:

```
on $(\mathsf{establish}, B_{\mathcal{C}})$ from $\mathcal{C}_i$ for some $i \in \{1, \dots, n\}$ :
  if $(B_{\mathcal{C}} < 0)$ return
  Sample unique $cid$
  Record $\mathbb{C}[cid, \mathcal{C}_i, B_{\mathcal{C}}, \perp, \textsc{escrow-pending}, \perp]$
  Send $(\mathsf{establish}, cid, \mathcal{C}_i, B_{\mathcal{C}})$ to $\mathcal{C}_i, \mathcal{M}$

on $(\mathsf{establish}, cid, B_{\mathcal{M}})$ from $\mathcal{M}$ :
  if $(\nexists \mathbb{C}[cid]$ and $\mathbb{C}[cid].\mathit{chan\text{-}status} \neq \textsc{escrow-pending})$ return
  Set $\mathbb{C}[cid].B_{\mathcal{M}} = B_{\mathcal{M}}, \mathbb{C}[cid].\mathtt{status} = \textsc{escrow-confirmation-pending}$
  Send $(\mathsf{establish\text{-}accepted}, cid, B_{\mathcal{C}}, B_{\mathcal{M}})$ to $\mathcal{C}_i, \mathcal{M}$

on $(\mathsf{reject\text{-}connection}, cid)$ from $\mathcal{M}$ :
  if $(\nexists \mathbb{C}[cid]$ or $(\mathbb{C}[cid].\mathit{chan\text{-}status} \neq \textsc{escrow-pending}))$ return
  Remove $\mathbb{C}[cid]$
  Send $(\mathsf{failed}, cid)$ to $\mathbb{C}[cid].\mathit{customer}, \mathcal{M}$

on $(\mathsf{establish\text{-}confirm}, cid)$ from $\mathcal{S}$ :
  if $(\nexists \mathbb{C}[cid]$ and $\mathbb{C}[cid].\mathit{chan\text{-}status} \neq \textsc{escrow-confirmation-pending})$ return
  if $(\mathbb{B}[\mathbb{C}[cid].\mathit{customer}] < \mathbb{C}[cid].B_{\mathcal{C}}$ or $\mathbb{B}[\mathcal{M}] < \mathbb{C}[cid].B_{\mathcal{M}})$ return
  if $(\mathbb{C}[cid].B_{\mathcal{C}} \leq 0$ or $\mathbb{C}[cid].B_{\mathcal{M}} < 0)$ return
  Set $\mathbb{C}[cid].\mathtt{status} = \textsc{escrowed}$
  Set $\mathbb{B}[\mathbb{C}[cid].\mathit{customer}] -= \mathbb{C}[cid].B_{\mathcal{C}}, \mathbb{B}[\mathcal{M}] -= \mathbb{C}[cid].B_{\mathcal{M}}$
  Send $(\mathsf{established}, cid, B_{\mathcal{C}}, B_{\mathcal{M}})$ to $\mathcal{C}_i, \mathcal{M}$

on $(\mathsf{activate}, cid)$ from $\mathcal{M}$ :
  if $(\nexists \mathbb{C}[cid]$ and $\mathbb{C}[cid].\mathit{chan\text{-}status} \neq \textsc{escrowed})$ return
  Set $\mathbb{C}[cid].\mathtt{status} = \textsc{open}$
  Send $(\mathsf{activate}, cid)$ to $\mathbb{C}[cid].\mathit{customer}$
```

Figure 5.8: Ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$, Timer and $\mathsf{Establish}$ commands.

```
on $(\mathsf{prepare\text{-}payment}, cid, \epsilon, valid, tx\text{-}in)$ from $\mathcal{C}_i$ :
  if $(\nexists \mathbb{C}[cid]$ or $\mathbb{C}[cid].customer \neq \mathcal{C}_i)$ return
  if $(\mathbb{C}[cid].pay\text{-}info \neq \bot)$ return
  if $(valid = \textbf{false}$ and $\mathbb{C}[cid].customer \notin \mathcal{S})$ return
  Sample unique $pid$
  Set $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, prepared, valid, \bot)$
  Send $(\mathsf{prepare\text{-}payment}, pid, \epsilon, tx\text{-}in)$ to $\mathcal{C}_i, \mathcal{M}$

on $(\mathsf{accept\text{-}payment}, pid, tx\text{-}out)$ from $\mathcal{M}$ :
  if $(\nexists cid$ s.t. $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, prepared, valid))$ return
  if $((\mathbb{C}[cid].chan\text{-}status \notin \{\textsc{open}, \textsc{punitive-close}\})$ or $\ldots$
          $\ldots \mathbb{C}[cid].B_{\mathcal{C}} - \epsilon < 0$ or $\mathbb{C}[cid].B_{\mathcal{M}} + \epsilon < 0) \ldots$
          $\ldots \epsilon = 0$ or $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, prepared, \textbf{false})) \{$
    Set $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, failed, valid, \bot)$
    Send $(\mathsf{payment\text{-}failed}, pid)$ to $\mathcal{C}_i, \mathcal{M}$
  } else {
    Send $(\mathsf{confirm\text{-}payment}, pid)$ to $\mathcal{S}$
    Set $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, pending, valid, tx\text{-}out)$
  }

on $(\mathsf{complete\text{-}payment}, pid)$ from $\mathcal{S}$ :
  if $(\nexists cid$ s.t. $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, pending, valid, tx\text{-}out))$ return
  Set $\mathbb{C}[cid].B_{\mathcal{C}} \mathrel{-}= \epsilon, \mathbb{C}[cid].B_{\mathcal{M}} \mathrel{+}= \epsilon$
  Set $\mathbb{C}[cid].chan\text{-}status = \textsc{open}$
  Send $(\mathsf{paid}, pid)$ to $\mathcal{M}$ and $(\mathsf{paid}, pid, tx\text{-}out)$ to $\mathcal{C}_i$

on $(\mathsf{freeze\text{-}channel}, pid, \mathcal{P})$ from $\mathcal{S}$ :
  if $(\mathcal{P}$ uncorrupted$)$ return
  if $(\nexists cid$ s.t. $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, pending, valid))$ return
  if $(\mathcal{P} = \mathcal{M})\{$
    Set $\mathbb{C}[cid].B_{\mathcal{C}} \mathrel{-}= \epsilon, \mathbb{C}[cid].B_{\mathcal{M}} \mathrel{+}= \epsilon$
    Set $\mathbb{C}[cid].pay\text{-}info = (pid, \bot, frozen, valid, tx\text{-}out)$
    Send $(\mathsf{channel\text{-}frozen}, pid)$ to $\mathcal{C}_i$
  } else if $(\mathcal{P} = \mathcal{C}_i)\{$
    Set $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, frozen, valid, tx\text{-}out)$
    Send $(\mathsf{channel\text{-}frozen}, pid)$ to $\mathcal{M}$
  } else return

on $(\mathsf{reject\text{-}payment}, pid)$ from $\mathcal{M}$ :
  if $(\nexists cid$ s.t. $\mathbb{C}[cid].pay\text{-}info = (pid, \epsilon, prepared, valid, tx\text{-}out))$ return
  Set $\mathbb{C}[cid].pay\text{-}info = (pid, \bot, frozen, valid)$
  Send $(\mathsf{reject\text{-}payment}, pid)$ to $\mathcal{C}_i, \mathcal{M}$
```

Figure 5.9: Ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$, Pay commands.

```
on $(\mathsf{merchclose}, cid)$ from $\mathcal{M}$ :
  if $(\nexists \mathbb{C}[cid]$ or $\mathbb{C}[cid].chan\text{-}status \neq \textsc{open})$ return
  Set $\mathbb{C}[cid].chan\text{-}status = \textsc{close-pending}$
  Record $\mathbb{T}[cid, \mathsf{CurrentTime} + \delta, \perp, 0, \mathbb{C}[cid].B_{\mathcal{C}} + \mathbb{C}[cid].B_{\mathcal{M}}]$
  Send $(\mathsf{close\text{-}pending}, cid)$ to $\mathcal{C}_i$

on $(\mathsf{custclose}, cid, type)$ from $\mathcal{C}_i$ :
  if $((\nexists \mathbb{C}[cid])$ or $(\mathcal{C}_i \neq \mathbb{C}[cid].customer))$ return
  if $(\mathbb{C}[cid].chan\text{-}status \notin \{\textsc{open}, \textsc{close-pending}\})$ return
  if $(type = \mathsf{mutual})\{$
    if $(\mathbb{C}[cid].chan\text{-}status \neq \textsc{open})$ return
    Set $\mathbb{C}[cid].chan\text{-}status = \textsc{closed}$
    Record $\mathbb{T}[cid, \perp, \perp, \mathbb{C}[cid].B_{\mathcal{C}}, \mathbb{C}[cid].B_{\mathcal{M}}]$
  $\}$ else $\{$
    Set $\mathbb{C}[cid].chan\text{-}status = \textsc{close-pending}$
    Set $\mathbb{T}[cid]$ to $\mathbb{T}[cid, \perp, \mathsf{CurrentTime} + \delta', \mathbb{C}[cid].B_{\mathcal{C}}, \mathbb{C}[cid].B_{\mathcal{M}}]$
  $\}$
  Send $(\mathsf{close\text{-}pending}, cid, \mathbb{C}[cid].B_{\mathcal{C}}, \mathbb{C}[cid].B_{\mathcal{M}}, type)$ to $\mathcal{M}$

on $(\mathsf{close}, cid, B'_{\mathcal{C}}, B'_{\mathcal{M}})$ from $\mathcal{S}$ :
  if $(\nexists \mathbb{C}[cid])$ or $(\mathbb{C}[cid].customer \notin \mathcal{S})$ return
  if $(\mathbb{C}[cid].chan\text{-}status \notin \{\textsc{escrow-pending}, \textsc{open}, \textsc{close-pending}\})$ return
  if $(\mathbb{C}[cid].B_{\mathcal{C}} + \mathbb{C}[cid].B_{\mathcal{M}} \neq B'_{\mathcal{C}} + B'_{\mathcal{M}})$ return
  if $(\mathbb{T}[cid].cust\text{-}timelock \neq \perp)$ return
  if $((pay\text{-}info = (pid, \epsilon, frozen, valid, tx\text{-}out))$ and $((B'_{\mathcal{C}} = \mathbb{C}[cid].B_{\mathcal{C}} - \epsilon)$ and $(B'_{\mathcal{M}} = \mathbb{C}[cid].B_{\mathcal{M}} + \epsilon))\{$
    Set $\mathbb{C}[cid].chan\text{-}status = \textsc{close-pending}$
    Set $\mathbb{T}[cid]$ to $\mathbb{T}[cid, \perp, \mathsf{CurrentTime} + \delta', B'_{\mathcal{C}}, B'_{\mathcal{M}}]$
  $\}$ else if $((B'_{\mathcal{C}} \neq \mathbb{C}[cid].B_{\mathcal{C}})$ or $(B'_{\mathcal{M}} \neq \mathbb{C}[cid].B_{\mathcal{M}}))\{$
    Set $\mathbb{C}[cid].chan\text{-}status = \textsc{punitive-close}$
    Update $\mathbb{T}[cid]$ to $\mathbb{T}[cid, \perp, \perp, 0, B'_{\mathcal{C}} + B'_{\mathcal{M}}]$
  $\}$

on $(\mathsf{cashout}, cid)$ from $\mathcal{P}$ :
  if $(\nexists \mathbb{T}[cid]$ or $\mathcal{P} \notin \{\mathbb{C}[cid].customer, \mathcal{M}\})$ return
  if $(\mathbb{C}[cid].chan\text{-}status \in \{\textsc{closed}, \textsc{punitive-close}\})\{$
    Set $\mathbb{B}[\mathbb{C}[cid].customer] = \mathbb{T}[cid].B_{\mathcal{C}}, \mathbb{B}[\mathcal{M}] = \mathbb{T}[cid].B_{\mathcal{M}}$
    Send $(\mathsf{cashout}, cid, \mathbb{T}[cid].B_{\mathcal{C}}, \mathbb{T}[cid].B_{\mathcal{M}})$ to $\mathcal{P}, \mathbb{C}[cid].customer, \mathcal{M}$
  $\}$ else $\{$
    Let $(t_1, t_2) = (\mathbb{T}[cid].merch\text{-}timelock, \mathbb{T}[cid].cust\text{-}timelock)$
    if $((t_1 \neq \perp$ has passed$)$ and $(\mathcal{P} = \mathcal{M})$ or $((t_2 \neq \perp$ has passed$)$ and $(\mathcal{P} = \mathbb{C}[cid].customer)))\{$
      Set $\mathbb{C}[cid].chan\text{-}status = \textsc{closed}$
      Set $\mathbb{B}[\mathbb{C}[cid].customer] = \mathbb{T}[cid].B_{\mathcal{C}}, \mathbb{B}[\mathcal{M}] = \mathbb{T}[cid].B_{\mathcal{M}}$
      Send $(\mathsf{cashout}, cid, \mathbb{T}[cid].B_{\mathcal{C}}, \mathbb{T}[cid].B_{\mathcal{M}})$ to $\mathcal{P}, \mathbb{C}[cid].customer, \mathcal{M}$
    $\}$ else return
  $\}$
```

Figure 5.10: Ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$, Close commands.

arbitrary number of customers and a single merchant. The ideal functionality integrates the desired properties of both payment channels and currency management (handled by the arbiter in the real protocol). The ideal functionality has three main sets of interfaces: (1) interfaces for establishing a channel, (2) interfaces for making payments, and (3) interfaces for closing a channel. Our ideal functionality provides the properties described above by controlling how its internal state is updated and the information it shares with each party. $\mathcal{F}_{\mathsf{zkChannels}}$ is defined in Figures 5.8, 5.9, and 5.10.

To establish a channel, the ideal functionality exposes an interface for customers to request a new channel; a merchant can accept or reject this request. This interaction explicitly leaks the identity of the customer to the merchant and internally handles the escrow of funds. After the merchant calls an interface for activating the channel, the customer can initiate payments. When the customer offers to make a payment, the ideal functionality sends a notification to the merchant that *does not leak* the customer's identity. The merchant can accept the payment, which increments balances in a table held by the ideal functionality, or reject the payment, which freezes the channel. Finally, either party can initiate a channel closure; the ideal functionality ensures that channels close on proper balances or punishment is meted out, as appropriate.

To adequately capture the capabilities of the adversary, there are additional interfaces that only the adversary can call. These interfaces allow the adversary to adaptively "back-out" of opening a channel or making a payment on behalf of corrupted parties, capturing the real-world case of a party maliciously aborting the protocol or sending a bad message. Backing out of a payment freezes the channel, blocking all future payments; if the adversary freezes a channel during a payment, they are able to force the channel to close on either the pre- or post-payment balance.

Hybrid model and security results. We define an ideal UTXO-arbiter functionality, which acts as a funds-controlling entity that allows for movement of funds according to pre-specified authorization rules. Our motivation is to formalize the range of operations we utilize in our instantiation of zkChannels on a real-world UTXO-cryptocurrency like Bitcoin. Our definition captures the basic notion of accounts on a UTXO-network and the types of encumbrances we need for zkChannels, namely revocation locks, public-key signature requirements, and relative timelocks. We parametrize the arbiter ideal functionality with schemes Π_{Rlock} and Π_{Sig} as tools to realize the first two encumbrances; the arbiter also has access to a helper scheme Π_{ID}, to create unique identifiers for network transactions. We also make use of a 2PC-ideal functionality, $\mathcal{F}_{\mathsf{2PC}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Com}},\Pi_{\mathsf{MAC}}}$, with non-black-box use of $\Pi_{\mathsf{Sig}}, \Pi_{\mathsf{Com}}$, and Π_{MAC}.

We then prove that our zkChannels protocol Π realizes the ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$ in the (2PC, arbiter)-hybrid model; that is, we assume Π has access to these functionalities. For our corruption model, we consider probabilistic polynomial time (PPT) adversaries with abort capabilities, and allow the adversary to corrupt either the merchant or any subset of customers. We show, in the standalone model, in the presence of admissible corruptions, the following:

Theorem 1. *The zkChannels protocol* Π *securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{zkChannels}}$ *in the* $(\mathcal{F}_{\mathsf{2PC}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Com}},\Pi_{\mathsf{MAC}}}, \mathcal{F}_{\mathsf{Arbiter}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Rlock}},\Pi_{\mathsf{ID}}})$*-hybrid model with abort in the presence of malicious adversaries.*

Proof. The proof is deferred to [AGH+21]. □

This result immediately implies the following corollary:

Corollary 1. Given secure realizations of $\mathcal{F}_{\mathsf{2PC}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Com}},\Pi_{\mathsf{MAC}}}$ and $\mathcal{F}_{\mathsf{Arbiter}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Rlock}},\Pi_{\mathsf{ID}}}$, the zkChannels protocol Π securely realizes the ideal functionality $\mathcal{F}_{\mathsf{zkChannels}}$ with abort in the presence of malicious adversaries.

That is, as long as we securely instantiate our primitives, zkChannels achieves the desired properties. In particular, any real-world UTXO arbiter that realizes the functionality captured by $\mathcal{F}_{\mathsf{Arbiter}}^{\Pi_{\mathsf{Sig}},\Pi_{\mathsf{Rlock}},\Pi_{\mathsf{ID}}}$ will suffice. The Bitcoin instantiation of the arbiter is quite straightforward, relying on ECDSA signatures, a SHA-256 hash lock scheme, and the standard SHA-256-based mapping of serialized Bitcoin transaction instructions to transaction identifiers. We provide more details on our implementation choices in Section 5.4.

5.3.6 Protocol discussion

The main focus of zkChannels is the achievement of customer payment history privacy. This choice involves certain tradeoffs. While we always guarantee an unlinkable method to close and that no honest party loses their share of the escrowed funds, we do not ensure (1) the usability of the channel for future payments; (2) the merchant actually provides the requested service; or, most interestingly, (3) payments occur only on channels with an open on-network escrow account. We argue that the lack of these properties is relatively benign in the context of a traditional customer-merchant relationship.

Table 5.1: Gate counts for modules in the UpdateState circuit (K=thousand).

Module	AND	XOR	INV
UpdateState	2510K	7220K	92K
SHA-256 (1 block)	42K	118K	2K
SHA-256 (5 blocks)	71K	212K	4K
ECDSA	673K	1760K	1K

From the perspective of a customer, the most natural use cases for a point-to-point private payment channel involve a merchant who is *semi-trusted*. That is, the merchant either already has an established reputation or the customer is willing to build a trusted relationship with an unknown merchant over the course of a sequence of payments. The customer does not risk their entire channel balance in engaging with the merchant; they risk only a single payment amount, a risk profile that maps well to current practice. In other words, we do not attempt to replace all mechanisms of trust in society with zkChannels.

The perspective of the merchant with respect to the lack of property (3) above is a more interesting question. Say the customer makes a series of payments corresponding to intermediate states $s_0, s_1, \ldots, s_m$, and then closes the channel on s_i for some $i < m$. Our use of on-network punishment actually amounts to a *soft punishment*: the customer is still able to make a valid, correct off-network payment using the most recent state s_m, since they do not reveal the associated nonce or revocation lock on closing. This amounts to the conversion of the payment channel from one backed by on-network funds to one backed by a merchant gift card. The issue is that the transactions the customer may use to close the escrow account do not include enough information for the merchant to infer anything about the closing state *except whether or not it is outdated.* That is, we do not provide a reconciliation mechanism for the merchant to derive the most recent state and prevent further payments. However, we stress that, in this situation, the customer cannot make payments for which they have not already transferred the corresponding funds to the merchant.

Let G be an elliptic curve point of order q, $x \in \mathbb{Z}_q$ be a private key, m be a message, and H be a hash function.

```
**Merchant.** On input $x$:
    Sample $k \leftarrow \mathbb{Z}_q$.
    Define $(r_x, r_y) = k \cdot G$.
    Compute $rx = r_x \cdot x \mod q$.
    Calls **2PC** with input $(k^{-1}, rx)$.
**Customer.** On input $m$:
    Call **2PC** with input $(m)$
**2PC.** On inputs $(m, k^{-1}, rx)$:
    Compute $s = k^{-1} \cdot (H(m) + rx) \mod q$
    Output $s$.
```

Figure 5.11: Computing ECDSA under 2PC with precomputation. The Merchant and Customer sections are computed locally in the clear; 2PC is computed securely.

5.4 Implementation

5.4.1 Instantiation of cryptographic primitives

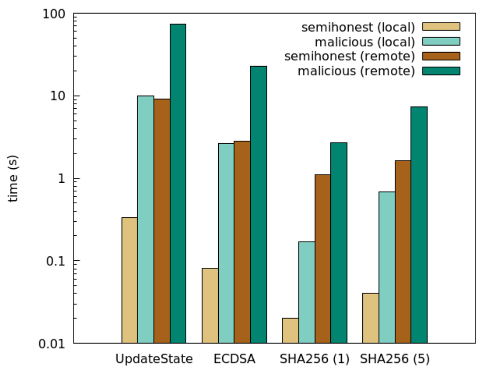
Our implementation targets the $\lambda = 128$ security level. We instantiate the hash function H, used by Π_{Com} and Π_{Rlock}, with SHA256 as defined in FIPS PUB 180-4 [Dan15]. We instantiate Π_{MAC} using a standard HMAC described in RFC 2104 [KBC97], also using SHA256 for a hash function. We instantiate Π_{Sig} using ECDSA [Por13], since Bitcoin transactions require ECDSA signatures with SHA256.

5.4.2 2PC with EMP-toolkit

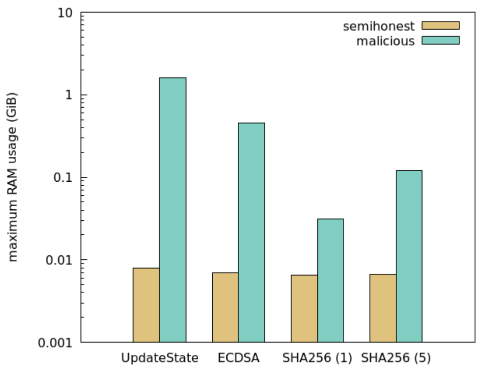
We instantiate the 2PC protocol required by UpdateState using the EMP-toolkit [WMK16] software framework. This provides multiple garbled circuit protocol implementations; we use the optimized implementation of Yao's semi-honest protocol [Yao86] and a maliciously secure protocol based on [WRK17a].

Garbled circuit protocols represent functions as a boolean circuit and data as wires in the circuit. This lends itself to an efficient SHA256 implementation. EMP-toolkit provides a high-level C++ library to express functions and produces two output streams: it can directly execute the Yao semi-honest protocol, or it can compile the entire computation into a circuit. The circuit file is used as input to other protocol implementations provided by the framework.

The EMP-toolkit circuit compiler imposes some limitations on the software description of the



(a) Total wall-clock time for tests.



(b) Total RAM usage for local tests (This did not change in remote tests).

Figure 5.12: Average (over 100 trials) of runtime and RAM usage for a selection of operations used in the UpdateState protocol. Note the logarithmic scale: in both settings, the malicious implementation is 1-2 orders of magnitude more expensive than semihonest.

function it compiles, some of which are due to the circuit file format [ST]. For example, all variables must be declared before any of them can be used.

The circuit format assigns all inputs to exactly one party. Shared public inputs must be provided by both parties (*e.g.* as private input) and compared for correctness. Constant values (such as the ECDSA modulus) must be passed in the same way; we define a `constants` module to contain such values. The format does not specify a party for output values, so EMP-toolkit can either provide output to one party, or provide it to both and does not support different outputs for each party.

Finally, the EMP-toolkit malicious framework takes input as a bit array. Since our input contains complex types, we implemented a translation layer that converts inputs to their boolean representation and assigns them to the correct wires in the circuit.

5.4.3 Implementation

We implemented a complete, tested version of zkChannels in Rust and C++.[13] The vast majority of the implementation is in Rust; we use C++ only for the UpdateState functionality in the Pay protocol. The Rust implementation follows a modular design that comprises an on-network component that builds Bitcoin transactions (for escrow and closing) and an off-network component that establishes the channel and interacts with UpdateState. The on-network component relies on Rust generics and traits to instantiate for Bitcoin or any other UTXO-based blockchain.

[13]Our code is public at `https://github/boltlabs-inc/libzkchannels`

The UpdateState component is implemented in C++ using the EMP-toolkit framework. Some cryptographic primitives we implement for UpdateState are not tailored to a boolean circuit representation. To produce data-oblivious, boolean-circuit-friendly representations, we narrowed the scope of these generic primitives. For example, the SHA256 specification takes an arbitrary length message, pads it to a multiple of the block length, and hashes each block, using the hash of the previous block as input (and using a fixed value for the first block). However, the data-independent circuit model requires a fixed upper bound on the size of a loop. Instead, we define separate functions for each block size needed and pad messages outside the module.

Standard ECDSA requires elliptic curve point multiplication and inversion, which are expensive in the boolean circuit model. In our protocol, the customer provides the message m while the merchant provides the signing key x and randomness k. This lets the merchant precompute the terms in an ECDSA signature that require curve operations and provide them as input to the secure computation (Figure 5.11). This significantly reduces the scope of the operations computed under 2PC to only require modular arithmetic and a hash function.

The UpdateState functionality must construct valid Bitcoin transactions that encode the new state of the channel and spend from the correct sources. Bitcoin transaction formatting is notoriously complex. To perform this validation efficiently, we define a fixed format for zkChannels Bitcoin transactions.

5.4.4 Implementation limitations

We pass some Bitcoin-specific parameters to the UpdateState implementation that are not explicitly described in Figure 5.5. In order to ensure the close transaction will be accepted onto the blockchain, it must reserve some money from the channel balances to pay a market-rate fee. Since this may change over the lifetime of a channel, we pass as public parameters a fee amount that is paid by a second transaction spending from the first (the "child-pays-for-parent" paradigm [Daf16]). We also pass minimum balances for each party, which ensure the transaction outputs stay above the dust limit, and a self-delay value, which forces the customer to wait before spending a `close` transaction.

Our ECDSA signature optimization in Figure 5.11 has a privacy limitation. ECDSA signatures are not randomizable [FKP16], so an adversarial merchant can keep a record of k, ϵ pairs. If a customer publicizes a signature (*e.g.* posts it the blockchain to close the channel), the merchant can identify the signature by its randomness k and associate that customer with their final transaction. One mitigation is to allow customers to make a "dummy payment" with $\epsilon = 0$, which effectively decouples their channel from any meaningful transactions. Another is to explore use of Schnorr signatures, which may be easier to compute under 2PC and are likely to be added to Bitcoin [WNR].

5.4.5 Bitcoin instantiation

In the zkChannels implementation on Bitcoin, we address practical matters not surfaced in the abstract protocol. This ranges from efficiency tweaks to improving deployability features.

1. We include `hashPrevouts` for both `expiry` and `escrow` in the definition of *cid*, instead of simply $\texttt{txid}_{\mathsf{esc}} \| \texttt{txid}_{\mathsf{exp}}$. These values are computable from their respective transaction instructions, and we include them to avoid extra rounds of hashing in the 2PC.

2. We explicitly handle transaction fees by setting them as a parameter within the channel state.

3. We use a *childs-pays-for-parent (CPFP)* construction (similar to Lightning Network *anchor outputs*) to allow the participants to independently bump up the fees for their transactions, just in case the original fee was insufficient for the transaction to be confirmed.

5.5 Benchmarks

We evaluate the performance of our implementation, including end-to-end execution of the Establish and Pay protocols and the UpdateState function in isolation.

We measured the total time spent on the Establish and Pay stages of the protocol using a command line interface which executes the full protocol but does not interact with the blockchain. We ran our tests on an AWS `t2.medium` instance with 2 virtual CPUs and 4GiB of RAM running Ubuntu 20.04. We ran both parties locally and used the semi-honest 2PC protocol. We executed each protocol 100 times.

The vast majority of time spent on these stages is taken up by the 2PC computation. Establish takes, on average, 380ms to execute, and spends 371ms (97.6%) executing 2PC. Pay takes 379ms to execute and spends 370ms (97.7%) executing 2PC.

5.5.1 UpdateState performance

We evaluated the UpdateState function, which clearly is the bottleneck of the zkChannels protocol. For each measurement, we looked at the full function as well as SHA256 module (with 1- and 5-block inputs) and our modified ECDSA module.

We report on circuit sizes in Table 5.1. In garbled circuit protocols, AND gates are more expensive to evaluate than XOR and INV gates.

Next, we measured total time and maximum RAM usage when executing the circuits with the semihonest and malicious garbled circuit protocols. We deployed the tests on Amazon AWS instances running Amazon Linux on Intel Xeon processors, with the tests themselves deployed in a Docker container. In the local test, we executed both parties on the same machine: a "t2.large" instance with 2 virtual CPUs and 8 GiB of RAM, based in the "us-east-2" region, in Ohio. In the remote test, we deployed one party on the Ohio-based machine, and the other on a "t2.medium" instance with 2 virtual CPUs and 4 GiB of RAM based in the "us-west-2" region, in Oregon.

For each trial, we execute the circuit 100 times and report the average time and RAM usage. The results are reported in Figure 5.12. The malicious implementation is 1-2 orders of magnitude more expensive (in both time and RAM usage) than the semi-honest version. The remote tests are significantly slower than the local verisons due to network latency, but have nearly identical RAM usage. We note that the two parties had similar resource usage, despite different roles in the protocols.

5.5.2 Discussion

Our implementation is comparable to other payment channels such as the Bitcoin Lightning network [PD16], which reports payment times of "milliseconds to seconds", and TumbleBit [HAB^{+}17], which averages payments of 1.2 seconds.

We have not optimized the circuits used in our computation; minimizing the circuit specification would reduce the total computation. For instance, Goldfeder cites a SHA-256 circuit [Gol] with half as many AND gates as ours for a single round.

Algorithmically, although network communications can pose a significant amount of overhead, we may be able to modify the protocol to use a streaming model, where the garbler sends early layers of the circuit before it finishes garbling later layers. The Obliv-C [ZE15] framework has done something similar in the semi-honest model. It may also be possible to improve software parallelization in the protocol implementation. Finally, it is reasonable for a merchant to deploy servers in many regions to minimize network latency.

5.6 Related work

Anonymity for payment channels is an active area of research which can largely be divided into two areas: unlinkability for individual payments on a channel and routing privacy in payment channel networks.

Unlinkability in individual payment channels or hub networks has limited prior work. This chapter's introduction compares this work to Bolt [GM17], by Green and Miers, which introduced the notion of individually unlinkable payment channels. TumbleBit, by Heilman et al. [HAB$^+$17], operates in the hub model, where each participant opens a channel with a central *tumbler*, who facilitates payments between any pair. It operates in an eCash-like model [Cha83] and can only send fixed denominations. The unlinkability property of TumbleBit and its successors [TMM19] applies to the tumbler, who cannot link the sender and receiver of a given payment, but not to the payment parties themselves as in our work.

Routing privacy of PCNs has received more attention; here the privacy of a payment is maintained by routing a payment through a onion-routing style network [MMK$^+$17, MMS$^+$19]. This approach faces challenges practical challenges with respect to network topology and linkability concerns from payment amounts. Several works [MSBL$^+$20, ZLL$^+$18, TMSS20] consider the problem of adding non-anonymous payment channels to privacy preserving cryptocurrencies.

Recent work has demonstrated that MPC and 2PC are practical for deployment. The Zcash Foundation [BGM17], Boston Women's Workforce Council [LJDA$^+$18, BLV17] and Estonian Tax and Customs Board [BJSV15] have all used MPC techniques to address real-world problems. For an overview of the current tooling for generic MPC, see [HHNZ19] and the citations therein. There has also been significant recent research effort on threshold ECDSA [Lin17, DKLs18, LN18, CGG$^+$20], motivated by the use of ECDSA by cryptocurrencies. These techniques apply in cases where multiple entities each hold shares of an ECDSA key and must collaborate to produce a signature; this is different from the single-keyholder blind ECDSA signature we require in zkChannels.

A related line of work examines ways to approximate fairness in secure two- or multi-party computation by financially penalizing dishonest parties [BK14, ADMM14, KB14]. These works aim to compensate honest parties in generic MPC computations in the case of a malicious abort. We use similar techniques to ensure valid close procedures on Bitcoin (including hash locks and timelocks), but these works operate in the "single-execution" setting (that is, they guarantee fairness for a single secure computation) and do not provide the infrastructure needed to maintain privacy and correctness across multiple 2PC executions in a payment channel.

5.7 Conclusion

In this work, we present zkChannels, the first fully privacy-preserving payment channel protocol that integrates seamlessly with existing payment networks. Our protocol allows a customer to send and receive anonymous and unlinkable point-to-point payments, achieves better privacy properties than prior work, and offers critical features that overcome previous deployment barriers. We provide both a formal, simulation-based proof of security in the standalone model and an end-to-end, tested, open-source implementation for Bitcoin. To our knowledge, our work is one of the first implemented, substantial uses of generic two-party computation techniques. We believe that zkChannels is an important step towards efficient, usable, and private payment networks.

Our contribution includes several techniques and tools that may be of independent interest. On the implementation side, our very large circuit may be helpful pushing 2PC/MPC benchmarking beyond toy examples. On the theory side, we provide an abstraction of the arbiter needed to realize payment channels; we believe this abstraction is useful in considering the many possible extensions to these protocols, such as generalizations of state beyond tracking channel balances, and more complex arbitration rules. We also touch on novel tricks that can be used to achieve various privacy and closing properties.

CHAPTER 6

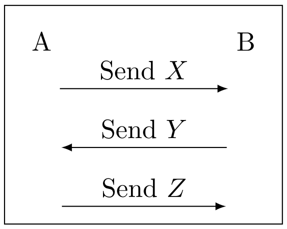
Cocoon: Production-Quality Secure Computation

Cocoon is a Microsoft software framework for implementing secure computation protocols. The framework, implemented in C#, provides a robust architecture that allows users to implement two-party protocols. It abstracts away from many of the burdensome practical complexities of such a protocol, especially those related to networking, message routing for concurrent protocol executions, and message serialization.

This chapter describes an implementation of the BaRK private set intersection protocol [KKRT16] in the Cocoon framework. It also describes an integration of the protocol into Seclūd, a platform designed to automatically deploy privacy-preserving applications using Microsoft Azure resources. The goals of this chapter are twofold: first, to describe the software architecture and development process used to produce production-quality code; second, to give a sense of the computational resources and infrastructure required to deploy such a secure computation.

To implement a protocol in the Cocoon framework, one defines information and data structures used by the protocol, including parameters (constants defined for a whole protocol execution) and internal state (elements that may change or update). The body of the protocol is a series of message handlers. Each party defines their behavior on receipt of a specific type of message; their actions may include internal computation and state update, recursively calling another protocol, setting an output value, and creating and sending a message in response.

Unlike other software frameworks in this dissertation, Cocoon can be used to implement either general-purpose MPC protocols or optimized, special-purpose protocols. This chapter focuses on *private set intersection* (PSI), a primitive that allows two or more parties to compute the intersection



```
  // in Party A's protocol description
  async Task HandleMessageAsync(YMessage y) { ... }

// in Party B's protocol description
async Task HandleMessageAsync(XMessage x) { ... }
async Task HandleMessageAsync(ZMessage z) { ... }
```

Figure 6.1: Cocoon protocols are implemented on a message-by-message basis, matching the traditional format for theoretical protocol specifications.

of their sets without revealing any other elements. It can be computed using general-purpose MPC techniques, but a wide body of literature describes efficient, special-purpose functions for computing PSI.

The Cocoon work includes two main projects: (1) Refactoring an existing BaRK protocol implementation to better reflect its modular, compositional structure; and (2) Integrating the implementation into a larger developer platform to enable use by a more diverse set of users.

6.1 BaRK protocol and implementation

This work uses the PSI protocol by Kolesnikov et al. [KKRT16]. They define an oblivious pseudo-random function (OPRF), a two-party functionality for computing a keyed pseudo-random function. One party provides an input value and receives the PRF evaluated on that input; the other party provides no input and receives the key, as described in Figure 6.2. Kolesnikov et al. drop their OPRF protocol (referred to as BaRK-OPRF) into an existing semi-honest PSI protocol [PSSZ15] to produce BaRK.

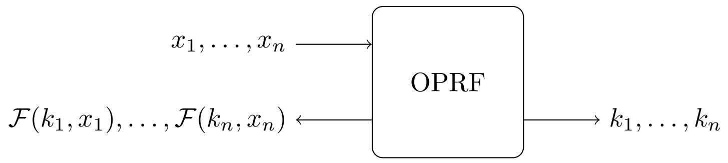


Figure 6.2: High-level functionality for a batched oblivious pseudo-random function computing a pseudo-random function $\mathcal{F}$ on inputs $\mathbf{x} = x_1, \dots, x_n$ with keys $\mathbf{k} = k_1, \dots, k_n$.

At the beginning of the project, the Cocoon development team had already defined an abstract PSI class (a generic wrapper that describes the general PSI functionality) and had implemented the BaRK protocol as an instantiation of this class. However, the monolithic BaRK implementation did not separate the OPRF component from the overall protocol.

The main contribution of this project is an new architecture and implementation of the BaRK protocol with an explicit separation between the PSI and OPRF components. The OPRF component now has an explicit API to abstract from the specific implementation and simplify future versions with different OPRF protocols. The API expands the basic OPRF definition to allow the key holder to compute the PRF on their own inputs with the keys they receive as output from the protocol. This restricts its utility primarily for use in PSI protocols, where the goal is to mask multiple input sets with the same set of keys.

The new implementation includes a robust class for OPRF inputs that enforces the invariants required by the protocol. For example, the calling protocol passes parameters for a data structure used by the OPRF; the input class checks that the data structure is large enough to hold all the input elements.

The updated BaRK PSI protocol is greatly simplified: Party A initializes the OPRF sub-protocol on her inputs. When it completes, A passes its outputs to B, who finds the intersection of hashes and adds the corresponding input elements to the output set.

Integration tests. The implementation is augmented with a suite of integration tests for the PSI protocol. They run on both fixed and random data of many sizes and compare the results of the two-party protocol to the intersection computed using standard C# set libraries. The tests are included in the repository's continuous integration; this means all code merged into the main branch of the repository must pass these tests. These currently have parameters hard-coded for the BaRK PSI protocol, but the tests themselves are not BaRK-specific and it would not be difficult to expand the suite to test other PSI protocol implementations.

6.2 Seclūd integration

Seclūd[1] is a Microsoft product that deploys privacy-preserving applications on Microsoft Azure services. It is designed to make secure applications accessible without a cryptographic background. The product offers end-to-end deployment of secure applications, such as private prediction on homomorphically encrypted data. For private set intersection, a typical flow requires each participant

[1]`https://seclud.microsoft.com`

to specify the Azure database containing their input set; Seclūd deploys the computing infrastructure that will execute the protocol, computes the result, and writes it to the recipient's database. The second contribution of this chapter is the design and implementation of an integration of the updated BaRK protocol into Seclūd.

The use case we imagined has two parties whose data are changing over time. They want to compute PSI on a regular basis to find new intersection elements. One party deploys a Cocoon server that handles PSI requests. This server is hosted in a Docker container and sits behind a Kubernetes load balancer, allowing the server party to handle multiple concurrent requests.

The ideal client infrastructure is a simple web server. On request from the user, the API spins up a new Docker container that reads the client's dataset, executes PSI with the server, and writes the output. This implementation contributes a scaled-down version: a single-execution client that spins up a one-time-use container and does not persist a web server for multiple executions. There are two main disadvantages to this approach: first, it doesn't take advantage of shared pre-processing data that can be reused across multiple executions of BaRK between two parties. Second, it requires the client to re-enter all their setup information each time they want to execute PSI (including, for example, the connection information for their Azure data and connection information for the server).

The deployment uses a wide variety of Azure resources and services, representative of the type of infrastructure required to deploy MPC in practice. The server infrastructure uses Kubernetes to maintain a load balancer and containers for each client connection. The client uses Azure Containers to initiate and execute the computation. Both parties use Cosmos DB to specify and hold their input sets, and the client also specifies a second table to write the computation output. Seclūd maintains a storage account with the container images (specified using Docker), which specify the computation (including reading input data, executing BaRK, and writing the output if appropriate). Each of these resources requires configuration of appropriate permissions and access keys to deploy correctly.

6.3 Deployment observations

Team motivation. This work is sponsored by the Microsoft Research Cryptography and Privacy group. This group works in an intersection between academia and industry: they develop and implement new cryptographic protocols like an academic group, but they maintain their software to the relatively high standard of industry. As a group within Microsoft, they have access to product teams and other potential “customers” within the company, who can identify internal applications for secure computation products.

Software quality and tests. The team has a significant focus producing high-quality software. New software for the repository is divided into pull requests, each implementing a specific feature, improvement, or bug fix. Before merging into the main repository, each pull request goes through a code review process, where 2-4 developers review and comment on the changes. These reviews enforce a high level of documentation, correct various instances of code diverging from best practices, and prevent software bugs. Indirectly, they ensure that multiple people are familiar with each part of the codebase, which improves maintainability.

The framework features a growing library of tests, including small-scale unit tests and end-to-end integration tests. Unit tests mainly apply to the supporting software, such as message serializing, data structures, and other components with clear specifications. Protocols implemented in Cocoon use end-to-end tests to ensure correct behavior on random inputs. The test suite is included in the repository’s continuous integration infrastructure. A pull request must pass the full suite of tests on multiple operating systems before it can be merged to prevent accidental breaking changes.

Security and infrastructure limitations. The Cocoon framework alone requires a fairly high degree of cryptographic knowledge to use because the user must specify a full set of parameters for each protocol and sub-protocol used. During this project, the protocols implemented in Cocoon had neither default settings for their parameters nor internal consistency checks.

For example, the BaRK PSI protocol defines a security parameter which is a function of parameters of the data structure used in the protocol, but the implementation does not verify that these (separately defined) parameters agree. The PSI protocol calls sub-protocols with their own security parameters, but does not verify that the sub-protocol security parameters are at least as large as the PSI security parameter. The PSI protocol defines the data structure parameters and the size of each party’s input, but does not verify that the structure is large enough to hold the input.

There are also no “secure defaults”, which could minimize the amount of external mistakes that can be made, e.g. by automatically setting the data structure to be larger than the input sizes.

In practice, many of these issues can be solved by adding more robust validation in the implementation, especially where two protocols come together (e.g. in input classes and protocol initialization functions). This is not an issue with the framework itself, but with the protocol implementations. In fact, the framework architecture includes initialization functions for each protocol where the protocol designer can add such consistency checks. In the current implementation, most of the parameters are fixed (team members manually validated their consistency in code review). The only user-provided parameters are the input set sizes which we use to customize the data structure.

Cocoon is not optimized for rounds of communication which might be an issue in low-bandwidth settings. In most cases, Cocoon only has one or two extra rounds of communication to correctly handle protocol terminations.

Documentation. Despite being a mainly internal tool, the Cocoon framework has a reasonable amount of documentation. There is a wiki page that describes the architecture of the framework and details the purpose of each component.

There is a large body of example code, mostly in the form of "hello world" protocols. These don't do any interesting computation but demonstrate the overall setup of a protocol: defining inputs, outputs, and their recipients; sending messages between parties; initializing state; and calling sub-protocols.

These examples are documented to various degrees; many include extended software documentation. The wiki explains each protocol at a high level and includes pages on constructing a new protocol from scratch, describing the various components.

Happily, new software added to the repository, including the BaRK and BaRK-OPRF protocols, is held to a higher standard of documentation, including documentation of all public methods and data structures. The software documentation uses XML comments that integrate with robust code editors and integrated development environments (IDEs), and can be processed by the C# compiler to produce a documentation file.

6.4 Conclusion

The Cocoon framework and Seclūd integration demonstrate the type of work required to make a software tool suitable for a production environment. Development on the Cocoon framework includes a significant focus on code quality and manual review, documentation, and automated testing. The

Seclūd integration focuses primarily on automatically provisioning the computing infrastructure required to execute secure computation, and illustrates the complexity of a robust deployment and the variety of necessary resources. Further work is required to tailor these resources to the specific computation being executed.

CHAPTER 7

Conclusion

This dissertation aims to illuminate barriers to the practical deployment of secure multi-party computation. The work in the previous chapters identifies specific issues and presents solutions in the context of several novel applications. As a technology, MPC continues to move rapidly toward practicality. This dissertation concludes with several recommendations for future work, encompassing both high-level research questions and practical engineering improvements.

One major area for improvement is in compiler design and optimization. The end-to-end frameworks in Chapter 3 often focus on either optimizing the MPC protocol implementation or on correctness, making guarantees about the equivalence of the described and compiled programs. For instance, in Chapter 5, EMP-toolkit [WMK16] provides a fast garbled circuit implementation, but generates SHA-256 circuits that are larger than existing optimized versions. The independent circuit compilers focus on optimizing intermediate representations, but require additional translation into circuit formats compatible with the best protocol implementations. Generating better translation tools or standardized circuit formats could easily improve overall efficiency of MPC executions by connecting the smallest circuits with the fastest protocols.

In fact, the utility of optimized circuit compilers extends beyond MPC deployments. Other privacy-preserving techniques such as homomorphic encryption and ZK proof systems also depend on compact, data-independent representations of general-purpose functions, although they may have different constraints for optimal performance. A valuable area for future work is to distill the relative requirements of each technique into a single circuit compiler that can optimize for each of these systems. Such a compiler would be particularly useful if users could add new primitives or constraints, to support the development of hybrid MPC frameworks (as defined in Chapter 3). The compiler and hardware communities likely have valuable insight to offer the developer of such a constraint solver.

The positive response to Chapter 3 suggests a need for continued maintenance of these resources. Indeed, at least eight frameworks have been introduced since the study concluded in mid-2018. Some have been added to the open-source repository by this author or the original framework designers, but a systematic analysis of recent improvements would be a continued benefit to the community.

As MPC techniques gain popularity in practical settings, companies will expect standardized protocols and implementations. Few, if any, of the existing frameworks are interoperable: two parties wishing to execute an MPC computation must use the same software. The eventual convergence on a standard set of MPC protocols, preprocessing techniques, and optimized communication architectures will fall on a body like NIST [BDV20]. However, it does not appear that the community has begun to select for any specific type of protocol. In fact, there is not even a clear selection for basic primitives, such as replicated secret-sharing, oblivious transfer, or oblivious transfer extension, that underpin the security of many MPC protocols.

Practical use of MPC frameworks will require high-quality software that is resistant to misuse. The proof-of-concept frameworks used throughout this dissertation illustrate common pitfalls in implementations. Future engineering work should focus on minimizing these, and related, issues.

Computation on real numbers remains difficult. Many MPC frameworks implement some real number operations, typically using a fixed-point representation. However, their functionality remains limited, and the increased complexity of such operations present additional usability issues. For example, many techniques require some amount of "extra space" in a fixed-point variable (relative to the actual numerical value) to correctly evaluate without truncation or overflow. This was an issue in the network algorithms in Chapter 4. This type of overhead should be explicitly documented or handled entirely by the MPC engine.

Many potential users of MPC frameworks do not have a strong grounding in the underlying primitives and techniques. Thus, frameworks designed for a general-purpose audience should aim for simplicity that make it difficult for a user to accidentally make insecure and inefficient choices. For example, Chapter 6 introduced a discussion on secure defaults. A production-quality framework that offers options for multiple protocols should validate that each protocol is instantiated at an equivalent security level and notify the user of misaligned parameter settings. Similarly, MPC compilers should document thoroughly—or avoid entirely—high-level language features that offer similar operations with different performance characteristics. For example, early versions of SCALE-MAMBA [AKR+19] feature multiple for-loop constructions, one of which is significantly more efficient (later versions introduce a new high-level language that removes this option).

Few practical applications exist in isolation or use MPC as a complete solution. MPC software frameworks must be integrable into existing software systems. Frameworks that design domain-specific languages should include robust systems for receiving input and writing output data for the computation. The most straightforward method is writing to and from files; this is typically adaptable to read and write from standard input. Framework designers must take care to specify exactly how such I/O systems operate. Little is more frustrating to a user than finding that their little-endian data was interpreted as big-endian. Indeed, the implementation in Chapter 5 required a translation layer to break complex types into correctly-oriented bitwise representations and assign them to the corresponding wires in the circuit. Chapter 3 describes a variety of other pitfalls in I/O manipulation. Although this topic is not particularly novel in a pure academic sense, it is essential to an effective deployment.

Although significant work lies ahead, this author remains optimistic about the future of secure computation. With growing awareness of MPC comes increased corporate investment, higher quality engineering, and increased interest in standardization. These improvements, along with the contributions of this dissertation, will continue to reduce the "great expertise" required to successfully deploy secure computation.

BIBLIOGRAPHY

[ABL$^+$04] Mikhail Atallah, Marina Bykova, Jiangtao Li, Keith Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 103–114, New York, NY, USA, 2004. ACM.

[ABL$^+$18] David W. Archer, Dan Bogdanov, Y. Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases – real-world applications of secure multi-party computation. Cryptology ePrint Archive, Report 2018/450, 2018. `https://eprint.iacr.org/2018/450`.

[ABPP15] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. Cryptology ePrint Archive, Report 2015/1039, 2015. `http://eprint.iacr.org/2015/1039`.

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014 Workshops*, volume 8438 of *Lecture Notes in Computer Science*, pages 105–121. Springer, Heidelberg, March 2014.

[AG98] Franklin Allen and Douglas Gale. Optimal Financial Crises. *The Journal of Finance*, 53(4):1245–1284, August 1998.

[AG00] Franklin Allen and Douglas Gale. Financial Contagion. *Journal of Political Economy*, 108(1):1+, February 2000.

[AGH$^+$21] J. Ayo Akinyele, Matthew Green, Marcella Hastings, Gabriel Kaptchuk, Ian Miers, Darius E. Parvin, Colleen M. Swanson, and Gijs Van Laer. zkChannels: Fast, unlinkable payments for any blockchain using 2PC. Technical report, Bolt Labs, Inc., 2021.

[AIK14] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[AKL12] Emmanuel A. Abbe, Amir E. Khandani, and Andrew W. Lo. Privacy-Preserving Methods for Sharing Financial Risk Exposures. *American Economic Review*, 102(3):65–70, May 2012.

[AKR$^+$19] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. Scale-mamba. https://homes.esat.kuleuven.be/ nsmart/SCALE/, 2019.

[Ale] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. `https://github.com/aicis/fresco`. Accessed 20 October 2019.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 535–548. ACM Press, November 2013.

[AOTS15] Daron Acemoglu, Asuman Ozdaglar, and Alireza Tahbaz-Salehi. Systemic risk and stability in financial networks. *American Economic Review*, 2015:564 – 608, 2015.

[ARS$^+$15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, Heidelberg, April 2015.

[Bar] libscapi. `https://github.com/cryptobiu/libscapi`. Developed by Bar Ilan University Cryptography Research Group. Accessed 25 June 2018.

[BBBC19] Bruno Biais, Christophe Bisiere, Matthieu Bouvard, and Catherine Casamatta. The blockchain folk theorem. *The Review of Financial Studies*, 32(5):1662–1715, 2019.

[BBC89] Francesco Brioschi, Luigi Buzzacchi, and Massimo G. Colombo. Risk capital financing and the separation of ownership and control in business groups. *Journal of Banking & Finance*, 13(4-5):747–772, September 1989.

[BBSdV19] Frank Blom, Niek J. Bouman, Berry Schoenmakers, and Niels de Vreede. Efficient secure ridge regression from randomized Gaussian elimination. Cryptology ePrint Archive, Report 2019/773, 2019. `https://eprint.iacr.org/2019/773`.

[BCD$^+$09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, Heidelberg, February 2009.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multiparty computation for (parallel) RAM programs. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 742–762. Springer, Heidelberg, August 2015.

[BD18] Sean Bowe and Jason Davies. Conclusion of the Powers of Tau ceremony. `https://www.zfnd.org/blog/conclusion-of-powers-of-tau/`, 2018. Accessed: 2020-12-01.

[BDK$^+$18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 847–861. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, Heidelberg, May 2011.

[BDST20] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION - A framework for mixed-protocol multi-party computation. Cryptology ePrint Archive, Report 2020/1137, 2020. `https://eprint.iacr.org/2020/1137`.

[BDV20] Luís T. A. N. Brandão, Michael Davidson, and Apostol Vassilev. NIST roadmap toward criteria for threshold schemes for cryptographic primitives. Technical Report NIST Internal or Interagency Report 8214A, National Institute of Standards and Technology, Gaithersburg, MD, 2020.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, Heidelberg, August 1992.

[Bei11] Amos Beimel. Secret-Sharing Schemes: A Survey. In YeowMeng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, volume 6639 of *Lecture Notes in Computer Science*, pages 11–46. Springer Berlin Heidelberg, 2011.

[BEK$^+$11] Lawrence E. Blume, David A. Easley, Jon M. Kleinberg, Robert Kleinberg, and Éva Tardos. Which networks are least susceptible to cascading failures? In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 393–402. IEEE Computer Society Press, October 2011.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. `http://eprint.iacr.org/2017/1050`.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, May 1988.

[BHKL18] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM*

CCS 2018: 25th Conference on Computer and Communications Security, pages 695–712. ACM Press, October 2018.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 784–796. ACM Press, October 2012.

[BI93] Michael Bertilsson and Ingemar Ingemarsson. A construction of practical secret sharing schemes using linear block codes. In Jennifer Seberry and Yuliang Zheng, editors, *Advances in Cryptology – AUSCRYPT'92*, volume 718 of *Lecture Notes in Computer Science*, pages 67–79. Springer, Heidelberg, December 1993.

[BJSV15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015: 19th International Conference on Financial Cryptography and Data Security*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234. Springer, Heidelberg, January 2015.

[BJSV16] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Technical report, Technical Report T-4-24. Cybernetica AS, http://research. cyber. ee, 2016.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 421–439. Springer, Heidelberg, August 2014.

[BKK$^+$16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings on Privacy Enhancing Technologies*, 2016(3):117–135, July 2016.

[BKL$^+$14] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-Preserving Statistical Data Analysis on Federated Databases. In Bart Preneel and Demosthenes Ikonomou, editors, *Privacy Technologies and Policy*, volume 8450 of *Lecture Notes in Computer Science*, pages 30–55. Springer International Publishing, 2014.

[BKLS14] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014. `http://eprint.iacr.org/2014/512`.

[BL90] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 27–35. Springer, Heidelberg, August 1990.

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. *Proceedings of AFIPS 1979 National Computer Conference*, 48:313–317, 1979.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 578–590. ACM Press, October 2016.

[BLV17] Azer Bestavros, Andrei Lapets, and Mayank Varia. User-centric distributed solutions for privacy-preserving analytics. *Communications of the ACM*, 60(2):37–39, 2017.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008:*

13th European Symposium on Research in Computer Security, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, Heidelberg, October 2008.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513. ACM Press, May 1990.

[BMT20a] Volodymyr Babich, Simone Marinesi, and Gerry Tsoukalas. Does crowdfunding benefit entrepreneurs and venture capital investors? *Manufacturing & Service Operations Management*, 2020.

[BMT20b] Elena Belavina, Simone Marinesi, and Gerry Tsoukalas. Rethinking crowdfunding platform design: mechanisms to deter misconduct and improve efficiency. *Management Science*, 2020.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008: 15th Conference on Computer and Communications Security*, pages 257–266. ACM Press, October 2008.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, September 2012.

[Bri90] Ernest F. Brickell. Some ideal secret sharing schemes. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology – EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 468–475. Springer, Heidelberg, April 1990.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security 2010: 19th USENIX Security Symposium*, pages 223–240. USENIX Association, August 2010.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *FC 2012: 16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer, Heidelberg, February / March 2012.

[CC06] Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 521–536. Springer, Heidelberg, August 2006.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19. ACM Press, May 1988.

[CDC$^+$16] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Towards securing internet exchange points against curious onlookers. In *ANRW*, pages 32–34, New York, NY, USA, 2016. ACM.

[CDC$^+$17] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. SIXPACK: Securing Internet eXchange Points Against Curious onlooKers. In *13. International Conference on emerging Networking EXperiments and Technologies (CoNEXT'17)*, pages 120–133. ACM, 2017.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, Heidelberg, May 2000.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. *Secure Multiparty Computation and Secret Sharing An Information Theoretic Approach.* Cambridge University Press, 2015.

[CDNN15] Martine de Cock, Rafael Dowsley, Anderson CA Nascimento, and Stacey C Newman. Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2015.

[CF02] Ronald Cramer and Serge Fehr. Optimal black-box secret sharing over arbitrary Abelian groups. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 272–287. Springer, Heidelberg, August 2002.

[CGG$^+$20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20: 27th Conference on Computer and Communications Security*, pages 1769–1787. ACM Press, November 2020.

[CGR$^+$17] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. Cryptology ePrint Archive, Report 2017/1109, 2017. `https://eprint.iacr.org/2017/1109`.

[CH15] Erika Check Hayden. Extreme cryptography paves way to personalized medicine. *Nature*, 519(7544):400–401, March 2015.

[Cha83] David Chaum. Blind signature system. In David Chaum, editor, *Advances in Cryptology – CRYPTO'83*, page 153. Plenum Press, New York, USA, 1983.

[CHK$^+$12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *Topics in Cryptology – CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer, Heidelberg, February / March 2012.

[CK19] Ning Cai and Steven Kou. Econometrics with privacy preservation. *Operations Research*, 67(4):905–926, 2019.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th*

International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, pages 168–176, 2004.

[CL20] Jiri Chod and Evgeny Lyandres. A theory of icos: Diversification, agency, and information asymmetry. *Management Science*, 2020.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th Annual ACM Symposium on Theory of Computing*, pages 364–369. ACM Press, May 1986.

[CLW20] Lin William Cong, Ye Li, and Neng Wang. Tokenomics: Dynamic adoption and valuation. Technical report, National Bureau of Economic Research, 2020.

[CMR] Brent Carmer, Alex J. Malozemoff, and Marc Rosen. swanky: A suite of rust libraries for secure multi-party computation. `https://github.com/GaloisInc/swanky`.

[Coi] Global charts — CoinMarketCap. `https://coinmarketcap.com/charts/`. Accessed: 2020-11-29.

[Cou18] Geoffroy Couteau. New protocols for secure equality test and comparison. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 303–320. Springer, Heidelberg, July 2018.

[Cré88] Claude Crépeau. Equivalence between two flavours of oblivious transfers. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 350–354. Springer, Heidelberg, August 1988.

[CS20] Soudipta Chakraborty and Robert Swinney. Signaling to the crowd: Private quality information and rewards-based crowdfunding. *Manufacturing & Service Operations Management*, 2020.

[CTT$^+$20] Jiri Chod, Nikolaos Trichakis, Gerry Tsoukalas, Henry Aspegren, and Mark Weber. On the financing benefits of supply chain transparency and blockchain adoption. *Management Science*, 2020.

[CWB18] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547, 2018.

[DA01a] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 273–282. IEEE, 2001.

[DA01b] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative statistical analysis. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, pages 102–110. IEEE, December 2001.

[DA01c] Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on new security paradigms*, NSPW '01, pages 13–22, New York, NY, USA, 2001. ACM.

[DAF] Kinan Dak Albab and Peter Flockhart. JIFF: JavaScript implementation of federated functionalities. `https://github.com/multiparty/jiff`. Boston University Software & Application Innovation Lab.

[Daf16] Suhas Daftuar. Mining: Select transactions using feerate-with-ancestors. Github `https://github.com/bitcoin/bitcoin/pull/7600`, 2016. Pull request for bitcoin repository by user sdaftuar.

[Dan15] Quynh H Dang. FIPS PUB 180-4: secure hash standard. Technical report, National Institute of Standards and Technology, 2015.

[DDK+15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 1504–1517. ACM Press, October 2015.

[DEs16] Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1602–1613. ACM Press, October 2016.

[DF10] Dodd-Frank. Dodd-Frank wall street reform and consumer protection act. Public Law 111-203, US Statutes at Large, 2010.

[DFA19] DFAST. Dodd-Frank Act Stress Tests (DFAST). `https://www.fhfa.gov/SupervisionRegulation/DoddFrankActStressTests`, 2019.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, Heidelberg, March 2006.

[DGK08] Ivan Damgard, Martin Geisler, and Mikkel Kroigard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, Heidelberg, March 2009.

[DGN$^+$17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 2263–2276. ACM Press, October / November 2017.

[DHC04] Wenliang Du, Yunghsiang S. Han, and Shigang Chen. Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification. In *In Proceedings of the 4th SIAM International Conference on Data Mining*, pages 222–233, 2004.

[DHSS17] Daniel Demmler, Kay Hamacher, Thomas Schneider, and Sebastian Stammler. Privacy-preserving whole-genome variant queries. In Srdjan Capkun and Sherman S. M. Chow, editors, *CANS 17: 16th International Conference on Cryptology and Network Security*, volume 11261 of *Lecture Notes in Computer Science*, pages 71–92. Springer, Heidelberg, November / December 2017.

[DKLs18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pages 980–997. IEEE Computer Society Press, May 2018.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, Heidelberg, August 2012.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 523–535. ACM Press, October / November 2017.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*. The Internet Society, February 2015.

[Dwo11] Cynthia Dwork. Differential privacy. *Encyclopedia of Cryptography and Security*, pages 338–340, 2011.

[Eco14] The Economist. Note to future self. *The Economist*, November 2014.

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012. `http://eprint.iacr.org/2012/629`.

[EGJ14] Matthew Elliott, Benjamin Golub, and Matthew O Jackson. Financial networks and contagion. *American Economic Review*, 104(10):3115–53, 2014.

[EGL82] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology – CRYPTO'82*, pages 205–210. Plenum Press, New York, USA, 1982.

[Els11] Helmut Elsinger. Financial networks, cross holdings, and limited liability. Technical Report 156, Oesterreichische Nationalbank, 2011.

[EN01] Larry Eisenberg and Thomas H. Noe. Systemic Risk in Financial Systems. *Management Science*, 47(2):236–249, February 2001.

[FHK$^+$14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer Berlin Heidelberg, 2014.

[FHT94] M. Fedenia, J. E. Hodder, and A. J. Triantis. Cross-holdings: estimation issues, biases, and distortions. *Review of Financial Studies*, 7(1):61–96, January 1994.

[FKOS13] Mark Flood, Jonathan Katz, Stephen Ong, and Adam Smith. Cryptography and the Economics of Supervisory Information: Balancing Transparency and Condentiality. Technical Report 0011, Office of Financial Research, September 2013.

[FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1651–1662, 2016.

[FP91] Kenneth R. French and James M. Poterba. Were Japanese stock prices too high? *Journal of Financial Economics*, 29(2):337–363, October 1991.

[FPR$^+$18] Zachary Feinstein, Weijie Pang, Birgit Rudloff, Eric Schaanning, Stephan Sturm, and Mackenzie Wildman. Sensitivity of the Eisenberg–Noe clearing vector to individual interbank liabilities. *SIAM Journal on Financial Mathematics*, 9(4):1286–1325, 2018.

[Gen96] Rosario Gennaro. *Theory and Practice of Verifiable Secret Sharing*. PhD thesis, MIT, 1996.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178. ACM Press, May / June 2009.

[GFAW17] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. Pretzel: Email encryption and provider-supplied functions are compatible. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 169–182. ACM, 2017.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 491–520. Springer, Heidelberg, October / November 2016.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, Heidelberg, May 2014.

[GHM12] C. Gouriéroux, J. C. Héam, and A. Monfort. Bilateral exposures and systemic solvency risk Expositions bilatérales et risque systémique pour la solvabilité . *Canadian Journal of Economics/Revue canadienne d'économique*, 45(4):1273–1309, November 2012.

[GK10] Prasanna Gai and Sujit Kapadia. Contagion in financial networks. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 466(2120):2401–2423, August 2010.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th Annual Symposium on Foundations of Computer Science*, pages 210–229. IEEE Computer Society Press, October 2015.

[GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 473–489. ACM Press, October / November 2017.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM Press, May 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[Gol] Steven Goldfeder. A boolean circuit for SHA-256. `http://stevengoldfeder.com/projects/circuits/sha2circuit.html`.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM Symposium Annual on Principles of Distributed Computing*, pages 101–111. Association for Computing Machinery, June / July 1998.

[GSB+17] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies*, 2017(4):345–364, October 2017.

[GTK+] Martin Geisler, Tomas Toft, Mikkel Krøigård, Thomas Pelle Jakobsen, Jakob Illeborg Pagter, Sigurd Meldgaard, Marcel Keller, Tord Reistad, Ivan Damgård, and Janus Dam Nielsen and. VIFF. `http://viff.dk`.

[GTN20] Rowena J Gan, Gerry Tsoukalas, and Serguei Netessine. Initial coin offerings, speculators, and asset tokenization. *Management Science*, 2020.

[HAB+17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society, February / March 2017.

[HCB18] Brian Hie, Hyunghoon Cho, and Bonnie Berger. Realizing private and practical pharmacological collaboration. *Science*, 362(6412):347–350, 2018.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 772–783. ACM Press, October 2012.

[HHFT20] Marcella Hastings, Brett Hemenway Falk, and Gerry Tsoukalas. Privacy-preserving network analytics. SSRN, 2020.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

[HIKN08] Danny Harnik, Yuval Ishai, Eyal Kushilevitz, and Jesper Buus Nielsen. OT-combiners via secure computation. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 393–411. Springer, Heidelberg, March 2008.

[HJS19] Franz J Hinzen, Kose John, and Fahad Saleh. Bitcoin's fatal flaw: The limited adoption problem. *NYU Stern School of Business*, 2019.

[HK16] Brett Hemenway and Sanjeev Khanna. Sensitivity and computational complexity in financial networks. *Algorithmic Finance*, 5(3-4):95–110, 2016.

[HLOI16] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16: 10th International Conference on Security in Communication Networks*, volume 9841 of *Lecture Notes in Computer Science*, pages 169–187. Springer, Heidelberg, August / September 2016.

[HM00] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, January 2000.

[HOS18] Shuichi Hirahara, Igor Carboni Oliveira, and Rahul Santhanam. Np-hardness of minimum circuit size problem for or-and-mod circuits. ECC TR-18-030, 2018.

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, Heidelberg, August 2014.

[HS15] Shai Halevi and Victor Shoup. HElib . `https://github.com/homenc/HElib/`, 2015.

[HWB14] Brett Hemenway, William Welser, and Dave Baiocchi. Achieving Higher-Fidelity Conjunction Analyses Using Cryptography to Improve Information Sharing. Technical report, RAND Corporation, 2014.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, Heidelberg, August 2003.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, Heidelberg, August 2008.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 294–314. Springer, Heidelberg, March 2009.

[JKS08] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society Press, May 2008.

[JLE17] Bargav Jayaraman, Haina Li, and David Evans. Decentralized certificate authorities. arXiv preprint arXiv:1706.03370, 2017.

[Joh82] Charles R. Johnson. Inverse M-matrices. *Linear Algebra and its Applications*, 47:195–216, October 1982.

[JP19] Matthew O Jackson and Agathe Pernoud. What makes financial networks special? distorted investment incentives, regulation, and systemic risk measurement. SSRN, 2019.

[JWB$^+$17] Karthik A. Jagadeesh, David J. Wu, Johannes A. Birgmeier, Dan Boneh, and Gill Bejerano. Deriving genomic diagnoses without revealing patient genomes. *Science*, 357(6352):692–695, 2017.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 30–41. ACM Press, November 2014.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[KBLV13] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, April 2013.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20: 27th Conference on Computer and Communications Security*, pages 1575–1590. ACM Press, November 2020.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31. ACM Press, May 1988.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 818–829. ACM Press, October 2016.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* CRC press, 2014.

[KLS$^+$17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017(4):177–197, October 2017.

[KLSR09] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Privacy-preserving analysis of vertically partitioned data using secure matrix products. *Journal of Official Statistics*, 25(1):125, 2009.

[KMW16] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. `http://eprint.iacr.org/2016/184`.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, Heidelberg, August 2015.

[Kra21] Kraken. Cryptocurrency deposit processing times. `https://support.kraken.com/hc/en-us/articles/203325283-Cryptocurrency-deposit-processing-times`, 2021.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, Heidelberg, July 2008.

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *USENIX Security 2012: 21st USENIX Security Symposium*, pages 285–300. USENIX Association, August 2012.

[KSS13] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *ISOC Network and Distributed System Security Symposium – NDSS 2013*. The Internet Society, February 2013.

[KW14] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.

[Lar15] Enrique Larraia. Extending oblivious transfer efficiently - or - how to get active security with constant cryptographic overhead. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014: 3rd International Conference on Cryptology and Information Security in Latin America*, volume 8895 of *Lecture Notes in Computer Science*, pages 368–386. Springer, Heidelberg, September 2015.

[LDAI$^+$19] Andrei Lapets, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, Azer Bestavros, and Frederick Jansen. Role-based ecosystem for the design, development, and deployment of secure multi-party data analytics applications. In *Proceedings of the IEEE Secure Development Conference, SecDev*, September 2019.

[LHS$^+$14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *2014 IEEE Symposium on Security and Privacy*, pages 623–638. IEEE Computer Society Press, May 2014.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 1–17. Springer, Heidelberg, August 2013.

[Lin16] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. `http://eprint.iacr.org/2016/046`.

[Lin17] Yehuda Lindell. Fast secure two-party ECDSA signing. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 613–644. Springer, Heidelberg, August 2017.

[Lin21] Yehuda Lindell. Secure multiparty computation. *Communications of the Association for Computing Machinery*, 64(1):86–96, January 2021.

[LJDA+18] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1837–1854. ACM Press, October 2018.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, Heidelberg, May 2013.

[LP09a] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[LP09b] Yehuda Lindell and Benny Pinkas. Secure Multiparty Computation for Privacy-Preserving Data Mining. *The Journal of Privacy and Confidentiality*, 1(1):59–98, 2009.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338. Springer, Heidelberg, August 2015.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 476–494. Springer, Heidelberg, August 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 579–590. ACM Press, October 2015.

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 554–581. Springer, Heidelberg, October / November 2016.

[LWN$^+$15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[MGC$^+$16] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 112–127, March 2016.

[MMK$^+$17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 455–471. ACM Press, October / November 2017.

[MMS$^+$19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*. The Internet Society, February 2019.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004: 13th USENIX Security Symposium*, pages 287–302. USENIX Association, August 2004.

[Mon] Monero (XMR). `https://www.getmonero.org/`. Accessed: 2020-11-24.

[Mor00] Stephen Morris. Contagion. *Review of Economic Studies*, 67(1):57–78, 2000.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52. ACM Press, October 2018.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 591–602. ACM Press, October 2015.

[MSBL$^+$20] Pedro Moreno-Sanchez, Arthur Blue, Duc V. Le, Sarang Noether, Brandon Goodell, and Aniket Kate. DLSAG: Non-interactive Refund Transactions for Interoperable Payment Channels in Monero. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 325–345, Cham, 2020. Springer International Publishing.

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[Nak19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[NMRL16] Shen Noether, Adam Mackenzie, and the Monero Research Lab. Ring confidential transactions. *Ledger*, 1:1–18, Dec. 2016.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh

Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, Heidelberg, August 2012.

[NP06] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 35(5):1254–1281, 2006.

[NPH14] Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen. Compute globally, act locally: Protecting federated systems from systemic threats. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, 2014.

[NWI$^+$13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*, pages 334–348. IEEE Computer Society Press, May 2013.

[PBS12] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. Cybernetica Research Reports T-4-17, 2012.

[PCCX09] Ignacio Cascudo Pueyo, Hao Chen, Ronald Cramer, and Chaoping Xing. Asymptotically good ideal linear secret sharing with strong multiplication over any fixed finite field. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 466–486. Springer, Heidelberg, August 2009.

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments, 2016.

[PGFW] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N Wright. Ssc protocol comparison tool. `https://code.google.com/archive/p/syssc-ui/`. Accessed 2018-06-25.

[PGFW14] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N. Wright. Systematizing secure computation for research and decision support. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14: 9th International Conference on Security in Communication*

Networks, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer, Heidelberg, September 2014.

[Por13] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, August 2013.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 515–530. USENIX Association, August 2015.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, Heidelberg, April / May 2018.

[Rab81] Michael Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, 1981.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE Computer Society Press, May 2014.

[Rot18] Dragoș Rotaru. awesome-mpc. `https://github.com/rdragos/awesome-mpc`, 2018.

[RQA$^+$18] Anjana Rajan, Lucy Qin, David W. Archer, Dan Boneh, Tancrède Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In Ellen W. Zegura, editor, *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS 2018, Menlo Park and San Jose, CA, USA, June 20-22, 2018*, pages 49:1–49:4. ACM, 2018.

[RR16] Kurt Rohloff and Gerard Ryan. The PALISADE lattice cryptography library. `https://git.njit.edu/palisade/PALISADE`, 2016.

[RS20] Ioanid Rosu and Fahad Saleh. Evolution of shares in a proof-of-stake cryptocurrency. *Management Science*, 2020.

[RSH17] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations. arXiv preprint arXiv:1711.06467, 2017.

[SCG$^+$14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[Sch] Berry Schoenmakers. MPyC: Secure multiparty computation in python. `https://github.com/lschoe/mpyc`. Accessed 30 October 2019.

[SEA20] Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL`, November 2020. Microsoft Research, Redmond, WA.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SHS$^+$15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[SPD18] SPDZ. Spdz discussion group. `https://groups.google.com/forum/#!forum/spdz`, 2018. Accessed 2018-05-24.

[SRBW18] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. Practical secure computation outsourcing: A survey. *ACM Computing Surveys (CSUR)*, 51(2):31, 2018.

[ST] Nigel Smart and Stefan Tillich. (Bristol format) circuits of basic functions suitable for MPC and FHE. `https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html`.

[SvA$^+$19] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. In Ian Goldberg and Tyler Moore, editors, *FC 2019: 23rd International Conference on Financial Cryptography and Data Security*, volume 11598 of *Lecture Notes in Computer Science*, pages 605–623. Springer, Heidelberg, February 2019.

[SVR$^+$20] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High throughput cryptocurrency routing in payment channel networks. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 777–796, 2020.

[SZ19] Gustavo Schwenkler and Hannan Zheng. The network of firms implied by the news. Available at SSRN 3320859, 2019.

[Tez] Adding support for the pairing-equipped elliptic curve bls12–381 to tezos. `https://bit.ly/3lYvLs5`. Accessed: 2020-11-24.

[TF20] Gerry Tsoukalas and Brett Hemenway Falk. Token-weighted crowdsourcing. *Management Science*, 2020.

[TJGE16] Lu Tian, Bargav Jayaraman, Quanquan Gu, and David Evans. Aggregating private sparse learning models using multi-party computation. In *NIPS Workshop on Private Multi-Party Machine Learning, Barcelona, Spain*, 2016.

[TMB19] Gerry Tsoukalas, Simone Marinesi, and Volodymyr Babich. Updating the crowdfunding narrative. *Wharton Public Policy Initiative*, 2019.

[TMM19] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^{2}L: Anonymous atomic locks for scalability and interoperability in payment channel hubs. Cryptology ePrint Archive, Report 2019/589, 2019. `https://eprint.iacr.org/2019/589`.

[TMSS20] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. Paymo: Payment channels for monero. Cryptology ePrint Archive, Report 2020/1441, 2020. `https://eprint.iacr.org/2020/1441`.

[VD97] Marten Van Dijk. *Secret key sharing and secret key generation.* PhD thesis, Eindhoven University of Technology, 1997.

[VJH20] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020.

[Wie83] Stephen Wiesner. Conjugate coding. *SIGACT News*, 15(1):78–88, January 1983.

[Wil77] R. A. Willoughby. The inverse M-matrix problem. *Linear Algebra and its Applications*, 18(1):75–94, January 1977.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[WMK17] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 399–424. Springer, Heidelberg, April / May 2017.

[WNR] Pieter Wuille, Jonas Nick, and Tim Ruffing. Bip 0340. `https://en.bitcoin.it/wiki/BIP_0340`. Accessed: 2020-12-02.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 21–37. ACM Press, October / November 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 39–56. ACM Press, October / November 2017.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society Press, November 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986.

[ZBA17] Yihua Zhang, Marina Blanton, and Ghada Almashaqbeh. Implementing support for pointers to private data in a general-purpose secure multi-party compiler. *ACM Trans. Priv. Secur.*, 21(2):6:1–6:34, December 2017.

[Zca] Zcash NU5 proposals. `https://github.com/zcash/zips/labels/NU5%20proposal`. Accessed: 2020-12-02.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. `http://eprint.iacr.org/2015/1153`.

[Zel14] Nicholette Zeliadt. Cryptographic methods enable analyses without privacy breaches. *Nature Medicine*, 20(6):563, June 2014.

[ZIP] Zip 222: Transparent Zcash extensions. `https://zips.z.cash/zip-0222`. Accessed: 2020-11-24.

[ZLL$^+$18] Yuncong Zhang, Yu Long, Zhen Liu, Zhiqiang Liu, and Dawu Gu. Z-channel: Scalable and efficient scheme in zerocash. In Willy Susilo and Guomin Yang, editors, *ACISP 18: 23rd Australasian Conference on Information Security and Privacy*, volume 10946 of *Lecture Notes in Computer Science*, pages 687–705. Springer, Heidelberg, July 2018.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, Heidelberg, April 2015.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 813–826. ACM Press, November 2013.

[ZWR+16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234. IEEE Computer Society Press, May 2016.