Implementation and Analysis of an Active-Security Compiler for Passively Secure Multi-Party Computation Protocols

July 2024

A thesis submitted by

Lorenzo Rota

to obtain the degree of Master of Science
at the University of Groningen.

Supervisors:
dr. F. Turkmen (UG, First Supervisor)
T.H. Bontekoe, M.Sc. (UG, Second Supervisor)
dr. ir. P.N. Langenkamp (TNO, Daily Supervisor)
V.A. Dunning, M.Sc. (TNO, External Supervisor)

University of Groningen (UG):
Faculty of Science and Engineering
Information Systems
TNO:
Applied Cryptography & Quantum Algorithms

Abstract

Secure multi-party computation (MPC) allows joint computation over private data via cryptographic protocols, typically designed to withstand passive and active adversaries. Passive adversaries follow the protocol honestly but try to learn private information from the messages they receive, while active adversaries may deviate from the protocol to pursue malicious goals. Although many efficient passively secure protocols exist, their security is deemed insufficient for real-world applications, necessitating procedures to achieve active security. Such a procedure was first proven possible for all instances of MPC through the use of Zero-Knowledge Proofs (ZKPs) via the GMW compiler (Goldreich et al., STOC '87). While conceptually powerful and simple, its usage, particularly for general-purpose protocols, has been largely disregarded due to its reliance on inefficient ZKPs.

Recent advancements in practical non-interactive ZKPs, paired with the widespread presence of passively secure special-purpose protocols, motivates the revival of the GMW compiler. In this thesis, we design a versatile compiler compatible with plug-and-play ZKP systems, achieving active security with abort in a dishonest majority setting. It is suited for multi-party protocols in the synchronous broadcast-only model and supports initial point-to-point communication, making it appropriate for protocols constructed from (threshold) homomorphic encryption and secret sharing. For n-party protocols requiring $O(n)$ random field elements per party, our analysis shows that the compiler introduces a runtime overhead of $O(n\,\mathrm{poly}(n))$ and increases communication by $O(n^3)$ elements.

We demonstrate feasibility through a proof-of-concept implementation for protocols programmed in a modest subset of Python 3.10, encompassing the majority of frequently used language constructs. Using the Groth16 (Groth, EUROCRYPT 2016) and Bulletproofs (Bünz et al., S&P 2018) ZKP schemes, we evaluate the compiler's performance on a multi-party sum protocol based on additive secret sharing, obtaining benchmarks indicative of practical utility for a small number of parties. Specifically, for three parties and excluding setup, the slowest runtime is approximately 46 seconds, with maximum communication totaling 20 kilobytes.

Contents

Acronyms

ASDL Abstract Syntax Description Language.

ASIC Application-Specific Integrated Circuit.

AST Abstract Syntax Tree.

BNF Backus-Naur Form.

CLI Command-Line Interface.

CRS Common Reference String.

DL Discrete Logarithm.

DSL Domain-Specific Language.

EQC Existentially Quantified Circuit.

ETDP Enhanced Trapdoor Permutation.

GC Garbled Circuit.

GG Generic Group.

ID-MPC Identifiable MPC.

IR Intermediate Representation.

IT Information Theoretic.

LWE Learning With Errors.

MAC Message Authentication Code.

MPC Multi-Party Computation.

MPCitH MPC-in-the-head.

NIZK Non-Interactive Zero-Knowledge.

NP Nondeterministic Polynomial.

OT Oblivious Transfer.

OWF One-Way Function.

PA-MPC Publicly Auditable MPC.

PKE Public-Key Encryption.

PKI Public-Key Infrastructure.

PoK Proof of Knowledge.

PPT Probabilistic Polynomial-Time.

QAP Quadratic Arithmetic Program.

R1CS Rank-1 Constraint System.

RAM Random-Access Machine.

RISC Reduced Instruction Set Computer.

RO Random Oracle.

SHE Somewhat Homomorphic Encryption.

SMT Satisfiability Modulo Theories.

SNARK Succinct Non-Interactive Argument of Knowledge.

STARK Scalable Transparent Non-Interactive Argument of Knowledge.

TSS Threshold Secret Sharing.

UC Universal Composability.

UML Unified Modeling Language.

ZK Zero-Knowledge.

ZKP Zero-Knowledge Proof.

ZKPoK Zero-Knowledge Proof of Knowledge.

ZKPoPK Zero-Knowledge Proof of Plaintext Knowledge.

ZKPyC Zero-Knowledge Python Compiler.

ZKPyToolkit Zero-Knowledge Python Toolkit.

List of Symbols

$\perp$ Signal to abort.

λ Security parameter.

p Prime number.

pk Public key.

sk Secret key.

pp Public parameters.

$R_{\square}$ Named NP-relation / ZK-statement.

$P_{\square}$ Numbered party.

$\Pi_{\square}$ Named cryptographic protocol.

$\mathcal{F}_{\square}$ Named cryptographic functionality.

1 Introduction

1.1 Actively Secure Multi-Party Computation

Secure *multi-party computation* (MPC) addresses the problem of enabling mutually distrusting parties to collaboratively compute a function while ensuring the privacy of their inputs. This dilemma dates back to the early 80s and was motivated by Yao's "Millionaire's Problem" [Yao82], in which two distrusting parties wish to learn who of the two is wealthier without either party directly revealing any information about their wealth. A cryptographic solution to this problem was later proposed in [Yao86], which introduced the machinery for solving general 2-party computation problems through the usage of *garbled circuits* (GCs). Since its inception, many works paved the way for generalized secure *multi*-party computation, such as the existence of MPC protocols over insecure communication channels based on *computational hardness* assumptions [GMW87] and the existence of protocols over secure communication channels that do not directly require such assumptions [BGW88, CCD88].

Over the past decades, MPC has flourished into a mature field of cryptography, studying protocols under various security and communication settings. The two most widely studied security models for MPC protocols are *passive* security and *active* security, commonly referred to as semi-honest and malicious security, respectively. Loosely speaking, the two models are defined as follows: The passive security model concerns an adversary who follows the protocol in an honest manner but attempts to illegitimately learn information from the messages it receives during the protocol execution. The active security model concerns an adversary who deviates from the protocol in an attempt to illegitimately learn information from, deny access to or corrupt information used by other parties.

When it comes to designing MPC protocols, there are generally two approaches:

1. A function or MPC problem is described as a boolean or arithmetic circuit, and is consumed by a general-purpose passively secure protocol such as [Yao86, GMW87, BGW88], or an actively secure protocol such as [CDN01, DPSZ12].

2. A special-purpose protocol is designed as a composition of simpler and well-understood cryptographic *primitives*, which are regarded as building blocks. Such protocols are typically based on the following three primitives: *oblivious transfer* (OT), *secret sharing* and *homomorphic encryption*.

In recent years, MPC protocols have shifted from once solely theoretical constructs to practical solutions for real-world problems. Notable examples are their deployment in digital market auctions [BCD+09], multi-party neural network training [WGC19] and money laundering detection [vDv+24]. Even though their corresponding MPC-based solutions demonstrate both feasibility and efficiency, they have the following aspects in common:

1. The secure protocols that are utilized in the MPC protocol are special-purpose and tailored specifically to the problem it aims to solve.

2. The MPC protocol offers full *passive* security, as opposed to *active* security.

These characteristics are not necessarily considered to be problematic, and in fact, even prove to be beneficial under the right circumstances. For example, special-purpose protocols can be more practical[1] than solutions that rely on general-purpose protocols. An example of such a protocol is a *mixed protocol* [RW19] that combines GC and secret sharing. Moreover, a passively secure MPC solution has a legitimate use-case when data confidentiality is more important than data integrity, and the parties can be trusted to follow the protocol when they interact.

Despite the rapid transition from theoretical to practical MPC marking an encouraging trajectory, the continued reliance on passively secure protocols impedes their widespread adoption due to the presence of malicious actors in the real world. Indeed, there are valid grounds for designing passively secure protocols over actively secure ones. Constructing a passively secure protocol from scratch is generally less complicated and costly as there are fewer security requirements to take into consideration. Although there exists a trade-off between security and performance that cannot be circumvented, it can be minimized. Additionally, it is desirable to start with an already existing passively secure protocol and make modifications such that it becomes actively secure. A procedure that achieves this is called a *passive-to-active compiler*, hereafter referred to as active-security compiler.

In this thesis, we broach the overarching question:

> *How can a practical active-security compiler be constructed for special-purpose passively secure multi-party computation protocols?*

To answer this, we outline and justify supplementary research questions in the following section.

1.2 Research Questions

The goal is to refine the scope of the overarching question and derive research questions that align with one of the active-security compiler paradigms described in Table 2.1, namely, the GMW [GMW87], IPS [IPS08], and BDOZ-SPDZ [BDOZ11, DPSZ12] paradigm. To understand their properties and constructions, the reader is advised to read the literature review in Chapter 2 before proceeding.

To begin, we examine the key challenges of obtaining a compiler for special-purpose protocols within each of the three paradigms. In the GMW paradigm, the key challenge is obtaining an efficient zero-knowledge proof to validate the correctness of a message from any arbitrary protocol round. Meanwhile, in the IPS paradigm, the key challenge is obtaining appropriate inner and outer protocols from any given passively secure protocol. Lastly, in the BDOZ-SPDZ paradigm, compilers are designed exclusively for general-purpose protocols, thus restricting their usage to mixed protocols that involve the compiled general-purpose protocol.

Considering our goal of constructing a *practical* compiler that can transform any *special*-purpose protocol, we find that the BDOZ-SPDZ paradigm is unsuitable for this pursuit. Moreover, while the IPS paradigm seems advantageous due to its black-box reliance of the passively secure protocol, the need to obtain both inner and outer protocols from the special-purpose protocol's code warrants an additional non-black-box compiler. This leaves us with the GMW paradigm, which, due to its simplicity, can theoretically be applied to any

[1] Practical solutions often mix and match primitives to improve performance and deviate from one-fits-all solutions.

passively secure protocol in relatively few steps. Given that, under standard computational assumptions, zero-knowledge proof computations are proportional to the statement they intend to prove, it is reasonable to explore what types of proof systems could be practical for large statements. Additionally, it is important to consider how these proofs can be automatically constructed for any passively secure protocol.

In alignment with the GMW paradigm, this thesis aims to advance the development of practical active-security compilers that are capable of compiling special-purpose MPC protocols. Specifically, the compiler should transform any passively secure protocol, regardless of its construction, into one that is actively secure with abort, while retaining most of its original properties. This development will be explored through the lenses of theory and practice, and will be addressed by the following research questions:

RQ1: *How can zero-knowledge proof systems be used to obtain active security from a passively secure protocol?*

RQ2: *How can we build an active-security compiler with minimal assumptions, capable of transforming a passively secure protocol, independent of its construction?*

RQ3: *How practical is the active-security compiler, and how can it be further improved?*

1.3 Contributions

This thesis aims to bridge the gap between the theory of active-security compilers for special-purpose protocols, and the practice of applying them to real-world implementations. The contributions are the following:

1. We carry out a literature review to identify the *three major active-security compiler paradigms*, namely, GMW, IPS and BDOZ-SPDZ. Then, we contextualize relevant (state-of-the-art) works into a *taxonomy* according to their respective paradigm and accompanied by a classification of essential properties and desirable features. Specifically, we classify whether the resulting protocols are secure with identifiable abort and public auditability, along with properties like threshold structure, adversarial power, the class of protocols they apply to, and the necessary cryptographic models of computation.

2. We design a *conceptual compiler following the GMW paradigm* that applies to MPC protocols for $n \geq 3$ parties, realized in *two synchronous communication models*: those that involve initial private communication via point-to-point channels, and those that solely communicate over a public broadcast channel. Moreover, our compiler inherits many of the nice properties from its ancestor, e.g., it provides *active security with identifiable abort* and admits a corruption threshold of $t < n$ parties.

3. We provide *actively secure constructions of the compiler's sub-protocols*, namely, the multi-party coin-tossing, multi-party commitment, and non-interactive zero-knowledge proof of knowledge protocols. Specifically, we construct efficient actively secure commitment and coin-tossing protocols from the Pedersen commitment scheme, and admit any non-interactive zero-knowledge protocol in the common-reference string or random oracle model.

4. We propose a *software design for implementing our active-security compiler*. Specifically, the compiler extends a passively secure protocol runtime environment by *automatically generating the required zero-knowledge proofs* and carrying out the necessary compilation steps. The "engine" that handles zero-knowledge proofs allows for a

drop-in replacement of any non-interactive zero-knowledge proof system.

5. We implement a *proof-of-concept compiler for protocols written in a strict subset of Python 3.10 and above*. For this, we develop a *zero-knowledge proof compiler* using the CirC circuit compiler infrastructure [OBW22]. Moreover, we develop a *zero-knowledge proof toolkit with built-in library* that provides the necessary compilable cryptographic building blocks. The proof-of-concept supports the Groth16 [Gro16] and Bulletproofs [BBB+18] proof systems, and can be extended to other systems via zkInterface [BGK+20].
6. We evaluate the *practicality of the active-security compiler* in two parts. First, we determine its *asymptotic efficiency* and empirically validate it via a *benchmark of our implementation*. Next, we explore its usability and utility through a *case study on a multi-party sum protocol*.

1.4 Thesis Structure

In Chapter 2, we conduct a literature review wherein we describe three active-security compiler paradigms, comparing and contrasting them based on their key properties. Following this comparison, we discuss other relevant works and categorize state-of-the-art compilers into a taxonomy. We provide essential background on cryptographic protocols in Chapter 3, covering topics such as zero-knowledge proofs and secure multi-party computation, thereby laying the groundwork relevant for answering the research questions.

The first question is addressed in Chapter 4, where we embark on a detailed, step-by-step construction of the GMW-style active-security compiler for special-purpose protocols. This chapter begins with a specification of the compiler's components and other important definitions. Subsequently, we present a theoretical construction of the overall compiler, followed by a discussion on the concrete utilization of generic zero-knowledge proof systems. In Chapter 5, we address the second question by devising a concrete implementation design for the active-security compiler. We also carry out an in-depth discussion on the design and implementation of a custom zero-knowledge proof compiler, alongside a custom toolkit unifying all components.

We shift our focus to assessing the practicality of the active-security compiler in Chapter 6, where we address the third question. Initially, we analyze the asymptotic performance of the compiler, followed by a case study concerning a multi-party sum protocol, which we use to evaluate the implementation's performance. Finally, in Chapter 7, we conclude the thesis with a discussion of our findings and recommendations for future research directions.

2

Literature Review

To embark on this exploration, we begin by offering a detailed explanation of seminal contributions in active-security compilers, each establishing a distinct *paradigm*. We then compare these paradigms and utilize them to construct a taxonomy of additional works, providing a structured view of the complex landscape of this topic. Considering the objective of defining research questions from a broad understanding of the relevant works, we refrain from using rigorous definitions and delving into mathematical intricacies. Nonetheless, there will be occasions where the usage of particular terminologies is necessary; these will be appropriately denoted with *italics* and further explained in Chapter 3.

2.1 Active-Security Compiler Paradigms

This section provides an overview of the GMW, IPS and BDOZ-SPDZ paradigms[1] for constructing active-security compilers. The conceptual framework of each paradigm will be presented alongside an outline of a compiler construction within that paradigm. A summary of various properties for each paradigm can be found in Table 2.1.

2.1.1 The GMW Paradigm

The earliest known active-security compiler originates from the work that marks a cornerstone in the feasibility of actively secure general-purpose MPC. Loosely speaking, the work proves a theorem, stating that with the usage of public-key cryptographic primitives, it is possible to construct an MPC protocol for any function that is secure against any number of passive adversaries and any minority of active adversaries [GMW87]. The proof is given in two parts:

1. The existence of MPC protocols, tolerating any number of *semi-honest* adversaries, is demonstrated by proving the security and correctness of the popular GMW protocol.
2. The existence of MPC protocols, tolerating any minority of *malicious* adversaries, is demonstrated by transforming *any* passively secure protocol into an actively secure protocol in a manner that is commonly referred to as the GMW compiler.

The original GMW compiler was described in the framework of game theory, where a protocol is modeled as a playable *Turing machine* game and the honest, semi-honest and malicious adversaries are modeled as players. Conceptually, the compiler works by taking the construction of any passively secure protocol and adapting it such that all parties are forced to behave semi-honestly with respect to each message that it sends. In a refined version of the compiler, given in [Gol04], the compiled protocol's security in the strict multi-party setting is further subdivided into active security *with abort* and *full* active security. In the active-security-with-abort model, the compiled protocol tolerates any majority of malicious

[1] These acronyms are derived from the initial letters of the authors' surnames.

adversaries and assumes only the existence of *one-way functions* (OWFs). To achieve this, it makes use of two cryptographic primitives: The first one is a commitment scheme, which is collection of algorithms used by a protocol that allows a party to initially commit to a private value, and later reveal the committed value in a way that ensures the value remains unchanged. The second one is a *zero-knowledge proof* (ZKP) system, which is a protocol that allows a party to prove the validity of some statement without revealing any information beyond its validity.

The GMW compiler, roughly speaking, enhances the passively secure MPC protocol in the following steps:

1. The parties engage with each other by
   (a) committing to their private input,
   (b) committing to private unbiased randomness via a coin-tossing protocol.
2. The parties emulate the actively secure protocol sequentially by
   (a) running the passively secure sub-protocol from the sequence,
   (b) providing a ZKP that assures the resulting message with respect to its private input, randomness and previously received messages, was computed correctly.

In the full active-security model, all parties are guaranteed the correct output despite a malicious adversary deviating from the protocol. Conceptually, the malicious adversary is deterred from cheating as the honest parties gain the ability to learn the adversary's input when cheating is detected. This is accomplished through the usage of a primitive called a *verifiable secret-sharing scheme*, where an adversary's value can be jointly reconstructed by a majority of honest parties upon detecting any inconsistencies in the value. This primitive assumes the existence of *enhanced trapdoor permutations* (ETDPs), which is a stronger assumption than assuming the existence of OWFs.

Although the compiler accommodates both active-security models and can be applied to MPC protocols regardless of their construction, it should be noted that this level of security is restricted to security under *sequential composability*. Consequently, this requires the MPC protocol to be carried out over a *synchronous communication* network. Furthermore, the construction of ZKPs for message correctness cannot be realized in a non-black-box manner due to their dependence on OWFs, meaning that the proof for a given statement requires access to the code of the statement.

2.1.2 The IPS Paradigm

A completely different approach for obtaining active-security from any passively secure protocol, requiring only black-box access to its underlying cryptographic primitives, was first described in the work of [IPS08], commonly referred to as the IPS compiler. Unlike the GMW compiler, the IPS compiler does not carry out a direct passive-to-active security transformation. Instead, it relies on a passively secure protocol to transform an actively secure protocol in the *honest-majority* setting into an actively secure protocol in the *dishonest-majority* setting. Conceptually, the compiler requires the parties to use a passively secure *inner* protocol to simulate the execution of an honest-majority *outer* protocol, which involves more virtual parties than there are real parties. Both protocols are realized in the *client-server* model [HM00, DI05] where the real parties are typically referred to as *clients*, while the virtual parties are referred to as *servers*. Active security is obtained from all clients independently monitoring a random subset of all servers, ensuring that the protocol is followed.

Since the outer protocol must be actively secure in the honest-majority setting, it is possible to realize this without relying on computational hardness assumptions, which is also understood as achieving *information theoretic* (IT) security. The inner protocol, on the other hand, orchestrates the secure emulation of the outer protocol using so-called *oblivious watchlists*, which makes use of a symmetric-key cipher obtained from an OWF and OT channels obtained from an ETDP. This can be achieved in more than one way, but it generally makes use of an OT primitive and implements the outer protocol simulation according to the *MPC-in-the-head* (MPCitH) technique [IKOS07].

The IPS compiler can be sketched out as follows:

1. Each client sets up an oblivious watchlist for every other client by
   (a) sampling private randomness for every passively secure inner sub-protocol, equal to the total number of servers,
   (b) generating a unique private key for each server,
   (c) obtaining a fraction of each party's randomness and keys using OT.
2. The clients securely emulate the outer protocol at each round by
   (a) carrying out the computations of each server through their respective inner sub-protocols, providing their private randomness as input,
   (b) broadcasting the encrypted outputs for each sub-protocol, and checking that the servers on their watchlist computed the correct output.

For the IPS compiler to operate as an active-security compiler for any passively secure MPC protocol, it is necessary to select appropriate inner and outer protocols in the client-server model where the outer protocol computes the same function as the passively secure protocol. Additionally, since there are n clients (or real parties) and k many watchlists per client, the compiler requires $O(n^2k)$ servers[2]. Furthermore, the compiler is secure under *universal composability* (UC) in the framework of [Can01], specifically in the OT-hybrid model where inner sub-protocols based on OT may be composed in a parallel or concurrent manner. Lastly, the compiler makes black-box use of the inner and outer protocols, since the watchlist constructions of the inner sub-protocols only depend on the input and output behavior of the outer protocol.

2.1.3 The BDOZ-SPDZ Paradigm

Techniques from the BDOZ and SPDZ Protocols

More recently, the focus has switched to achieving practically efficient actively secure MPC, specifically targeting protocols for arithmetic computation based on secret sharing. A powerful technique called *authenticated secret sharing* was first introduced in [BDOZ11] and considerably improved in [DPSZ12] for obtaining active security in the dishonest-majority information-theoretic setting. Both works, commonly referred to as the BDOZ and SPDZ protocol, achieve this by *augmenting* all secret shares through the use of an inexpensive signature, which only needs to be verified once the overall computation is completed. Conceptually, both protocols initially compute these signatures using an IT-secure primitive and proceed to carry out all arithmetic operations on the shares as well as on their signatures. To enable these operations while ensuring privacy and integrity, the primitive requires a secret key and a collection of correlated randomness, both of which are obtainable under computational hardness assumptions. Due to the large computational overhead associated with

[2] In [LOP11], the IPS compiler is fundamentally improved and the number of servers is reduced to $O(nk)$.

obtaining these values, as well as the values being independent from the protocol input, their acquisition is delegated to a so-called *pre-processing* protocol. Protocols that separate computationally expensive from inexpensive operations in this way are realized in the *pre-processing* model [DO10, BDOZ11], and are divided into an *offline* pre-processing and *online* phase.

Generally speaking, the BDOZ-SPDZ paradigm may be viewed as an active-security compiler for general-purpose passively secure MPC protocols based on secret sharing. Both BDOZ and SPDZ-style constructions utilize a primitive called an information-theoretic *message authentication code* (MAC), which can be thought of as a cryptographic hash function that leaks zero information about the key and plaintext. Its security isn't reliant on computational assumptions, and it typically has the property of being additively homomorphic, allowing for computing the sum of two MACs as the MAC of the sum. The primary distinction in the two types of protocols lies in how the MAC is computed: in BDOZ, each party possesses distinct MACs of a secret share for every other party, while in SPDZ, this is reduced to a single MAC of a secret share derived from a global key that is only revealed at the end. This results in both smaller communication and computational overhead.

In extending the BDOZ approach, the SPDZ protocol introduces augmented shares in two varieties:

1. Those with a MAC tied to each party's individual private key.
2. Those with a MAC tied to a distributed global key.

For brevity, we will refer to them as augmented shares composed of pairwise and global MACs, denoted by $[\![\cdot]\!]$ and $\langle\cdot\rangle$, respectively.

Lastly, in terms of security, both BDOZ and SPDZ-style constructions are proven to be statistically UC-secure, specifically in the hybrid model that assumes secure pre-processing and secure input commitments. Both types of constructions derive their security from the existence of OWFs and the concrete Ring-*learning with errors* (LWE) assumption[3].

Description of the SPDZ Protocol

In order for parties to carry out arithmetic operations on augmented shares, SPDZ makes use of an additively homomorphic MAC. As a result, the linear operations $\langle x\rangle + \langle y\rangle = \langle x+y\rangle$, $c + \langle x\rangle = \langle c + x\rangle$ and $c \cdot \langle x\rangle = \langle c \cdot x\rangle$ can be carried out locally. Multiplication on the other hand is achieved by a technique known as *Beaver's trick* [Bea92], which works as follows: all parties consume a triple of correlated random shares $\langle a\rangle, \langle b\rangle, \langle a \cdot b\rangle$, typically called a *Beaver triple*, and broadcast their masked augmented secret shares $\epsilon := \langle x\rangle - \langle a\rangle$ and $\delta := \langle y\rangle - \langle b\rangle$. They then apply Beaver's trick where they compute $\langle c\rangle + \epsilon \cdot \langle b\rangle + \delta \cdot \langle a\rangle + \epsilon \cdot \delta = \langle x \cdot y\rangle$.

Moreover, the SPDZ protocol relies on three cryptographic primitives, which are used extensively during pre-processing. The first one is a *somewhat homomorphic encryption* (SHE) scheme, which is a cryptosystem enabling direct evaluation of specific arithmetic circuits on ciphertexts. In this context, an SHE scheme only needs to support circuits with a multiplicative depth of one, and its security is derived from the concrete LWE assumption for ring structures [LPR10]. The second primitive is a specific kind of ZKP, called a *zero-knowledge proof of plaintext knowledge* (ZKPoPK). In essence, if a verifier knows a ciphertext, a prover can demonstrate in zero-knowledge that it knows the corresponding plaintext. A simple and efficient construction can be derived from a variant of Schnorr's identification protocol [Sch91]. Lastly, the protocol utilizes a commitment scheme.

[3]This assumption is typically required for the construction of SHE primitives, but a different type of prepossessing protocol may involve other concrete assumptions.

The protocol operates as follows:

1. The parties perform pre-processing to obtain secret shares of a global key with pairwise MACs, along with secret shares of Beaver triples with global MACs, as follows:
   (a) Each party samples additive global key shares, encrypts them using a SHE scheme and obtains pairwise MACs of the resulting sum.
   (b) Similarly, all parties obtain Beaver triples and augment them using global MACs.
   (c) Finally, each party provides a ZKPoPK given their public ciphertexts.
2. The parties compute the MAC for each arithmetic operation as follows:
   (a) If two shares are added, all parties locally sum up their MACs.
   (b) If two shares are multiplied, all parties obtain a MAC of the multiplication by
      i. selecting a beaver triple whose product is checked via a partial opening,
      ii. applying beaver's trick.
3. After running the arithmetic computation, the parties authenticate the final output by
   (a) committing to their output share and all MACs,
   (b) reconstructing the global key,
   (c) opening the commitments to all MACs and validating all partial openings with respect to the global key,
   (d) opening the commitments of all output shares and validating their MAC with respect to the global key.

Since the usage of IT-MACs is applied to each arithmetic circuit gate, protocols in the BDOZ-SPDZ paradigm are not black-box with respect to the passively secure protocol.

2.2 Active-Security Compiler Landscape

We conducted a chronological review of three major paradigms concerning active-security compilers up to the present day. Specifically, these paradigms are GMW [GMW87, Gol04], IPS [IPS08, LOP11], and BDOZ-SPDZ [BDOZ11, DPSZ12]. Even though their constructions vary widely and serve slightly different purposes, each paradigm unveils a powerful technique that aids in the design and construction of an active-security compiler for passively secure MPC protocols. Their properties are summarized and compared in Table 2.1, followed by a discussion on their relevance in shaping concrete research directions.

Table 2.1: Comparison of the major active-security compiler paradigms.

Paradigm	Technique	Back-box	Security	Assumptions	Protocol Purpose
GMW	ZKP*		sequential	$\exists$ OWFs	special/general
IPS	MPCitH†	✓	UC	$\exists$ {ETDPs, OWFs}	special/general
BDOZ-SPDZ	IT-MACs‡		UC	Ring-LWE, $\exists$ OWFs	general

* Used for proving correct usage of sub-protocols in a sequential composition.
† Applied to a honest-majority protocol in the *client-server* model.
‡ Used for authenticating secret shares via SHE and ZKPoPK in the *pre-processing* model.

2.2.1 Similarities

We recall that a general-purpose protocol can be used to compute any MPC function, whereas a special-purpose protocol is composed of multiple protocols that compute a particular MPC function. Both the GMW and IPS paradigm are applicable to both types of protocols, whereas BDOZ-SPDZ is limited to general-purpose secret-sharing based MPC. While the overarching question posed earlier revolves around *special-purpose* protocols, theoretically rendering the BDOZ-SPDZ paradigm unsuitable, practical implementations of the SPDZ compiler suggest otherwise. For instance, the implementations MP-SPDZ [KSS13, Kel20] and SCALE-MAMBA [ACC+18] offer an actively secure protocol runtime environment along with a compiler that translates protocol descriptions into arithmetic circuits. Such protocol descriptions could be treated as sub-protocols, which can be composed with other UC-secure protocols in a modular manner to obtain a special-purpose actively secure protocol.

Moreover, all paradigms achieve active security on the basis of a shared principle: all parties are made to scrutinize the semi-honest behavior of every other party. Although the techniques serve distinct purposes and offer varying security guarantees, their usage within their respective paradigms suggests that they are, in some sense, implementing a ZKP of semi-honest behavior. This notion is exemplified through constructions of (non-)interactive ZKP protocols based on MPCitH [IKOS07], and interactive ZKP protocols based on IT-MACs [WYKW21, BMRS21].

Finally, we note that all paradigms, *in their elementary form*, support the compilation of protocols with passive security into active security *with* abort, tolerating all but one malicious corruption.

2.2.2 Differences

On the contrary, we note that the three paradigms exhibit fundamental differences across all aspects. As discussed above, both the GMW and IPS paradigms target *any* protocol, whereas BDOZ-SPDZ is restricted to secret-sharing based MPC protocols. Additionally, IPS-style compilers are inherently black-box with respect to the passively secure protocol, whereas GMW is not. As a consequence, this enables IPS-style compilers to both preserve IT-security and avoid an analysis of the underlying protocol, whereas GMW-style compilers impose computational security by default and require an analysis of the underlying protocol. From this stand-point the IPS paradigm is the least restrictive compared to the others.

In terms of security guarantees, both the IPS and BDOZ-SPDZ paradigms provide UC-security. This means that a passively secure protocol can be a sequential, concurrent, and/or parallel composition, and the resulting actively secure protocol can be securely composed with other protocols out of the box. Conversely, in the case of GMW, a passively secure protocol must strictly be sequential, and the resulting actively secure protocol may only be composed sequentially as well. Fortunately, a UC-secure variant of the original GMW compiler exists, as proposed in [CLOS02]. While its construction is similar, the underlying ZKP protocol must adhere to the *commit-and-prove* paradigm, which in addition assumes the existence of ETDPs and requires a *trusted setup*[4].

Lastly, in the chronological order of GMW $\prec$ IPS $\prec$ BDOZ-SPDZ, we observe an increase in conceptual complexity[5], which is also characterized by the increase in generic and concrete cryptographic assumptions. In the same order, the overall compiler performance improves,

[4] This is an assumption in which a trusted party instantiates a shared symmetric or public key.

[5] This is not a measure of complexity from computer science or mathematics, but instead means that something is conceptually harder, i.e., relying on a greater body of knowledge.

suggesting that an increasingly better performing compiler can be constructed by leveraging additional cryptographic assumptions.

2.2.3 Additional Works

Two-Party Active-Security Compilers

We have not discussed compiler paradigms for the particular case of two-party computation protocols. Although these protocols are theoretically compilable in the GMW and IPS paradigm, more efficient transformations can be obtained through the *cut-and-choose* technique [LP07]. This technique involves one party 'cutting' its message into pieces and another party obliviously 'choosing' the pieces to check. While primarily designed for GC-based two-party protocols, there are exceptions, such as the three-party protocol in [CKMZ14]. However, it is still uncertain whether a general extension to multi-party protocols is feasible.

General-Purpose Active-Security Compilers

When it comes to compilers for general-purpose protocols, there exist other types of fundamental active-security transformations that deviate from the aforementioned paradigms. For instance, the approach outlined in [CDN01], which utilizes *threshold homomorphic encryption*, predates the BDOZ-SPDZ paradigm while exhibiting some similarities. It similarly requires parties to prove correctness of pre-processed secret shares, but unlike BDOZ-SPDZ, it does not augment secret shares using input-independent randomness. Instead, the parties perform the arithmetic computation on encrypted shares and jointly decrypt the final output.

A fundamentally different approach introduced by [GIP+14, GIP15], involves a code-theoretic construction that transforms the underlying arithmetic circuit rather than the passively secure protocol itself. The transformation converts a corruptible arithmetic circuit into a fault-tolerant one, making it secure against *additive attacks*. It works on the premise that passively secure general-purpose protocols are corruptible when an adversary *adds* arbitrary values to a circuit gate. Since a fault-tolerant circuit is robust against a fraction of noisy wires, the passively secure protocol becomes actively secure at the sole expense of a larger circuit. Other compilers that expand on this approach include [CHI+23, BMY24].

State-Of-The-Art Active-Security Compilers

Lastly, we review and organize an extensive body of state-of-the-art research across the three compiler paradigms into a taxonomy, as shown in Table 2.2. When constructing an active-security compiler, several properties are of interest, including but not limited to, asymptotic and concrete performance, adversary or threshold structure, round complexity, and the class of protocols to which it can be applied.

A particularly desirable feature for compilers that guarantee security with abort is the notion of identifiable abort, better known as *identifiable MPC* (ID-MPC) [IOS12]. Such compiled protocols require parties not only to check for malicious behavior but also to identify the cheater. This is straightforward in the GMW and IPS paradigms due to the checking of ZKPs and watchlists, respectively. However, this feature is not readily present in BDOZ or SPDZ, where all parties can only validate all MACs at the end, allowing a cheater to remain undetected in the event of a premature abort.

Another desirable feature, especially for large-scale MPC, is public auditability, also known as *publicly auditable MPC* (PA-MPC) [BDO14], which enables non-participating parties to verify the correctness of the output. This should not be confused with *public verifiability*

[AO12], a stronger form of ID-MPC that can additionally be checked by non-participating parties and further implies PA-MPC.

Finally, different compilers rely on cryptographic primitives with varying assumptions, which can make standard security proofs challenging. To address this, their security is typically established within specific cryptographic models of computation. The least restrictive is the plain model, where all widely accepted assumptions are realized. However, some cases require the use of a *random oracle* (RO) or additional trust assumptions, such as a *common reference string* (CRS) or a *public-key infrastructure* (PKI).

Table 2.2: Overview of state-of-the-art active-security compilers across the major compiler paradigms. References in **bold** correspond to the seminal works and items in parentheses are optional.

Paradigm	**Reference**	**ID-MPC**	**PA-MPC**	**Corruptions** *(t out of n)*	**Security**[*] *(comp/IT)*	**Purpose** *(general/any)*	**Model**[**] *(plain/RO,CRS,PKI)*
GMW	**[GMW87]**	✓		$t < n$	comp	any	plain
	[CLOS02]	✓		$t < n$	comp	any	CRS
	[IOZ14]	✓	✓	$t < n$	IT	any	plain
	[KZZ16]	✓	✓	$t < n$	comp	any	PKI,CRS
	[ACGJ18][†]	✓		$t < n/2$	comp	general	plain
	[AKP22][†]	✓		$t < n/2 - o(n)$	comp	general	plain
	[AKP23][†]	✓		$t < n/2$	IT	general	plain
IPS	**[IPS08]**	✓		$t < n$	IT	any	RO/CRS/PKI
	[LOP11]	✓		$t < n$	IT	any	RO/CRS/PKI
	[DOS18]	✓		$t = \Omega(\sqrt{n})$	IT	any	plain
	[DOS20][‡]	✓	✓	$t < n$	comp	any	PKI
	[FHKS21][‡]	✓	✓	$t < n$	comp	any	PKI,CRS
	[ADEL22][‡]	✓	✓	$t < n/2$	comp	any	PKI
BDOZ-SPDZ	**[BDOZ11]**			$t < n$	IT	general	PKI
	[DPSZ12]			$t < n$	IT	general	PKI,(RO)
	[BDO14]		✓	$t < n$	comp	general	PKI,RO,(CRS)
	[SF16]	✓		$t < n$	IT	general	PKI
	[SSS22][‡]	✓	✓	$t < n$	comp	general	PKI
	[BMRS23]	✓		$t < n$	IT	general	PKI,CRS
GMW & SPDZ	[BGIN21]	✓		$t < n$	IT	general	plain

[*] Refers specifically to the (IT or computational) security of the *online* phase of a protocol.
[**] Refers specifically to cryptographic models of computation (described in Chapter 3).
[†] These compilers produce constant-round protocols.
[‡] These compilers provide *covert* security, which can be tuned to approach active security.

3
Background

In this chapter, we delve into the foundational topics that lay the groundwork for designing an active-security compiler and, broadly speaking, addressing the three research questions discussed in Section 1.2. Moreover, we aim to provide clarity on terms and concepts introduced in the literature review found in Chapter 2.

We begin by introducing frequently used notation and basic concepts from mathematics and computer science in Section 3.1. These concepts are primarily needed in other definitions and basic cryptographic constructions. Next, in Section 3.2, we explore the relevant cryptographic concepts that are necessary for designing the active-security compiler. Section 3.3 gives a more elaborate introduction to multi-party computation, and finally Section 3.4 outlines both the theoretical and practical concepts of zero-knowledge proofs.

3.1 Notation and Preliminaries

To formulate cryptographic algorithms and establish a notion of security, we rely on constructs from mathematics and theoretical computer science. We will introduce fundamental concepts such as groups and rings in mathematics, as well as complexity classes and reducibility in computer science. We will also extensively use the notations outlined in Table 3.1, alongside symbols that are listed in the glossary preceding Chapter 1.

Table 3.1: Frequently used notation.

Notation	Definition
$[n]$	The set of elements $\{1, 2, \ldots, n\}$.
$X \sim D$	Random variable X follows distribution D.
$u \in_{\mathrm{R}} S$	Randomly sampled u from set S with uniform probability.
1^x	Unary notation denoting string "$11\cdots1$" of length x.
$a \| \| b$	Concatenation of strings a and b.
$a \stackrel{?}{=} b$	Equality test, i.e., $\mathbf{if}\ a = b\ \mathbf{return}\ 1\ \mathbf{else}\ \ \mathbf{return}\ 0$.
$\mathbf{v} \in K^n$	Vector of elements $(v_1, \ldots, v_n)$, each in field K.
$\mathbf{M} \in K^{m \times n}$	Matrix of n rows by m columns with elements in field K.
$f_i(\mathbf{x}) = y_i$	The i-th component in the image of $f : K^n \to K^m$.

3.1.1 Primer on Groups and Rings

In this section, we will explore a couple of basic algebraic structures such as groups, rings, and fields, giving informal definitions and highlighting their usage in cryptography. Additionally, we will also explore a particular structure, in connection to group theory, called an elliptic curve.

Groups

Many of the standard cryptographic *hardness* assumptions, which we will discuss in Section 3.2.2, make use of an algebraic structure called a *group*. Informally speaking, a group is a structure comprising a set G with an associative binary operation $\cdot$, denoted $(G, \cdot)$, where each element has an inverse and there exists an identity element. Under this definition alone, there is a vast number of groups that can be described, such as the *additive* group of integers.

Example 3.1.1 (Additive group)**.** The set of integers $\mathbb{Z}$ forms a group under addition, i.e., $(\mathbb{Z}, +)$, and is typically denoted $\mathbb{Z}^+$.

We are mostly interested in groups with additional properties. For example, the additive group of integers is an *abelian* group, which additionally has the property that the operation is commutative. A particularly relevant family of groups utilized in cryptography are *cyclic* groups.

Definition 3.1.1 (Cyclic group)**.** Let $\langle a \rangle := \{a^n : n \in \mathbb{Z}\}$ denote a subgroup of G generated by an element $a \in G$. A group G is called cyclic if there exists an element $g \in G$ such that $G = \langle a \rangle$.

The archetypical cyclic group, encountered in cryptographic algorithms that assume the "hardness" of integer factorization, is the *multiplicative* group of integers modulo n.

Example 3.1.2 (Multiplicative group)**.** The set of integers coprime to n forms a group under multiplication modulo n, i.e., $\mathbb{Z}_n^* = \{a \mid 1 \leq a < n \text{ and } \gcd(a, n) = 1\}$.

Consequently, the multiplicative group modulo some prime number p is $\mathbb{Z}_p^* = [p - 1]$, and can be equivalently expressed as the quotient group $(\mathbb{Z}/p\mathbb{Z})^*$.

The takeaway is that groups are a simple yet powerful abstraction for reasoning about different mathematical structures, which in turn have direct applications in cryptography.

Rings and Fields

Another area of mathematics that serves great importance in cryptography is *ring theory*. Informally, a ring is a structure comprising a set R with two binary operations, typically addition $+$ and multiplication $\cdot$, denoted $(R, +, \cdot)$, where $(R, +)$ forms an abelian group and multiplication is associative and distributive over addition. In some sense, a ring is an extension of a group with an additional operation, which formalizes a wider range of structures, many of which we are already familiar with.

The archetypical examples of rings are *commutative* rings *with identity*, which additionally have the property that multiplication is commutative and has identity element 1.

Example 3.1.3 (Commutative rings with identity)**.** The sets of integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$, real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$, form a ring under addition and multiplication.

When a commutative ring has the additional property that each non-zero element has a multiplicative inverse, we speak of a *field*. The sets $\mathbb{Q}, \mathbb{R}$ and $\mathbb{C}$ are examples of fields, whereas $\mathbb{Z}$ is not. A field can be viewed as a generalization for a number system that is closed under all arithmetic operations.

A special kind of field that accounts for modular arithmetic is a *finite* field.

Definition 3.1.2 (Finite field)**.** A finite field is a field of the size q, denoted $\mathbb{F}_q$, where q is a prime power.

Finite fields are also a natural abstraction for the way classical computers deal with strings of binary numbers $\{1,0\}^{2^n}$, as they can be mapped to the set $\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z}$. In cryptography, it is typical to specifically assume a *prime field* $\mathbb{Z}_p$, where p is a prime number. This is because modular arithmetic is native to classical computer hardware, and in certain cases, enhances security compared to other finite fields [Ber06].

Other examples of rings that are prevalent in cryptography are *polynomial rings*, which informally is the set of all polynomials defined for some indeterminate variable x with coefficients in the field K, denoted as $K[x]$. Polynomial rings can be used to describe cryptosystems based on linear codes, such as Reed-Solomon codes [RS60], or to construct finite fields the size of some prime power, such as fields of the size 2^n for some power n.

Elliptic Curves

Finally, we look at an algebraic structure that unexpectedly comes up in many areas of mathematics and offers significant security improvements in traditional group-based cryptography, namely an *elliptic curve*. Informally, an elliptic curve is a geometric structure defined by the set of points (x, y) satisfying equations of the form $y^2 = x^3 + ax + b$, commonly known as the *Weierstrass equation*, where a and b are elements of a field K. The typical definition of an elliptic curve is as follows:

Definition 3.1.3 (Elliptic curve)**.** An elliptic curve E over a field K is the set of points

$$E(K) = \{(x, y) \in K \times K \ : \ y^2 = x^3 + Ax + B \text{ with } A, B \in K\} \cup \{O\},$$

where O represents a point at infinity.

In cryptography, an elliptic curve is defined over a finite field $\mathbb{F}_q$, where q is a prime power.

The connection to group theory becomes apparent when considering the group $(E(\mathbb{F}_q), +)$, where the addition operation is defined on all points $(x, y) \in E(\mathbb{F}_q)$ and the identity element is the infinity point. A specific kind of elliptic curve, which will become relevant in later chapters, is a so-called *Edwards curve* [Edw07].

Definition 3.1.4 (Edwards curve over finite field)**.** An Edwards curve over a finite field $\mathbb{F}_q$ is the set of points

$$E_{A,D} = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \ : \ Ax^2 + y^2 = 1 + Dx^2y^2\},$$

parameterized by $A, D \in \mathbb{F}_q$.

An Edwards curve forms a group $(E_{A,D}, +)$ under addition, for which the operation is defined as follows:

Definition 3.1.5 (Edwards addition law)**.** Let $O = (1, 0)$ be the identity point. The sum of any two points (x_1, y_1) and (x_2, y_2) on an Edwards curve $E_{A,D}$ is defined by

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right).$$

It should be noted that O is not strictly a point at infinity, but is typically still called that way to be consistent with the definition of an elliptic curve.

The takeaway is that elliptic curves can be used over traditional groups to improve computational efficiency, and are sometimes necessary to obtain a certain notion of security.

3.1.2 Primer on Computational Complexity

The core of cryptography revolves around constructing algorithms that are *provably secure* against adversaries whose behavior is probabilistic. This notion of security is best modeled by problems that are computationally hard to solve on average, even for most powerful computers known. In this section, we will briefly discuss the relevant concepts from computational complexity theory that are necessary for making sense of provable security.

Decidability and Reducibility

The notion of a (adversarial) party can described as a physical computer, which in turn can be modeled by an abstract machine called a deterministic *Turing machine*. Informally, a Turing machine is a device that carries out a computation on an infinite tape, with the ability to read from and write to it, based on a given description and input. This description comprises a set of states, including “accept” and “reject” states where the machine halts, alongside a transition function for moving between states and updating the tape.

In the framework of Turing machines, an algorithm can be viewed as a machine that solves a *problem*, described by a formal language. Cryptographic algorithms are based on *hard* problems, which roughly speaking, are problems that take an extremely long time to solve. Nonetheless, for a cryptographic algorithm to be useful, its underlying problem, regardless of its hardness, should be solvable in the first place. This notion is known as *decidability*, which can be determined for any problem by considering its associated *decision problem*. A decision problem of this sort is framed as follows: "Given input X, is the output to problem Y a valid solution?". By showing that there exists a Turing machine that halts with a "yes" or "no" response for any input, a problem can be shown to be decidable. Conversely, showing that such a Turing machine does not exist implies that the problem is undecidable.

Another concept that is imperative in cryptography is *reducibility*. The concept of reducibility has various flavors, but in its most general form a reduction from problem A to problem B entails that A is decidable relative to B, commonly termed Turing reducibility and denoted $A \leq_T B$. The notion of “relative decidability” assumes that a decider for A makes use of a black box that provides a solution to problem B, commonly called an *oracle*. Since oracles provide a solution to any problem, regardless of it being decidable, and parties are modeled as ordinary Turing machines, the following theorem is necessary to make meaningful reductions:

Theorem 3.1.1 ([Sip13, Thrm. 6.21])**.** If $A \leq_T B$ and B is decidable, then A is decidable.

Essentially, this means that a solver for A can make use of an ordinary Turing machine that solves B. In the context of cryptography, provable security is the result of a particular type of reduction, called a *security reduction*, to a well-understood hard problem. In practice, the hardness of many problems relevant to cryptography has yet to be proven and are therefore considered to be *hardness assumptions*.

Complexity Classes

We noted that provable security relies on hardness assumptions, which are widely conjectured to be hard since no obvious method for solving the problem efficiently has been found. The notion of hardness, also called *computational intractability*, is *the* central topic studied

in complexity theory. The hardness of a problem is also connected to the famous P vs NP problem.

Informally, the complexity class $\mathbf{P}$ captures all decision problems that are solvable by a *deterministic* Turing machine in a *polynomial* (P) amount of time. Usually these are problems that are considered tractable, which on their own are insufficient for constructing cryptographic algorithms. Analogously, the complexity class $\mathbf{NP}$ captures all decision problems solvable by a *nondeterministic* Turing machine in polynomial time, abbreviated as *nondeterministic polynomial* (NP) time. Equivalently, these are problems that are *verifiable* by a deterministic Turing machine in polynomial time. Therefore NP problems are guaranteed to be efficiently verifiable, but not necessarily efficiently solvable. This also leads to the fundamental result $\mathbf{P} \subseteq \mathbf{NP}$.

It is generally believed that many problems in $\mathbf{NP} \backslash \mathbf{P}$ are hard to solve due to their bruteforce-style Turing machine construction. It turns out that for some problems there are *heuristic* approaches that efficiently solve the problem. The set of NP-hard problems (as suggested by the name) circumvents this by only containing the "hardest" of problems. These are all the problems that admit a polynomial-time reduction from any other NP problem, which consequently means that an efficient solution to an NP-hard problem would also efficiently solve the hardest NP problem. NP-hard problems that are strictly NP are called NP-complete. For ordinary Turing machines not to be able to efficiently solve such problems, it is necessary that $\mathbf{P} \neq \mathbf{NP}$, which, to this day, is a widely accepted conjecture that has yet to be proved.

Since no real computer can non-deterministically solve problems, the next best alternative is to randomly solve a problem. In cryptography, an adversary is modeled as a probabilistic Turing machine that solves problems in *probabilistic polynomial-time* (PPT), which in practice can only solve problems by making random choices up to some small fixed error probability, captured by the $\mathbf{BPP}$ class. Trivially $\mathbf{P} \subseteq \mathbf{BPP}$. Similarly to P vs. NP, the intractability of cryptographic algorithms relies on the assumption that $\mathbf{NP} \not\subseteq \mathbf{BPP}$ and $\mathbf{P} \neq \mathbf{BPP}$. In addition, the problem must also be intractable on *average*, which is not always guaranteed since problems outside of $\mathbf{BPP}$ are intractable in the worst case. However, if a problem is *random self-reducible*, meaning it can be reduced to a random instance of itself, then it is guaranteed to be intractable on average.

The takeaway is that cryptographic algorithms are hard-on-average probabilistic algorithms that depend on hardness assumptions that are captured outside the $\mathbf{BPP}$ class.

3.2 Cryptographic Concepts

In this section we will discuss the necessary definitions and concepts for describing cryptographic protocols. Specifically, we start with meaningful definitions of provable security, followed by a description of various computational assumptions. We then describe the relevant cryptographic models of computation, which introduce additional assumptions that extend beyond computational hardness. Finally, we describe essential cryptographic primitives that form the basis of many cryptographic protocols, including the ones that are used for constructing the active-security compiler.

3.2.1 Computational and Information-Theoretic Security

Provable security is typically divided into two paradigms: the standard *game-based security* paradigm, where the goal is to show that an adversary loses a game against a legitimate challenger by showing that its "advantage" with respect to random guessing is negligible,

and the more general *simulation-based security* paradigm, which roughly involves showing that the knowledge gained by an adversary in the real world is equivalent to knowledge gained by a simulator in an idealized world.

To reason about provable security, it is helpful to first define what negligible functions and indistinguishability are.

Definition 3.2.1 (Negligible function)**.** A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if for every positive polynomial $p(\cdot)$, there exists an $N \in \mathbb{N}$ such that for all $n > N$, $f(n) < \frac{1}{p(n)}$.

A negligible function, typically denoted $\mathrm{negl}(\cdot)$, is used to describe vanishing probabilities in the context of PPT algorithms with respect to some input. Given this, we can define indistinguishability as follows:

Definition 3.2.2 (Indistinguishability, adapted from [Sch23, Def. 1.13])**.** Let $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ be two distribution ensembles defined over S, indexed by n. The ensembles X and Y are said to be:

1. **perfectly indistinguishable** if for all $s \in S$

$$\Pr[X_n = s] = \Pr[Y_n = s]$$

2. **statistically indistinguishable** if for all $s \in S$

$$|\Pr[X_n = s] - \Pr[Y_n = s]| \leq \mathrm{negl}(n)$$

3. **computationally indistinguishable** if for all PPT Boolean distinguishers D

$$|\Pr[D(X_n) = 1] - \Pr[D(Y_n) = 1]| \leq \mathrm{negl}(n).$$

The difference in probabilities of a distinguisher is also called its *advantage*. Moreover, in the computational case, it is common to use security parameter λ in place of n, and in the statistical case the statistical parameter κ. Among the three types, there exists a hierarchy where perfect $\implies$ statistical $\implies$ computational indistinguishability, where statistical is also referred to as *information theoretic* (IT). Finally, it is common to denote two computationally indistinguishable ensembles by $X \overset{c}{\equiv} Y$.

Game-Based vs Simulation-Based Security

In game-based security definitions, computational security can be described in terms of computational indistinguishability. An example for symmetric-key encryption schemes is the definition of *indistinguishibility under chosen-plaintext attack* (IND-CPA). Without going into detail, IND-CPA can be described through computational indistinguishability as follows:

Definition 3.2.3 (IND-CPA, simplified)**.** Let $\mathrm{SKE} = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$ be a symmetric-key encryption scheme. Let $\mathrm{Guess}^{\mathsf{CPA}}_{\mathrm{SKE}}(1^\lambda)$ be a chosen plaintext-attack game where an adversary $\mathcal{A}$ has to correctly guess the ciphertext that corresponds to a plaintext, and λ is the security parameter. The scheme is considered IND-CPA secure if

$$\left|\Pr[\mathcal{A} \text{ wins } \mathrm{Guess}^{\mathsf{CPA}}_{\mathrm{SKE}}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathrm{negl}(\lambda).$$

More specifically, the probability difference is understood as the advantage that a calculated adversary has over a randomly-guessing adversary.

In simulation-based security definitions, computational security is also described in terms of computational indistinguishability, however it involves showing that the advantage of a real adversary over an idealized adversary is negligible. Roughly speaking, a real adversary resides in the *real world*, where it can exploit the limitations of a cryptographic algorithm to learn meaningful information. In contrast, an idealized adversary resides in the *ideal world*, where there is no information leakage beyond the size of the input. Since idealized adversaries do not exist, it is typically called a *simulator*. The simulation-based definition paradigm stems from *semantic security* [GM82], which can be defined as follows:

Definition 3.2.4 (Semantic Security, adapted from [Gol04, Def. 5.2.1])**.** A symmetric-key encryption scheme $\mathrm{SKE} = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$ is semantically secure if for every PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for every distribution ensemble $\{X_n\}_{n\in\mathbb{N}}$ with $|X_n| \leq \mathrm{poly}(n)$ and every pair of polynomially-bounded functions f, h

$$|\Pr[\mathcal{A}(1^n, \mathrm{Enc}_k(X_n), h(X_n)) = f(X_n)] - \Pr[\mathcal{S}(1^n, |X_n|, h(X_n)) = f(X_n)]| \leq \mathrm{negl}(n).$$

It turns out that Definition 3.2.3 and 3.2.4 are equivalent, which, roughly speaking, can be proved by reducing one problem to the other and vice versa. This does not necessarily hold for definitions pertaining to other problems and/or cryptographic schemes.

Generally speaking, game-based and simulation-based security definitions vary depending on the system in question. We saw instances of both types for defining the security of a particular cryptographic *scheme*, which is typically a collection of stand-alone cryptographic algorithms used in no particular order. In the context of cryptographic *protocols*, where a particular usage and order of cryptographic algorithms applies, we will later see that the simulation-based paradigm is used due to its generality.

3.2.2 Computational Assumptions

In Section 3.1.2 we discussed the notions of reducibility and computational hardness, and in the previous subsection we described two paradigms concerning provable security. Implicitly, obtaining computational indistinguishability is equivalent to reducing the cryptographic algorithm in question to a hard problem. In the game-based definitions, constructing a game in which the adversary relies on an oracle that can solve a hard problem, is by definition a proof of reducibility. In simulation-based definitions, distinguishing between the simulator and a real-world adversary requires solving the underlying hard problem, therefore demonstrating reducibility.

The key building block for any computationally secure algorithm is is a problem that is computationally hard to solve on average. As previously discussed, NP-hard and NP-complete problems define the worst-case complexity of computational challenges. However, relying solely on worst-case complexity is insufficient, and obtaining worst-case to average-case reductions often necessitates certain *assumptions*. An example is the NP-complete *linear decoding problem* used in the McEliese cryptosystems [McE78]. In this context, the system relies on *random* decoding, which is widely believed to be hard on average but has yet to be proved. Currently, establishing worst-case to average-case reductions for decoding problems beyond the assumption that $\mathbf{P} \neq \mathbf{NP}$ remains an open problem, with recent results such as [DR22] continuing to rely on certain assumptions.

We will describe a couple of relevant computational assumptions, typically categorized as *generic* or *concrete* assumptions.

Generic Assumptions

The archetypical computational assumption is the existence of *one-way functions* (OWFs). These can informally be described as functions that are easy to compute, but hard to invert, which in the jargon described earlier can be defined as follows:

Definition 3.2.5 (One-way function, adapted from [Gol01, Def. 2.2.1])**.** We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is one-way if f is

1. **easy to compute**: there exists a polynomial-time algorithm F that computes f,
2. **hard to invert**: $\Pr[B(f(x)) = x \mid x \in_{\mathrm{R}} \{0,1\}^*] < \mathrm{negl}(n)$ for any PPT algorithm B.

The existence of OWFs is the most primitive computational assumption used in cryptography. Other generic assumptions that extend the OWF can be defined for, *trapdoor functions* (TDFs), which are easy to invert given a trapdoor, as well as *one-way permuations* (OWPs), *trapdoor permutations* (TDPs) and *enhanced trapdoor permutations* (ETDPs). Unlike TDPs, which can be inverted when partial trapdoor information gets leaked, an ETDP [Gol04, Def. C.1.1] requires full knowledge of the trapdoor. We summarize their definitions in Table 3.2. As a remark, the $\exists$ OWFs assumption corresponds to *Minicrypt* in Impagliazzo's

Table 3.2: Generic computational hardness assumptions.

Assumption	Informal Definition
$\exists$ OWFs	Definition 3.2.5
$\exists$ TDFs	$\exists f$ OWF and $\exists t$ trapdoor such that f^{-1} is easy to compute.
$\exists$ OWPs	$\exists f$ OWF and f is bijective
$\exists$ TDPs	$\exists f$ OWP and $\exists t$ trapdoor such that f^{-1} is easy to compute.
$\exists$ ETDPs	$\exists f$ TDP where f^{-1} is easy to compute if and only if t is a *full* trapdoor.

five worlds [Imp95], which is sufficient for achieving symmetric-key encryption. On the other hand, the $\exists$ TDPs corresponds to *Cryptomania*, which is necessary for public-key encryption and general secure communication.

Whenever a cryptographic algorithm relies on a generic assumption, it suffices to make use of a construction that satisfies the assumption's definition. To put this into perspective, we consider an example in which the generic assumption primitives are constructed in a particular way.

Example 3.2.1 (Constructions based on generic assumptions)**.**

- **OWFs**: The SHA-family hash functions.
- **TDFs**: The Rabin function $f_N(x) = x^2 \pmod N$, where $N = pq$ is a product of primes. The trapdoor is $t = (p, q)$.
- **OWPs**: The discrete logarithm $f_{p,g}(x) = g^x \pmod p$.
- **TDPs**: The ElGamal function $f_{p,g}(m, k) = (g^k, mh^k) \pmod p$ and $h = g^x \pmod p$. The trapdoor is $t = x$.
- **ETDPs**: The RSA function $f_{e,N}(x) = x^e \pmod N$, where $N = pq$ is a product of primes and e is coprime with $(p-1)(q-1)$. The trapdoor is $t = e^{-1} \pmod{(p-1)(q-1)}$.

This means that if an algorithm's security relies on one of these generic assumptions, it equivalently can be obtained from a concrete problem. For example, the active-security

compiler in [IPS08] assumes the existence of oblivious transfer for the construction of the inner-protocol, which can be constructed using an ETDP. As a consequence, the usage of any ETDP construction, such as the RSA function, would be valid.

As we see from Example 3.2.1, all generic assumptions admit a reduction from some concrete problem, such as problems that involve cyclic groups. The problems shown above are widely believed to be hard because no algorithm has been found that can solve them efficiently on average, making them valid hardness assumptions.

Concrete Assumptions

We already encountered problems that rely on concrete assumptions from the reductions shown in Example 3.2.1. The most basic one is the assumption that the *discrete logarithm problem* is computationally hard. The Rabin and RSA function can be further reduced to the *integer factorization problem* and slightly weaker *RSA* problem, whereas the ElGamal function can be reduced to the *computational diffie-hellman problem*, all of which are assumed to be hard.

Roughly speaking, a *concrete assumption* is an the assumption that a concrete problem is average-case hard. These problems are typically studied in the context of number theory, including integer factorization, discrete logarithm, lattice-based and quadratic residuosity problems [GTK16], but also stem from other disciplines such as coding theory [Gal62]. We summarize a list of concrete problems in Table 3.3 that are believed to be average-case hard. Many of the problems in the table are reducible to basic assumptions from the same category, sometimes in a hierarchical manner of increasing difficulty such as $\mathrm{DDH} \leq_T \mathrm{CDH} \leq_T \mathrm{DL}$. Therefore, it is desirable to prove some security property under the basic assumption.

3.2.3 Models of Computation

We discussed the topic of computational hardness assumptions and described them as building blocks in the context of provable security. The fundamental assumption for any proof is that hard problems cannot be reduced to easy problems, i.e., $\mathbf{P} \neq \mathbf{NP}$. However, there are cases in which provable security cannot solely depend on the computational limitations of an adversary. In this context, we consider a cryptographic *model of computation* to be a framework that formalizes the limitations of adversary.

Standard Model

The *standard* model, also called the *plain* model, is the most basic computational model for provable security as it solely relies on computational assumptions. Simultaneously, it is the strictest model, requiring all security claims to be demonstrable via security reductions to well-understood average-case hard problems, such as the ones in Tables 3.2 and 3.3.

Generic Group Model

The *generic group* (GG) model, introduced by [Sho97], extends the standard model by making the assumption that an adversary may only perform operations on group elements via an oracle, hence generalizing all groups and making them indistinguishable. This is a non-trivial assumption because, in practice, some group encodings may be more vulnerable to information leakage over others, making the GG model an *idealized* model. For example, the DL problem requires the usage of any cyclic group that is *sufficiently large*. Despite fixing a large security parameter, an encoding of $\mathbb{Z}_p^*$ allows for the discrete logarithm to be solved

Table 3.3: Concrete problems that are believed to be average-case hard.

Problem	Informal Definition
Integer Factorization (IF)	
IF	Given a composite integer N, factor N into its prime factors p and q.
RSA	Given $N = pq$, e, and $c = m^e \pmod N$, compute m without the private key d.
Discrete Logarithm (DL)	
DL	Given a large cyclic group $\langle g \rangle$ and $y = g^x$, compute x.
CDH[1]	Given a large cyclic group $\langle g \rangle$, g^a, and g^b, compute g^{ab}.
DDH[2]	Given a large cyclic group $\langle g \rangle$, g^a, g^b, and g^c, decide if $c = ab$ or c is random.
Lattice-Based	
SV[3]	Given a lattice L, find the shortest non-zero vector in L.
LWE[4]	Given $\mathbf{A} \in_{\mathrm{R}} \mathbb{Z}_p^{m \times n}$, $\mathbf{e} \in Z^m \sim \mathcal{N}(0,1)^\dagger$ and $\mathbf{b} = \mathbf{Ax} + \mathbf{e} \pmod p$, find $\mathbf{x} \in \mathbb{Z}_p^n$.
Code-Based	
SD[5]	Given $\mathbf{H} \in \mathbb{Z}_2^{(n-k) \times n}$, $s \in \mathbb{Z}_2^{n-k}$, and $w \in \mathbb{Z}$ such that $\mathbf{He}^T = s^T$ and $\lvert\mathbf{e}\rvert = w$, find $\mathbf{e} \in \mathbb{Z}_2^n$.
LPN[6]	Given $\mathbf{A} \in_{\mathrm{R}} \mathbb{Z}_2^{m \times n}$, $\mathbf{e} \in \{0,1\}^m \sim \mathrm{Bernoulli}(\frac{1}{\sqrt{n}})^\dagger$ and $\mathbf{b} = \mathbf{Ax} + \mathbf{e} \pmod 2$, find $\mathbf{x} \in \mathbb{Z}_2^n$.
Quadratic Residuosity	
DCR[7]	Given $N = pq$ and y, determine if there exists x such that $y \equiv x^N \pmod{N^2}$.
QR[8]	Given $N = pq$ and y, determine if there exists x such that $y \equiv x^2 \pmod N$.

† Normal distribution $\mathcal{N}(\mu, \sigma^2)$ and Bernoulli distribution $\mathrm{Bernoulli}(p)$
[1] Computational Diffie-Helman (CDH)
[2] Decisional Diffie-Helman (DDH)
[3] Shortest Vector (SV)
[4] Learning With Errors (LWE)
[5] Syndrome Decoding (SD)
[6] Learning Parity with Noise (LPN)
[7] Decisional Composite Residuosity (DCR)
[8] Quadratic Residuosity (QR)

at a reduced cost through an *index calculus* algorithm, whereas the group $E(\mathbb{Z}_p)$ cannot be solved the same way [Adl79].

Random Oracle Model

The *random oracle* (RO) model, formally introduced by [BR93], is an idealized model in which all cryptographic hash functions are replaced by a *random oracle*, causing the output of the assumed hash function to appear truly random to an adversary. We note that a *cryptographic hash function* is a hash function $H : \{0,1\}^* \to \{0,1\}^k$ that satisfies the following security properties:

1. **Pre-image resistant**: H is an OWF [Def. 3.2.5].
2. **2nd pre-image resistant**: Given x, it is hard to find $x' \neq x$ such that $H(x) = H(x')$.
3. **Collision resistant**: It is hard to find any $x' \neq x$ such that $H(x) = H(x')$.

Furthermore, a random oracle is simply a random function $\mathcal{O} : \{0,1\}^* \to \{0,1\}^k$, therefore in the RO model we assume $H(x) := \mathcal{O}(x)$ for all x.

Common Reference String Model

The *common reference string* (CRS) model, described in [BFM88, Dam00], extends the standard model whereby the setup parameters to some cryptographic algorithm are assumed to be legitimately generated by a trusted entity, making it a *trust* model. In the context of provable security, it is assumed that an adversary cannot generate or manipulate the common reference string and must use the same one that is accessible to all other entities.

Public-Key Infrastructure Model

The *public-key infrastructure* (PKI) model, is a trust model in which it is assumed that a public key of an entity used in a public-key cryptographic scheme is authentic. In the context of provable security, it is assumed that an adversary cannot tamper with anyone's public key. However, in practice, an adversary may hijack the announcement of a public key over an insecure channel, misleading others into believing the modified key belongs to the designated entity. Similar to the CRS model, a trusted entity, known as a *certification authority*, is required to certify the authenticity of a public key.

3.2.4 Cryptographic Building Blocks

Having established the foundation of provable security, computational assumptions and a few cryptographic models of computation, we are almost ready to transition into the more advanced topics of secure multi-party computation and zero-knowledge proofs. Before doing so, we will first discuss commonly used cryptographic building blocks, commonly referred to as *cryptographic primitives*.

We will describe two fundamental and frequently used *cryptographic schemes*, alongside two *cryptographic protocols* that are particularly important in secure multi-party computation and zero-knowledge proofs. Where applicable, we make use of the cyclic group $\mathbb{Z}_p^*$.

Public-Key Encryption Schemes

We begin with the definition of a *public-key encryption* (PKE) scheme, or public-key *cryptosystem*, as it is arguably the most widely encountered primitive. Informally, a PKE scheme enables anyone to securely encrypt a plaintext message using one's public key, ensuring it can only be decrypted using the private key. Formally, it is defined as follows:

Definition 3.2.6 (Public-key encryption scheme)**.** An encryption scheme is a triple of algorithms $(\mathrm{KeyGen}, \mathrm{Encrypt}_{\mathrm{pk}}, \mathrm{Decrypt}_{\mathrm{sk}})$, which are defined as follows:

- $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{KeyGen}(1^\lambda)$: A PPT algorithm that takes a security parameter λ and outputs a pair of keys: a public key pk and a secret key sk.
- $c \leftarrow \mathrm{Encrypt}_{\mathrm{pk}}(m)$: A PPT algorithm that takes a message m and the public key pk, and outputs a ciphertext c.
- $m \leftarrow \mathrm{Decrypt}_{\mathrm{sk}}(c)$: A deterministic algorithm that takes a ciphertext c and the secret key sk, and outputs the original message m.

Additionally, this scheme must satisfy the following properties:

- **Correctness:** For all $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{KeyGen}(1^\lambda)$ and for all messages m, it holds that $\mathrm{Decrypt}_{\mathrm{sk}}(\mathrm{Encrypt}_{\mathrm{pk}}(m)) = m$.
- **Security:** The scheme is semantically secure.

PKE schemes serve a multitude of purposes, including establishing private communication channels, creating identification protocols, and creating commitment schemes. The latter, which we will discuss afterwards, is also a fundamental cryptographic primitive. As a practical example, we describe the construction of the ElGamal cryptosystem, which is secure under the DDH assumption.

Construction 3.2.1 (ElGamal cryptosystem [ElG85])**.**

$\mathrm{KeyGen}(p)$	$\mathrm{Encrypt}_{\mathrm{pk}}(m)$	$\mathrm{Decrypt}_{\mathrm{sk}}(c)$
$g \in \mathbb{Z}_p^*$ is a generator	$r \in_{\mathrm{R}} \mathbb{Z}_{\mathrm{ord}(g)}$	$(c_1, c_2) \leftarrow c$
$x \in_{\mathrm{R}} \mathbb{Z}^*_{\mathrm{ord}(g)}$	$c_1 \leftarrow g^r \pmod p$	**return** $m \leftarrow c_2 \cdot c_1^{-x} \pmod p$
$h \leftarrow g^x \pmod p$	$c_2 \leftarrow m \cdot h^r \pmod p$	
return $(\mathrm{pk}, \mathrm{sk}) \leftarrow ((g,h),x)$	**return** $c \leftarrow (c_1, c_2)$	

While the ElGamal cryptosystem can be used as a traditional PKE, it is primarily used to construct other cryptographic primitives. Additionally, the ElGamal cryptosystem is an example of a *partially homomorphic encryption* scheme, and in particular, it is *multiplicatively* homomorphic:

$$\begin{aligned}\mathrm{Encrypt}_{\mathrm{pk}}(m) \cdot \mathrm{Encrypt}_{\mathrm{pk}}(m') &= (g^r, m \cdot h^r) \cdot (g^{r'}, m' \cdot h^{r'}) \\ &= (g^{r+r'}, (m \cdot m') \cdot h^{r+r'}) \\ &= \mathrm{Encrypt}_{\mathrm{pk}}(m \cdot m').\end{aligned}$$

Commitment Schemes

Next, we look at *commitment schemes*. Informally, a commitment scheme allows one party to commit to a chosen value and share the commitment with another party, keeping the value hidden until it must be truthfully revealed. Formally, it is defined as follows:

Definition 3.2.7 (Commitment scheme)**.** A commitment scheme is a triple of algorithms $(\mathrm{Setup}, \mathrm{Commit}_{\mathrm{pp}}, \mathrm{Open}_{\mathrm{pp}})$, which are defined as follows:

- $\mathrm{pp} \leftarrow \mathrm{Setup}(1^\lambda)$: A PPT algorithm that takes a security parameter λ and outputs public parameters pp.
- $c \leftarrow \mathrm{Commit}_{\mathrm{pp}}(m, d)$: A deterministic algorithm that takes a message m and randomness d, and outputs a commitment c.
- $\{0,1\} \leftarrow \mathrm{Open}_{\mathrm{pp}}(c, m, d)$: A deterministic algorithm that takes a commitment c, message m and randomness r, and outputs 1 if valid and 0 otherwise.

Additionally, a commitment scheme must satisfy the following security properties:

- **Binding:** For any PPT adversary $\mathcal{A}$, it holds that

$$\Pr\left[\begin{array}{c}(m, m', d, d') \leftarrow \mathcal{A}(1^\lambda) \\ m \neq m' \wedge \mathrm{Commit}_{\mathrm{pp}}(m, d) = \mathrm{Commit}_{\mathrm{pp}}(m', d')\end{array}\right] < \mathrm{negl}(\lambda).$$

- **Hiding:** For any $m \neq m'$, it holds that

$$\{\mathrm{Commit}_{\mathrm{pp}}(m, D_\lambda)\}_{\lambda \in \mathbb{N}} \stackrel{c}{\equiv} \{\mathrm{Commit}_{\mathrm{pp}}(m', D_\lambda)\}_{\lambda \in \mathbb{N}},$$

  where D_λ is uniformly distributed over $\{0,1\}^\lambda$.

In the presence of a PPT adversary, a binding property ensures the commitment cannot be forged, while a hiding property ensures the input value remains hidden. Currently, the properties are defined to be *computationally* binding and hiding. Their definitions can be modified to be IT binding and hiding, if it can be shown that above holds for computationally unbounded adversaries and statistically indistinguishable ensembles. It turns out that a commitment scheme can be either IT binding and computationally hiding, or computationally binding and IT hiding.

A practical construction can be obtained in the form of a Pedersen commitment scheme, which is secure under the DL assumption.

Construction 3.2.2 (Pedersen commitment scheme [Ped92])**.**

$\mathrm{Setup}(p)$	$\mathrm{Commit}_{\mathrm{pp}}(x, r)$	$\mathrm{Open}_{\mathrm{pp}}(c, x, r)$
$g \in \mathbb{Z}_p^*$ is a generator	**return** $c \leftarrow g^r \cdot h^x$	**return** $c \stackrel{?}{=} g^r \cdot h^x$
$h \in_{\mathrm{R}} \mathbb{Z}_p^* \setminus \{1\}$		
return $\mathrm{pp} \leftarrow (g, h)$		

Here, r is called a *blinding factor*. The Pedersen commitment scheme, like the ElGamal cryptosystem, is partially homomorphic; however, it is specifically *additively* homomorphic:

$$\begin{aligned}\mathrm{Commit}_{\mathrm{pp}}(x, r) \cdot \mathrm{Commit}_{\mathrm{pp}}(y, s) &= (g^r \cdot h^x) \cdot (g^s \cdot h^y) \\ &= g^{r+s} \cdot h^x x + y \\ &= \mathrm{Commit}_{\mathrm{pp}}(x + y, r + s).\end{aligned}$$

Finally, we note that the ElGamal cryptosystem (Construction 3.2.1) can be converted into a commitment scheme by eliminating the $\mathrm{Decrypt}_{\mathrm{sk}}(c)$ algorithm, replacing it with a deterministic opening algorithm, and transforming the PPT algorithm $\mathrm{Encrypt}_{\mathrm{pk}}(m)$ into a deterministic commit algorithm. We also note that the Pedersen construction is computationally binding and IT hiding, whereas the ElGamal construction is IT binding and computationally hiding.

Oblivious Transfer Protocols

The primitives we have discussed up to now have been cryptographic schemes that do not require any interaction. When interaction is required to securely accomplish a certain goal, we require a *cryptographic protocol*. In the context of secure multi-party computation, one of the most fundamental primitives is an *oblivious transfer* (OT) protocol. At its core, OT is a two-party protocol that involves a sender and a receiver. The objective, also called a *functionality*, is for the receiver to secretly query one piece of information out of many from the sender. The sender provides the information but remains unaware of which specific piece was queried.

We will focus on a 1-out-of-2 OT protocol, commonly denoted as $\binom{2}{1}$-OT. To formally define the protocol, we need to describe the interaction between the involved parties and a trusted third party that computes the functionality, known as the *ideal functionality*. We will provide a more detailed explanation of this with regards to proving security in the next section. For now, it is sufficient to describe the ideal functionality solely in terms of its input and output behavior, treating it as an oracle. The ideal functionality is illustrated in Figure 3.1. Here, a sender transmits two pieces of information, (x_0, x_1), to the trusted party, while a receiver queries one of the pieces using the index $s \in \{0, 1\}$. The protocol is considered successful when the receiver obtains x_s, and both the sender and receiver do not gain any additional information.

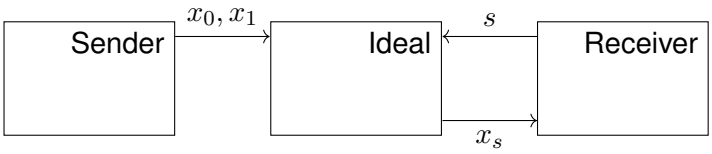


Figure 3.1: $\binom{2}{1}$-OT ideal functionality.

As an example, we consider a simple construction from [BM89], known as the Bellare-Micali construction, which is based on the ElGamal cryptosystem as described in Construction 3.2.1. The Bellare-Micali construction can be generalized to any PKE scheme, as shown in Figure 3.2, given that all public keys generate the same cyclic group. Furthermore, the

Sender		**Receiver**
$x_0, x_1 \in \{0,1\}$		$s \in \{0,1\}$
$(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{KeyGen}(1^\lambda)$		
	$\xrightarrow{\mathrm{pk}}$	
		$(\mathrm{pk}_s, \mathrm{sk}_s) \leftarrow \mathrm{KeyGen}(1^\lambda)$
		$\mathrm{pk}_{1-s} \leftarrow \mathrm{pk} \cdot \mathrm{pk}_s^{-1}$
	$\xleftarrow{(\mathrm{pk}_0, \mathrm{pk}_1)}$	
$\mathrm{pk}_0 \cdot \mathrm{pk}_1 \stackrel{?}{=} \mathrm{pk}$		
Abort if false		
$c_0 \leftarrow \mathrm{Encrypt}_{\mathrm{pk}_0}(m_0)$		
$c_1 \leftarrow \mathrm{Encrypt}_{\mathrm{pk}_1}(m_1)$		
	$\xrightarrow{(c_0, c_1)}$	
		$x_s \leftarrow \mathrm{Decrypt}_{\mathrm{sk}_s}(c_s)$

Figure 3.2: Bellare-Micali $\binom{2}{1}-$OT protocol using a PKE scheme.

security of the protocol relies on the security of the underlying PKE scheme. For example, the ElGamal cryptosystem is secure under the DDH assumption, which ensures that the receiver is unable to decrypt both pieces of information from just a single secret key.

Coin Tossing Protocols

It is evident that cryptographic algorithms rely on randomness for ensuring security. In many cases, the randomness is utilized by a party to secure its own values, making it irrelevant whether the randomness is sampled in a biased or unbiased manner. However, in other cases, the choice of randomness can impact the security of other parties' values. Such a case can be exemplified as follows:

Example 3.2.2. Consider a simplified version of the OT protocol from Figure 3.2 where the receiver independently selects both pk_s and pk_{1-s} without any involvement from the sender. The security for the sender is guaranteed only if the receiver obtains pk_s from sk_s and *randomly* samples pk_{1-s}.

In the example above, the receiver is supposed to choose a random public key, but it could clearly use *biased* randomness (the private key) to learn both pieces of information from the sender. To overcome this problem, both parties could jointly obtain a number of random bits

via a *coin-flipping* protocol, also called a coin-tossing protocol, to ensure the randomness is *unbiased*. Formally, a coin-flipping protocol can be defined by the ideal functionality shown in Figure 3.3. The idea is that two parties, e.g., Alice and Bob, have no influence over the random bit r that they both learn.

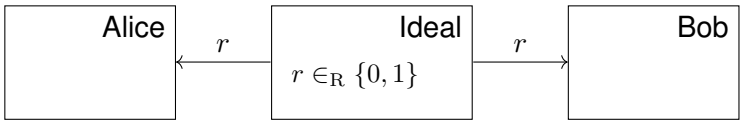


Figure 3.3: Coin tossing ideal functionality.

The first coin-flipping protocol was presented in [Blu81], commonly referred to as Blum's coin-flipping protocol, and is inspired by the concept of two parties flipping a coin over a phone call. It roughly involves one party making a choice and the other party revealing the coin flip in such a way that the initial choice cannot be revoked. The protocol solves this problem using a commitment scheme and is described in Figure 3.4.

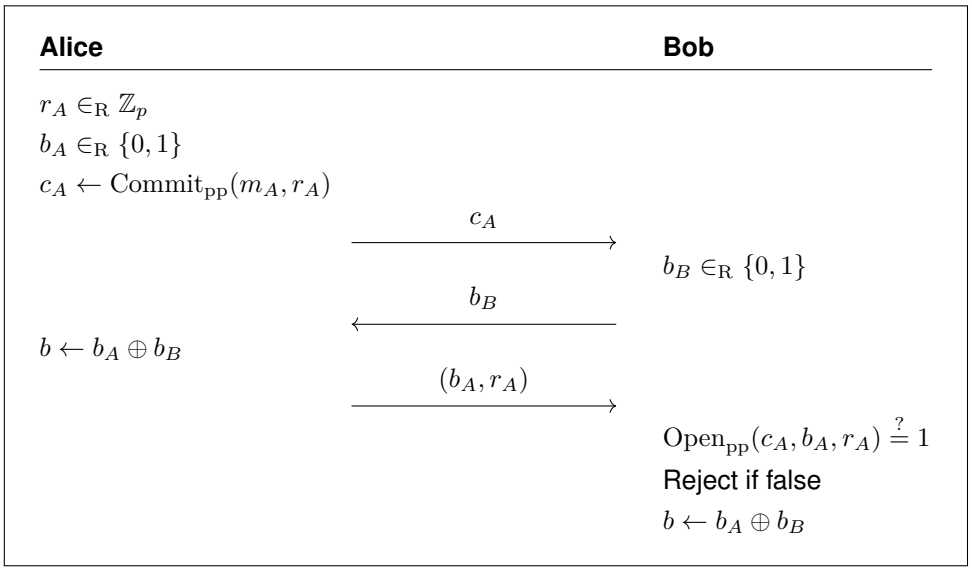


Figure 3.4: Blum's coin-flipping protocol using a commitment scheme.

3.3 Secure Multi-Party Computation

Secure *multi-party computation* (MPC), as introduced in Chapter 1, is the study of cryptographic protocols that enable mutually distrusting parties to compute a function together while keeping their inputs hidden. In this section, we will describe what an MPC protocol is, followed by an explanation of various models of interest and fundamental techniques.

3.3.1 Definition and Models

The problem of MPC is best described as follows: A set of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, who each hold a private value $x_1, x_2, \ldots, x_n$, want to compute a function $f(x_1, x_2, \ldots, x_n)$ without revealing anything about their values. Informally, an MPC protocol is a cryptographic protocol Π_f where each party $P_i \in \mathcal{P}$ communicates some representation of x_i over a communication network, carrying out computations on all received values until each party ends up learning $f(x_1, x_2, \ldots, x_n)$.

Furthermore, an MPC protocol can be best distinguished from other cryptographic protocols by its concrete properties. According to [Lin21], the widely undisputed properties of MPC are the following:

1. **Privacy**: No party should learn anything meaningful beyond the function output.
2. **Correctness**: All parties should learn the correct function output.

Additional desirable properties, albeit not strictly necessary, include:

3. **Input independence**: Corrupted parties have no insight into the inputs of the honest parties.
4. **Guaranteed output delivery**: Corrupted parties cannot prevent honest parties from obtaining their output.
5. **Fairness**: Corrupted parties cannot prevent honest parties from learning the correct output.

Security Models

To define security in the context of MPC, we must describe how an adversary behaves. An adversary is an entity or participant that aims to prevent honest parties from achieving the protocol's objective, thereby compromising the properties mentioned above. There are three main types of adversaries: the *passive* and *active* adversaries, as already described in Chapter 1, and a weaker notion of an active adversary known as a *covert* adversary, who only cheats if the probability of getting caught is small. We will only focus on security models for passive and active adversaries.

Both the passive and active security models are defined in the simulation paradigm, as described in Section 3.2.1; intuitively, an MPC protocol is secure when a distribution of real protocol executions by an adversary is computationally indistinguishable from a distribution of simulated protocol executions. In order for the simulator to obtain the protocol outputs, it relies on some ideal construction of the protocol, referred to as a *functionality*.

Definition 3.3.1 (Functionality, adapted from [Gol04])**.** An n-party functionality f is a function $f = (f_1, \ldots, f_n)$ where f_i is a random variable that denotes the output to P_i.

Essentially, a functionality is a black-box PPT algorithm that will always correctly compute the protocol execution.

Moreover, we need to describe the internal state of a party during a protocol execution. For this, it is common to describe the *view* and the *output* of a given party.

Definition 3.3.2 (View and output, adapted from [Lin17])**.** Let Π be an n-party MPC protocol and let λ be the security parameter. Each party is equipped with the following:

- $\mathrm{view}_i^{\Pi}(x_1, \ldots, x_n, \lambda) = (x_i, r^i; m_1^i, \ldots, m_t^i)$, where x_i is the input held by P_i, r^i its randomness, and m_j^i the j-th message that it received.
- $\mathrm{output}_i^{\Pi}(x_1, \ldots, x_n, \lambda)$ is the final output computed from $\mathrm{view}_i^{\Pi}(x_1, \ldots, x_n, \lambda)$.

Moreover, the joint output of all parties is denoted $\mathrm{output}^{\Pi} = (\mathrm{output}_1^{\Pi}, \ldots, \mathrm{output}_n^{\Pi})$.

This brings us to the definition of *passive security*, which states that a simulator producing the views of corrupted parties with the functionality output in the ideal world is indistinguishable from their actual views with the true joint output in the real world. Formally, this is stated as follows:

Definition 3.3.3 (Passive security, adapted from [Lin17, Gol04])**.** Let $\mathbf{x} = (x_1, \ldots, x_n)$ denote the inputs to some n-party MPC protocol Π. The protocol is passively secure if for any PPT adversary $\mathcal{A}$ that corrupts $I \subseteq [n]$ parties and learns $\{\text{view}_i^{\Pi}(\mathbf{x}, \lambda)\}_{i \in I}$, there exists a PPT simulator $\mathcal{S}$ such that

$$\{(\mathcal{S}(1^\lambda, I, \mathbf{x}_I, f_I(\mathbf{x})), f(\mathbf{x}))\}_{\mathbf{x},\lambda} \stackrel{c}{\equiv} \{(\mathcal{A}(1^\lambda, I, \mathbf{x}, \Pi), \text{output}^{\Pi}(\mathbf{x}, \lambda))\}_{\mathbf{x},\lambda}.$$

This definition specifically captures the notion of passive security, focusing on an adversary's capability to obtain the internal states of any number of corrupted parties, provided that they correctly follow the protocol and all parties ultimately learn the correct output. An active adversary, on the other hand, would be able to insert arbitrary inputs to potentially deny privacy and correctness.

To define active security in the simulation paradigm, a stronger notion of functionality is necessary, specifically an *ideal* functionality. This refers to a trusted third party that computes the functionality and *cannot* be corrupted by any adversary. Since describing the interaction between the simulator and the ideal functionality is lengthy, it is common to define ideal and real output distributions instead.

Definition 3.3.4 (Ideal and real distributions, adapted from [Lin17, Gol04])**.** Let a protocol Π, functionality f, simulator $\mathcal{S}$, and adversary $\mathcal{A}$ be defined as in Definition 3.3.3. Moreover, let $\mathcal{F}$ be an ideal functionality defined for f that only sends outputs to honest parties and can only be directed by $\mathcal{S}$ to either halt or continue. The output distributions in the real and ideal worlds are defined as follows:

- $\text{IDEAL}_{\mathcal{F},I,\mathcal{S}}(\mathbf{x}, \lambda)$ is the pair of outputs of honest parties and $\mathcal{S}$ using $\mathcal{F}$.
- $\text{REAL}_{\Pi,I,\mathcal{A}}(\mathbf{x}, \lambda)$ is the pair of outputs of honest parties and $\mathcal{A}$ using Π.

These distributions differ from the ones used in Definition 3.3.3 as the simulator cannot produce a meaningful internal state for any corrupted party, and can at most abort the execution early. Consequently, the definition of *active security with abort* can be stated as follows:

Definition 3.3.5 (Active security with abort, adapted from [Lin17, Gol04])**.** Let $\mathbf{x} = (x_1, \ldots, x_n)$ denote the inputs to some n-party MPC protocol Π. The protocol is actively secure if for any PPT adversary $\mathcal{A}$ that corrupts $I \subseteq [n]$ parties, there exists a PPT simulator $\mathcal{S}$ such that

$$\{\text{IDEAL}_{\mathcal{F},I,\mathcal{S}}(\mathbf{x}, \lambda)\}_{\mathbf{x},\lambda} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi,I,\mathcal{A}}(\mathbf{x}, \lambda)\}_{\mathbf{x},\lambda}.$$

The take away is that both passive and active security are defined in the simulation paradigm, with subtle differences in how the simulation is carried out. We also note that according to this definition of passive and active security, a fundamental theorem states that the *sequential composition* of two protocols is also secure. To ensure security under *concurrent* or *parallel composition*, the protocols must adhere to a stronger security definition as specified by the *universal composability* (UC) framework of [Can01].

Communication Models

Another crucial aspect of an MPC protocol is the way in which parties communicate with each other. Traditionally, there are two types of *communication channels*:

1. **Point-to-point channels**: Private channels between pairs of parties.

2. **Broadcast channels**: Public channels shared by a set of parties.

A communication channel derives its security guarantees from the cryptographic protocol that it employs, so it is common to model point-to-point and broadcast channels as ideal functionalities $\mathcal{F}_{\mathrm{P2P}}$ and $\mathcal{F}_{\mathrm{BC}}$, respectively. For instance, a secure point-to-point channel ensures that an adversary cannot eavesdrop on the transmitted information or tamper with it to deceive either party with invalid data. In a secure broadcast channel, an adversary is prevented from intercepting a broadcasted message and modifying it to transmit an incorrect version. For both types of communication channels, the functionalities can be securely implemented in the PKI model.

Secondly, another important aspect that underpins secure communication is the *synchronicity* of the communication network. Generally, there are two types of networks:

1. **Synchronous network**: All parties send messages sequentially, synchronized by a global clock.

2. **Asynchronous network**: All parties send messages at arbitrary times.

An example of a protocol for secure broadcasting over a synchronous network is the Dolev-Strong protocol [DS83]. We also note that protocols secure under sequential composition require a synchronous network, whereas UC-secure protocols can operate in an asynchronous network.

Adversary Models

Finally, in addition to modeling security with respect to the adversary's power, it is important to consider the adversary's corruption capabilities when more than two parties are involved. This capability to corrupt parties is represented by the set of all subsets of parties that an adversary can corrupt, commonly referred to as the *adversary structure*.

Definition 3.3.6 (Adversary structure)**.** Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set parties. An adversary structure $\mathcal{Q}$ is a collection of subsets of the set of all parties, i.e., $\mathcal{Q} \subseteq 2^{\mathcal{P}}$. Additionally, the structure $\mathcal{Q}$ must have the monotone property, i.e., if a set $A \in \mathcal{Q}$, then all subsets of A are also in $\mathcal{Q}$.

A particular class of adversary structures are those where an adversary can corrupt any subset of parties up to some threshold t, called a *threshold structure*. Threshold structures are also typically also called t-out-of-n or (n, t)-threshold structures. The three most common types of threshold structures are the following:

1. **Two-thirds honest majority** ($t < n/3$): In this structure, the adversary can corrupt less than one-third of the total number of parties. A well-known result is that all protocols achieving active security in the IT-secure setting require this structure [BGW88, CCD88].

2. **Honest majority** ($t < n/2$): In this case, the adversary can corrupt less than half of the total number of parties. It is known that *full* active security is achievable only with this structure [GMW87].

3. **Dishonest majority** ($t < n$): Here, the adversary can corrupt any number of parties up to just below the total number of parties. In this situation, active security with abort is achievable.

3.3.2 Fundamental Techniques

As discussed in Chapter 1, the two primary categories of MPC protocols are general-purpose protocols and special-purpose protocols.

General-Purpose Protocol Techniques

In a general-purpose protocol, any function f is represented as a circuit. Traditionally, these circuits are boolean circuits, but they can also be generalized to *arithmetic circuits*.

Definition 3.3.7 (Arithmetic circuit)**.** An arithmetic circuit C with variables $x_1, \ldots, x_n \in \mathbb{Z}_p$ is a directed acyclic graph defined as follows:

- All vertices v with indegree $\deg^-(v) = 0$ and outdegree $\deg^+(v) > 0$ are labeled by input variables or constants, which are called wires.
- All other vertices v have indegree $\deg^-(v) = 2$ and outdegree $\deg^+(v) \leq 1$ and are labeled either by $+$ or $\times$, which are called addition and multiplication gates respectively.

It is easy to see that an arithmetic circuit over $\mathbb{Z}_2$ is equivalent to a boolean circuit with the circuit gates XOR $\oplus$ and AND $\wedge$.

Moreover, general-purpose protocols can be further subdivided into two more categories based on the number of parties that it supports. In the case of two-party protocols, a boolean circuit can be securely computed by converting it into a *garbled circuit* (GC). Without going into detail, this approach employs a symmetric-key encryption scheme to encrypt each gate and utilizes an OT protocol to securely transfer encrypted inputs. In the multi-party case, a boolean circuit or arithmetic circuit can be jointly computed through a technique called *secret sharing*. The archetypical example is the general-purpose protocol from [GMW87], known as the GMW protocol, as described in Figure 3.5. The protocol involves each party

Protocol Π_{GMW}

Input: Each party P_i provides its private input $x_i \in \{0, 1\}$
Output: All parties learn the output y of the boolean circuit $C(x_1, x_2, \ldots, x_n)$.
Parameters: A boolean circuit C composed of AND and XOR gates.

1. **Input Sharing:** Each party P_i additively secret shares its input x_i with everyone.
2. **XOR Gate Evaluation:** Each party P_i locally evaluates $c_i = a_i \oplus b_i$.
3. **AND Gate Evaluation:** For each AND gate:
   (a) Party P_i wants to compute $c_i = a_i \wedge b_i + \sum_{i \neq j}(a_i \wedge b_j \oplus a_j \wedge b_i)$
   (b) All pairs of parties P_i and P_j compute $a_i \wedge b_j \oplus a_j \wedge b_i$ through $\binom{4}{1}$-OT
4. **Output Reconstruction:** Once all gates are evaluated, the parties reconstruct the output y from their shares.

Figure 3.5: The GMW protocol (simplified).

secret-sharing their bit with all other parties, evaluating the circuit using the received shares, and ultimately reconstructing the final output from the resulting output shares.

Apart from its usage in general-purpose protocols, secret sharing remains one of the most prevalent techniques in MPC, including special-purpose protocols.

Secret Sharing

We begin with a general definition of a (t, n)-*threshold secret sharing* (TSS) scheme:

Definition 3.3.8 (Threshold secret sharing scheme)**.** A (t, n)-threshold secret sharing scheme is a pair of algorithms $(\mathrm{Share}, \mathrm{Reconstruct})$, which are defined as follows:

- $\mathrm{pp} \leftarrow \mathrm{Gen}(1^\kappa)$: A PPT algorithm that takes a statistical parameter κ and outputs public parameters pp.
- $(s_i)_{i=1}^n \leftarrow \mathrm{Share}_{\mathrm{pp}}(m, n, t)$: A PPT algorithm that takes a message m, the number of shares n, and a threshold t, and outputs n shares $(s_i)_{i=1}^n$ such that any t or more shares can reconstruct the secret.
- $m \leftarrow \mathrm{Reconstruct}_{\mathrm{pp}}(\{s_i\}_{i\in S})$: A deterministic algorithm that takes a subset S of at least t shares and outputs the secret m.

A particular scheme that we already encountered in Figure 3.5 is the (n, n)-threshold *additive secret sharing* scheme, which is constructed as follows:

Construction 3.3.1 (Additive (n, n)-TSS scheme)**.**

$\mathrm{Gen}(n)$	$\mathrm{Share}_{\mathrm{pp}}(m, n)$	$\mathrm{Reconstruct}_{\mathrm{pp}}(\{s_i \mid i \in [n]\})$
pick any prime $p > n$	$s_1, s_2, \ldots, s_{n-1} \in_{\mathrm{R}} \mathbb{Z}_p$	**return** $m \leftarrow \sum_{\ell=1}^{n} s_i \pmod p$
return $\mathrm{pp} \leftarrow p$	$s_n \leftarrow m - \sum_{i=1}^{n-1} s_i \pmod p$	
	return $(s_1, s_2, \ldots, s_n)$	

When some errors are allowed, or not all shares are needed for reconstruction, a popular choice is *Shamir's secret sharing* scheme, which is constructed as follows:

Construction 3.3.2 (Shamir's (t, n)-TSS scheme [Sha79])**.** Given $I \subseteq [n]$ with $|I| = t$.

$\mathrm{Gen}(n)$	$\mathrm{Share}_{\mathrm{pp}}(m, n, t)$	$\mathrm{Reconstruct}_{\mathrm{pp}}(\{s_i \mid i \in I\})$
pick any prime $p > n$	$f \in_{\mathrm{R}} \mathbb{Z}_p[X]$ where	$m \leftarrow \sum_{i\in I} s_i \prod_{j\in I\setminus\{i\}} \frac{j}{j-i} \pmod p$
return $\mathrm{pp} \leftarrow p$	$\deg f < t$ and $f(0) = m$	**return** m
	for $i = 1$ to n do	
	$s_i \leftarrow f(i) \pmod p$	
	return $(s_1, s_2, \ldots, s_n)$	

The advantage of either type of construction is that it offers *perfect* secrecy, i.e., any two random shares are perfectly indistinguishable from one another. This consequently makes secret sharing one of the primary building blocks in the construction of IT-secure MPC protocols.

3.4 Zero-Knowledge Proofs

In this section, we will introduce the concept of *zero-knowledge proofs* (ZKPs), which are fundamental to the GMW paradigm. We will start by describing the notion of a proof of knowledge and describe how it can be made into a ZKP. Next, we will look at a concrete example of a ZKP and how it can be made non-interactive. Finally, we will describe various types of practical non-interactive proof systems and their properties.

3.4.1 Proof of Knowledge

We begin with the important definition of an NP-relation:

Definition 3.4.1 (NP-relation)**.** Let $R \subseteq V \times W$ be a binary relation. R is said to be an NP-relation if the language $L_R = \{x \mid \exists y \text{ such that } (x, y) \in R\} \in \mathbf{NP}$. Moreover, we denote the relation by $(x; y) \in R$ and say that y is a witness for the problem instance x.

In later sections, we will refer to NP-relations as ZK-statements.

Moreover, we adopt the following naming conventions:

- P and V are the (honest) prover and (honest) verifier, respectively.
- P^* and V^* are the malicious prover and malicious verifier, respectively.
- (P, V) denotes the interaction between P and V, resulting in success (1) or failure (0).

From this, a simple NP proof system, called an *interactive proof system*, can be defined as follows:

Definition 3.4.2 (Interactive proof system [GMR89])**.** An interactive proof system for an NP-relation R is a pair of PPT protocols (P, V) that satisfies the following conditions:

1. **Completeness**: If $x \in L_R$ then $\Pr[(P, V)(x) = 0] \leq \text{negl}(|x|)$
2. **Soundness**: If $x \notin L_R$ then for any prover P^*, $\Pr[(P^*, V)(x) = 1] \leq \text{negl}(|x|)$

Essentially, an interactive proof system can be *any* protocol between a prover and verifier where the prover can convince the verifier of the truth of some statement (completeness), but cannot convince the verifier of a false statement (soundness).

A *proof of knowledge* (PoK) is a special type of interactive proof system where the prover demonstrates not only the validity of the statement but *also* that it "knows" the witness to the statement. Formally, this is defined as follows:

Definition 3.4.3 (Proof of knowledge [BG92])**.** A proof of knowledge with knowledge error $\kappa(\cdot)$ is an interactive proof system (P, V) for an NP-relation R that additionally satisfies the following condition:

Knowledge Soundness: For every prover P^* and any $x \in L_R$, there exists a PPT extractor E such that

$$\Pr[(x; y) \in R \mid y \leftarrow E(x)] \geq \Pr[(P^*, V)(x) = 1] - \kappa(|x|).$$

This means that if a prover can convince a verifier of the truth of a statement, then with the help of an extractor, the prover must indeed know the witness to the statement.

Finally, both interactive proof systems and PoKs can be extended into ZKPs if they satisfy the *zero-knowledge* (ZK) property. This property is defined using the simulation paradigm, similar to the passive security model for MPC protocols.

Definition 3.4.4 (Zero-knowledge proof)**.** A zero-knowledge proof is an interactive proof system (P, V) for an NP-relation R that additionally satisfies the following condition:

Zero-Knowledge: For every PPT verifier V^*, there exists a PPT simulator $\mathcal{S}$ such that

$$\{S(x)\}_{x \in L_R} \stackrel{c}{\equiv} \{\text{view}_{V^*}((V^*, P)(x))\}_{x \in L_R}.$$

Essentially, the view of a malicious verifier should be indistinguishable from the view generated by a simulator, ensuring that the verifier learns nothing beyond the validity of the statement.

3.4.2 Interactive and Non-Interactive Proofs

There are many toy examples that illustrate the workings of a ZKP. In fact, an important theorem due to [GMW86] states that a ZKP can be constructed for *any* NP problem. This was demonstrated by constructing a simple ZKP for the NP-complete *three-coloring problem*, where the prover convinces the verifier that it knows a valid three-coloring of the vertices of a planar graph without revealing the actual coloring. Roughly speaking, a ZKP of 3-colorability involves a commitment scheme where the prover commits to a randomly permuted 3-colored graph and the verifier checks the opening of the revealed colors of random edges. By demonstrating that any NP problem can be reduced to the three coloring problem in polynomial time using structures called *gadgets*, it is shown that any NP problem decider can be efficiently emulated by a ZKP.

Interactive Zero-Knowledge Proofs of Knowledge

Apart from being a ZKP, the protocol for 3-colorability is also a *zero-knowledge proof of knowledge* (ZKPoK). While this protocol can be used to obtain a ZKPoK for any NP-relation, it is rather inefficient: a single proof round only verifies the correctness of one edge, requiring multiple repetitions for the verifier to gain confidence in the proof. However, there are more efficient proof techniques. For example, Schnorr's identification protocol [Sch91], as described in Figure 3.6, is a ZKPoK of a discrete logarithm exponent, requiring just three moves. In this case, the verifier's confidence increases proportionally to the size of

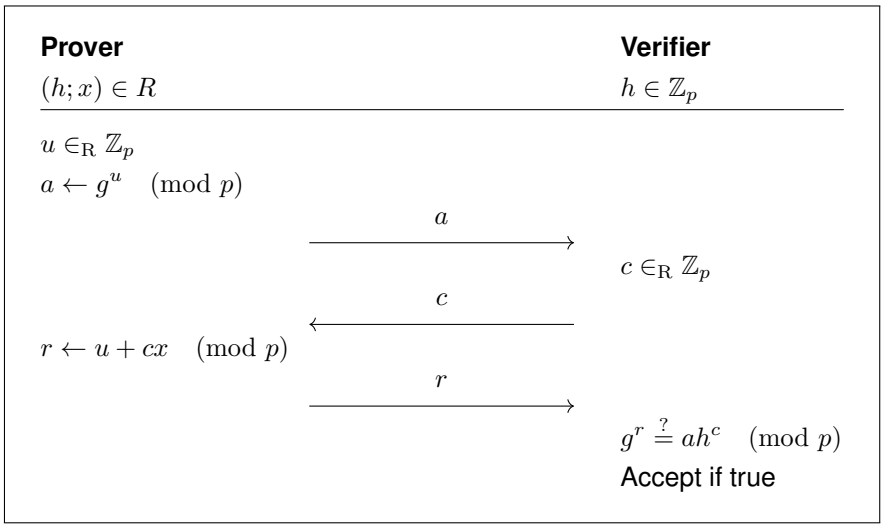


Figure 3.6: Schnorr's identification protocol.

the prime field. More generally, this 3-move protocol belongs to a wider class of ZKPoKs called Σ-protocols, which consist of an *announcement*, *challenge*, and *response* move. It can therefore be said that the protocol in Figure 3.6 is a Σ-protocol for ZK-statement $R = \{(h; x) \in \mathbb{Z}_p \times \mathbb{Z}_p \mid h = g^x \text{ with } g \in \mathbb{Z}_p^*\}$.

Non-Interactive Zero-Knowledge Proofs of Knowledge

For large-scale cryptographic protocols, as will also be the case with the active-security compiler, it is necessary to scale up a ZKP such that a prover can convince multiple verifiers

at once. By constructing a *non-interactive zero-knowledge* (NIZK) proof, a prover can eliminate the need for repeated interactions with verifiers and, additionally, publish a proof that anyone can independently verify offline. It turns out that NIZK proofs can be obtained from interactive ZK proofs through the *Fiat-Shamir transformation* [FS87]. It works by outsourcing the challenge to a random oracle, which in practice is obtained from a cryptographic hash function applied to an announcement and problem instance. By doing so, the protocol can be described as a cryptographic scheme that is secure in the RO model. For instance, Schnorr's identification protocol from Figure 3.6 can be transformed into a NIZK proof using the Fiat-Shamir transformation, as shown in Figure 3.7, resulting in what is commonly known as Schnorr's signature scheme. The transformation can be applied to any Σ-protocol

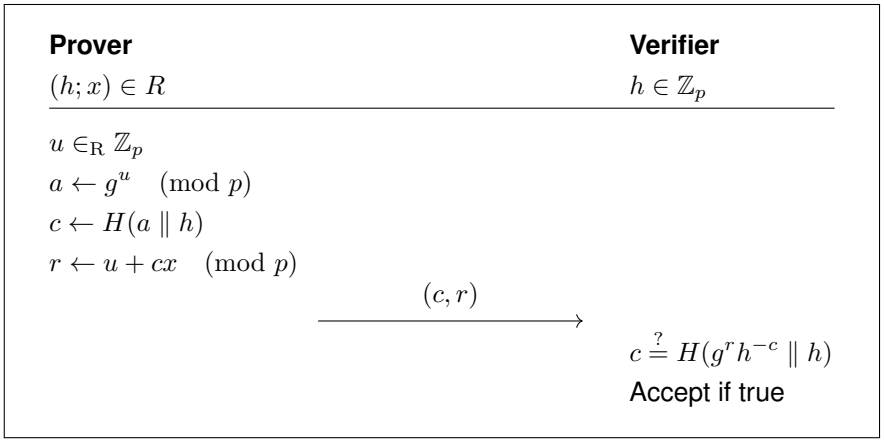


Figure 3.7: Fiat-Shamir transformation of Schnorr's identification protocol.

for ZK-statement R to obtain a NIZK proof, known in this context as a Σ-proof.

3.4.3 Practical Systems

Aside from the simple constructions we just discussed, practical and efficient ZKP protocols, especially NIZK proofs, tend to be significantly more complex. Although we will not delve into detailed constructions, we will explore the broader context and note distinctions among various proof systems. Moreover, our particular focus will be on NIZK schemes, which can be defined in the PKI model as follows:

Definition 3.4.5 (NIZK scheme)**.** A non-interactive zero-knowledge scheme is a triple of algorithms $(\mathrm{Setup}, \mathrm{Prove}, \mathrm{Verify})$, which are defined as follows:

- $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{Setup}(1^\lambda, R)$: A PPT algorithm that takes a security parameter λ and a ZK-statement R, and outputs public key pk and secret key sk.
- $\pi \leftarrow \mathrm{Prove}_{\mathrm{sk}}((x; w), R)$: A PPT algorithm that takes a problem instance x, a witness w, and a ZK-statement R, and outputs a proof π.
- $\{0, 1\} \leftarrow \mathrm{Verify}_{\mathrm{pk}}(x, \pi, R)$: A deterministic algorithm that takes a problem instance x, a proof π, and a ZK-statement R, and outputs 1 if the proof π is valid for x, and 0 otherwise.

Alternatively, in the CRS model, the public/secret key pair $(\mathrm{pk}, \mathrm{sk})$ may be replaced by public parameters pp. Some features that set apart NIZK schemes are:

1. **Setup**: Typically, schemes rely on a *trusted* setup in the CRS model to introduce the necessary randomness for non-interactive challenge generation corresponding to a specific ZK-statement R. In contrast, a *transparent* setup in the RO model eliminates

this requirement as it obtains randomness through a cryptographic hash function. In some cases, there is an intermediate solution where a setup is required only once for any ZK-statement up to a specified length, referred to as a *universal* setup.

2. **Succinctness**: The size and verification time of a proof differ depending on the system. A proof is *succinct* if its size is considerably smaller than that of the witness, and if the time required to verify the proof is significantly less than the time needed to manually check the witness.
3. **Purpose**: General-purpose proofs are designed for arbitrary NP problems created from arithmetic circuit gadgets, whereas special-purpose proofs are designed for specific NP problems.

These are just a few of the numerous properties used to compare different types of NIZK schemes. In relation to succinctness, one typically also considers the computational complexities for both the prover and verifier. However, describing these complexities for an entire class of NIZK schemes is not as straightforward.

Arguments of Knowledge

Practical NIZK schemes often achieve greater efficiency by creating a less stringent type of proof known as an *argument*. Conceptually, an argument of knowledge is defined as a prover-verifier system where soundness is relaxed to hold against PPT provers, while proofs traditionally hold even against unbounded provers [BCC88]. Common types of practical argument of knowledge systems are *succinct non-interactive argument of knowledge* (SNARK) schemes [PHGR13, Gro16, Set20] and *scalable transparent non-interactive argument of knowledge* (STARK) schemes [BBHR19].

Common Classes of NIZK Schemes

Other types of NIZK schemes include Bulletproofs [BBB$^+$18, EKRN24] and its related cousin Compressed Σ-protocols [AC20], as well as schemes based on *MPC-in-the-head* (MPCitH) [IKOS07, GMO16, AHIV17]. We provide an overview of various types in Table 3.4.

Table 3.4: High-level comparison of NIZK scheme types. (● = succinct, ◐ = partially succinct, ○ = not succinct).

Type	Setup	Succinctness	Security Model	Purpose
ZK-SNARKs	trusted*	●	CRS	general
ZK-STARKs	transparent	◐	RO	general
MPCitH	transparent	◐	RO	general
Σ-proofs	transparent	○	RO	special & general
Bulletproofs	transparent	○	RO	special & general

* Some SNARKs have a universal setup; transparent setups are uncommon.

Miscellaneous NIZK Schemes

Without going into detail, we will outline two additional types of NIZK schemes relevant to MPC. The first is *incrementally-verifiable computing* [Val08], which involves proofs that can be recursively composed with previously verified proofs, thereby minimizing the proof size. A notable application of it in MPC is *proof-carrying data* [BCL$^+$21]. The second type is *commit-and-prove* schemes [CLOS02], where a prover makes statements about values and their commitments. An example of this is the *LegoSNARK* framework [CFQ19].

4

Active-Security Compiler Design

Up to this point, we have examined various active-security compilers and discussed relevant theoretical concepts. In Chapter 2, we carried out a comprehensive literature review, exploring the landscape of different active-security compilers and their properties. This review identified that a compiler based on the GMW paradigm [GMW87] is particularly well-suited for compiling arbitrary passively secure special-purpose protocols. Following this, Chapter 3 covered the essential cryptographic primitives, security definitions, and a range of ZKPs that are relevant to the GMW paradigm.

In this chapter, we will address research question RQ1 from Section 1.2: In Section 4.1, we will first outline the steps necessary for designing the active-security compiler. Next, we will delve deeper into the GMW paradigm, gathering essential insights for our design. The active-security compiler design will be formalized as a protocol construction in Section 4.2. Finally, we discuss practical constructions of the cryptographic primitives required by the compiler in Section 4.3.

4.1 Conceptual Considerations

At a high level, the original GMW compiler transforms the security of any protocol from passive to active using ZKPs. Conceptually, this approach works by assuming the existence of a ZKP system that can process ZK-statements $R = \{(y; x, T) \mid \mathsf{NextMsg}_{\Pi}(x, T) = y\}$. Here, T represents the transcript of all previously received messages and inputs, x is a local input, and y is the resulting message.

Defining such a ZK-statement is non-trivial. Since any MPC protocol is probabilistic, the next-message function is also probabilistic, making proofs of knowledge ambiguous unless all randomness is well-defined. Moreover, the statement must be an appropriate NP-relation, meaning it should relate a problem instance to a witness. Finally, a suitable ZKP system must be chosen for that kind of ZK-statement.

To correctly address RQ1, we consider the following sub-questions:

RQ1: *How can zero-knowledge proof systems be used to obtain active security from a passively secure protocol?*

a. *What kind of MPC protocols admit a ZKP?*

b. *What ZK-statements are required by the compiler?*

c. *What ZKP systems can prove and verify the required ZK-statements?*

We will address RQ1a in the remainder of this section. Sub-question RQ1b will be covered in Section 4.2, and RQ1c in Section 4.3.

4.1.1 Compilers in the GMW Paradigm

We begin by recalling from Section 2.1.1 that the GMW compiler consists of two phases:

1. **Engagement**[1]: All parties commit to their input and commit to unbiased randomness via coin tossing.
2. **Emulation**: All parties run the passively secure protocol with a ZKP that the messages they send is correct with respect to the committed information and previous messages.

If all parties, including potentially malicious and at least one trusted party, follow the passively secure protocol using the two steps mentioned above, the protocol achieves active security [GMW87]. However, the current formulation is quite ambiguous. Therefore, we will examine a more precise formulation and construction of the GMW compiler, as described in [Gol04].

Ideal Functionalities

The first step to constructing the protocol is defining its ideal functionality, or as is the case with the GMW compiler, the ideal functionalities of its sub-protocols. [Gol04] defines three n-party functionalities, namely, for *input commitment*, *augmented coin tossing*, and *authenticated computation*:

Definition 4.1.1 (Multi-party functionalities, adapted from [Gol04, Section 7.5.4])**.** Given a set of parties $\{P_1, \ldots, P_n\}$, we represent an ideal n-party functionality by the mapping $(x_i, x_{j_1}, \ldots, x_{j_{n-1}}) \mapsto (y_i, y_{j_1}, \ldots, y_{j_{n-1}})$, where x_i is the input of P_i and y_i is the output it receives. The ideal functionalities of the GMW compiler are defined as follows:

1. **Input commitment**: For $x_i \in \{0,1\}^s$ and $r_i \in \{0,1\}^{s \cdot k}$ for some s and k, the functionality is described as follows:

$$\mathcal{F}_{\text{COMMIT}} : (x_i, \emptyset, \ldots, \emptyset) \mapsto (r_{x_i}, \text{Commit}(x_i, r_{x_i}), \ldots, \text{Commit}(x_i, r_{x_i}))\,.$$

2. **Augmented coin tossing**: For $u_i \in \{0,1\}^s$ and $r_i \in \{0,1\}^{s \cdot k}$ for some s and k, the functionality is described as follows:

$$\mathcal{F}_{\text{COIN}} : (\emptyset, \emptyset, \ldots, \emptyset) \mapsto ((u_i, r_{u_i}), \text{Commit}(u_i, r_{u_i}), \ldots, \text{Commit}(u_i, r_{u_i}))\,.$$

3. **Authenticated computation**: For any input and given an authentication function h and function to be computed f, the functionality is described as follows:

$$\mathcal{F}_{\text{AUTH}} : (x_i, x_{j_1}, \ldots, x_{j_{n-1}}) \mapsto \begin{cases} (\emptyset, f(x_i), \ldots, f(x_i)) & \text{if } h(x_i) = x_{j_i} = \cdots = x_{j_{n-1}} \\ (\emptyset, \perp, \ldots, \perp) & \text{otherwise.} \end{cases}$$

From the three functionalities, $\mathcal{F}_{\text{COMMIT}}$ and $\mathcal{F}_{\text{COIN}}$ make up the engagement phase, while the $\mathcal{F}_{\text{AUTH}}$ one describes the emulation phase, granted that the authentication function h is bijective. The reason that $\mathcal{F}_{\text{AUTH}}$ qualifies as a ZKPoK is not immediately obvious, but it becomes clear due to the following requirements[2]:

1. **Completeness**: All receiving parties must unanimously accept a valid input.

[1] In [GMW87] this is described as the *engagement protocol*, which forces parties to engage in the passively secure protocol according to their claimed inputs.

[2] See the proof of [Gol04, Prop. 7.5.25] for an explanation of why these requirements are essential for authenticated computation.

2. **Knowledge Soundness**: An invalid input must be unanimously rejected. If an input is accepted, then the receiving parties are convinced that party P_i knows the input.
3. **Zero-Knowledgeness**: The input must remain private, i.e., receiving parties learn $f(x_i)$ and not x_i.

Compiler Construction

To understand how active security with abort is achieved for any passively-secure protocol using the functionalities from Definition 4.1.1, we illustrate their sequential composition in Figure 4.1, which is a construction of the original GMW compiler.

Protocol $\Pi_{\text{GMW-COMPILER}}$

Inputs: Parties $P_1, \ldots, P_n$ hold inputs $x_1, \ldots, x_n \in \{0,1\}^s$ where s is fixed.
Outputs: Each party P_i learns the corresponding result from protocol Π.
Primitives: Functionalities $\mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{COIN}}, \mathcal{F}_{\text{AUTH}}$, a passively secure protocol Π, and a Commit algorithm (Definition 3.2.7) with public parameters pp.

Input-Commitment Phase:

1. For each $i \in [n]$, party P_i invokes $\mathcal{F}_{\text{COMMIT}}$ with input x_i, obtaining output r_{x_i}, while other parties obtain $c_{x_i} \leftarrow \text{Commit}_{\text{pp}}(x_i, r_{x_i})$.

Coin-Tossing Phase:

2. For each $i \in [n]$, party P_i invokes $\mathcal{F}_{\text{COIN}}$ and obtains (u_i, r_{u_i}) while every other party P_j obtains $c_{u_i} \leftarrow \text{Commit}_{\text{pp}}(u_i, r_{u_i})$.

Protocol-Emulation Phase:

3. All parties define the authentication function h and computation function f:
   - Let $h((x, r_x), (u, r_u), T) := (\text{Commit}_{\text{pp}}(x, r_x), \text{Commit}_{\text{pp}}(u, r_u), T)$.
   - Let $f((x, r_x), (u, r_u), T) := \text{NextMsg}_{\Pi}(x, r, T) = y$.

   In both cases, (x, r_x) is from the input-commitment phase, (u, r_u) is from the coin-generation phase, and T is the sequence of messages received earlier.
4. All parties invoke the $\mathcal{F}_{\text{AUTH}}$ functionality to emulate each step of protocol Π:
   - The sending party P_i sends $((x_i, r_{x_i}), (u_i, r_{u_i}), T)$ whereas the receiving parties P_j where $j \neq i$ send (c_{x_i}, r_{u_i}, T)
   - Each receiving party gets either $y_i \leftarrow f((x_i, r_{x_i}), (u_i, r_{u_i}), T)$ or aborts with $\perp$.
   - All parties update their transcript $T \leftarrow T \cup \{y\}$.

Figure 4.1: The GMW compiler, adapted from [Gol04, Constr. 7.5.32].

The compiler can be summarized as follows: The parties initially complete the engagement phase by running steps 1 and 2. By having the authentication function correspond to the input and unbiased randomness commitments in step 3, they emulate the protocol in the active-security-with-abort model by running step 4.

4.1.2 On the Security of the GMW Compiler

The three previously mentioned functionalities can be securely implemented in the active-security-with-abort model (Definition 3.3.5), as described in the following lemma:

Lemma 4.1.1 ([Gol04, Props. 7.5.31, 7.5.30, 7.5.27])**.** Assuming the existence of ETDPs, the ideal functionalities $\mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{COIN}}, \mathcal{F}_{\text{AUTH}}$ can be securely realized in the active-security-with-abort model.

Roughly speaking[3], since the sequential composition of protocols for the three functionalities is secure, and since the universal functionality can be securely emulated, any passively secure MPC protocol can be transformed into a protocol with active security with abort. An example of a protocol that implements the universal functionality is the general-purpose GMW protocol from Figure 3.5.

In the two-party case, these functionalities assume the weaker assumption of the existence of OWFs [Gol04, Prop. 7.4.22]. It turns out that ETDPs are necessary to securely emulate point-to-point communication over a broadcast channel.

Theorem 4.1.1 ([Gol04, Prop. 7.5.16])**.** Assuming the existence of ETDPs, any n-party functionality in the point-to-point communication model can be emulated in the broadcast communication model.

While it is possible to compile any passively secure protocol in the point-to-point communication model by emulating private communication over a broadcast channel, it is necessary for authenticated computation to be carried out over a broadcast channel. This ensures that all parties unanimously accept or reject an input. Furthermore, if the protocols for $\mathcal{F}_{\text{COMMIT}}$ and $\mathcal{F}_{\text{COIN}}$ are not inherently actively secure, their computation must also be proved in zero-knowledge.

4.2 Conceptual Design

In this section, we bridge the gap between the conceptual GMW compiler and our practical active-security compiler. We will construct a GMW-style compiler with a well-defined ZK-provable next-message function, and provide efficient constructions for the underlying functionalities. For simplicity, we consider all elements to be in the prime field $\mathbb{Z}_p$.

4.2.1 Sequential Decomposability

In order to concretely define the next-message function, we must have a way to describe a passively secure protocol's internal structure. One approach to this is to describe a protocol as a sequential composition of sub-protocols. More specifically, we would like to decompose a protocol into a sequential composition.

Definition 4.2.1 (Sequential k-decomposition)**.** A passively secure n-party protocol Π_f that computes the functionality f is sequentially k-decomposable if there exist sub-protocols for the functionalities $f_1, \ldots, f_k$, such that

$$\Pi_f = \Pi_{f_k} \diamond \Pi_{f_{k-1}} \diamond \cdots \diamond \Pi_{f_1},$$

where $\diamond$ denotes sequential composition. We denote the i-th sub-protocol by $\Pi_f^i := \Pi_{f_i}$.

Since the transformation outlined in the GMW compiler in Figure 4.1 is secure under sequential composability in the broadcast communication model, it admits MPC protocols in the *synchronous* communication model. We therefore claim that any k-round protocol in this model admits a sequential k-decomposition.

Proposition 4.2.1. For any k-round n-party protocol Π_f in the synchronous model, there exists a k-decomposition $\Pi_f = \Pi^k \diamond \Pi^{k-1} \diamond \cdots \diamond \Pi^1$.

[3] An accurate explanation for why the compiled protocols are actively secure can be found in the proof of [Gol04, Thrm. 7.5.33].

Proof. In the synchronous model, a protocol Π_f ensures message delivery in round r before starting round $r+1$. Each round's computations form a sub-protocol Π^r with inputs from the previous round and outputs to the next. Thus, Π_f is a sequential composition of k sub-protocols, i.e., $\Pi_f = \Pi^k \diamond \Pi^{k-1} \diamond \cdots \diamond \Pi^1$. □

In a k-decomposition of a k-round protocol, messages are passed only at the end of each sub-protocol. Thus, we define the next-message function as $\text{NextMsg}_\Pi(x, r, T_{<\ell}) := \Pi^\ell(x, r)$, where $T_{<\ell}$ is the sequence of messages received up to round ℓ, and $\Pi(x, r)$ is abuse of notation to indicate that the protocol is executed locally. Since the protocol takes its randomness as input, it can be considered a deterministic function.

4.2.2 ZK-Statements for Authenticated Computation

We can more concisely express the authenticated-computation functionality from Definition 4.1.1 as a problem for a ZK-statement. First, we consider the case where authentication does not take place[4], i.e., where a party simply wants to prove that it knows the output to some secure protocol. The key word here is *secure* protocol, as each party supplies the function with randomness, making it an NP problem to find the function's pre-image. We can describe this as a ZK-statement for *protocol evaluation*.

Definition 4.2.2 (ZK-Statement for Protocol Evaluation)**.** Let Π_f be a deterministic protocol for a function $f : \mathbb{Z}_p^\ell \times \mathbb{Z}_p^m \to \mathbb{Z}_p^n : (\mathbf{x}, \mathbf{u}) \mapsto \mathbf{y}$. The ZK-statement for protocol evaluation is defined as

$$R_{eval}(\Pi_f) = \{(\mathbf{y} \; ; \; \mathbf{x}, \mathbf{u}) \; : \; f(\mathbf{x}, \mathbf{u}) = \mathbf{y}\}.$$

We can similarly define a ZK-statement for authenticated computation, which we will call *protocol authentication*. We will consider the usual case where the protocol output needs to be authenticated and made public, as well as the case where the output should be authenticated but remain private.

Definition 4.2.3 (ZK-Statements for Protocol Authentication)**.** Let Π_f be a deterministic protocol for a function $f : \mathbb{Z}_p^\ell \times \mathbb{Z}_p^m \to \mathbb{Z}_p^n : (\mathbf{x}, \mathbf{u}) \mapsto \mathbf{y}$, where the inputs are private, and the output may be either public or private. Let Commit be from Definition 3.2.7 with public parameters pp. The ZK-statements are defined as follows:

Protocol Authentication with Public Output:

$$R^+_{auth}(\Pi_f) = \left\{ (\mathbf{c_x}, \mathbf{c_u}, \mathbf{y} \; ; \; \mathbf{r_x}, \mathbf{r_u}, \mathbf{x}, \mathbf{u}) \; : \; \begin{array}{r} \forall i \in [\ell] \, (\text{Commit}_{\text{pp}}(x_i, r_{x_i}) = c_{x_i}) \\ \wedge \forall i \in [m] \, (\text{Commit}_{\text{pp}}(u_i, r_{u_i}) = c_{u_i}) \\ \wedge f(\mathbf{x}, \mathbf{u}) = \mathbf{y} \end{array} \right\}.$$

Protocol Authentication with Private Output:

$$R^-_{auth}(\Pi_f) = \left\{ (\mathbf{c_x}, \mathbf{c_u}, \mathbf{c_y} \; ; \; \mathbf{r_x}, \mathbf{r_u}, \mathbf{r_y}, \mathbf{x}, \mathbf{u}) \; : \; \begin{array}{r} \forall i \in [\ell] \, (\text{Commit}_{\text{pp}}(x_i, r_{x_i}) = c_{x_i}) \\ \wedge \forall i \in [m] \, (\text{Commit}_{\text{pp}}(u_i, r_{u_i}) = c_{u_i}) \\ \wedge \forall i \in [n] \, (\text{Commit}_{\text{pp}}(f_i(\mathbf{x}, \mathbf{u}), r_{y_i}) = c_{y_i}) \end{array} \right\}.$$

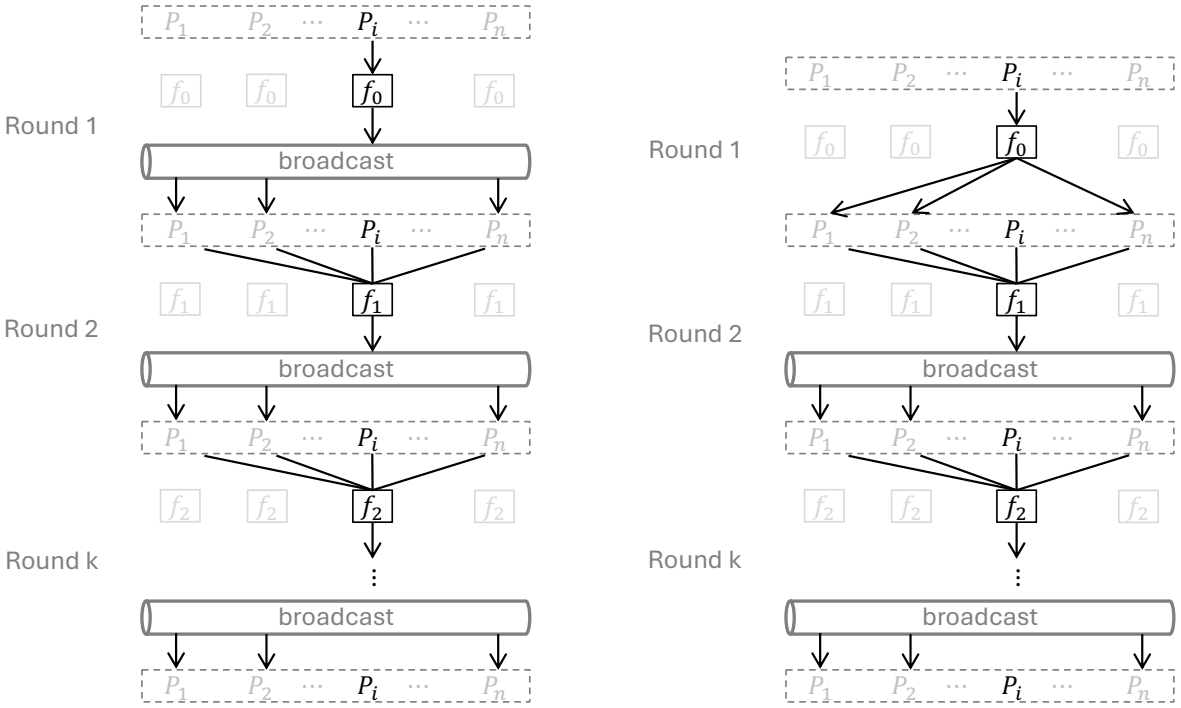
[4] In [Gol04] this is called the *image-transmission* functionality.

4.2.3 Active-Security Compiler

We will now use our definition of sequential decomposability and the ZK-statements for protocol authentication to construct the active-security compiler. Before doing so, we will first outline precisely what types of MPC protocols it can be applied to.

Communication Model

We consider MPC protocols in the synchronous broadcast model to avoid having to emulate point-to-point channels. However, by our definition of protocol authentication with *private* output, we will also allow for initial point-to-point communication. Broadly speaking, the two communication models can be visualized as shown in Figure 4.2.



(a) Broadcast-only model

(b) Initial point-to-point model

Figure 4.2: Communication models supported by the active-security compiler.

Ideal Functionalities and Constructions

We will modify $\mathcal{F}_{\text{COMMIT}}$ and $\mathcal{F}_{\text{COIN}}$ from Definition 4.1.1 such that input and randomness have equal lengths and replace $\mathcal{F}_{\text{AUTH}}$ with the ZK functionality from Figure 4.3.

Ideal Functionality $\mathcal{F}_{\text{ZK}}^{R}$

Parameterized by ZK-statement R and interacting with parties $P_1, \ldots, P_n$, where P_i is the prover and all P_j with $j \neq i$ are verifiers, the functionality proceeds as follows:

- Receive ZK-statement $(\mathbf{x}_i; \mathbf{w}_i) \in \mathbb{Z}_p^{N_x} \times \mathbb{Z}_p^{N_w}$ from party P_i.
- Receive instance $\mathbf{x}_j \in \mathbb{Z}_p^{N_x}$ from party P_j for $j \in [n] \setminus \{i\}$.
- If $\mathbf{x}_i = \mathbf{x}_j$ for all $j \in [n] \setminus \{i\}$ and $(\mathbf{x}_i; \mathbf{w}_i) \in R$, send `Valid` to P_j for all $j \in [n] \setminus \{i\}$, otherwise send `Invalid`.

Figure 4.3: The multi-party ZK functionality.

Next, we implement all three functionalities. To ensure that the input and commitment randomness are of equal size, we use the Pedersen commitment scheme to construct $\mathcal{F}_{\text{COMMIT}}$. By the additive homomorphic property of Pedersen, we can efficiently construct $\mathcal{F}_{\text{COIN}}$ in a single round. Finally, we can construct $\mathcal{F}_{\text{ZK}}$ from any NIZK scheme.

Protocol Π_{COMMIT}

Input: Party P_i provides its private input $x \in \mathbb{Z}_p$.
Output: All parties receive commitment $c_x \in \mathbb{Z}_p$, and P_i receives blinding factor $r_x \in \mathbb{Z}_p$.
Primitives: $\text{CS} = (\text{Setup}, \text{Commit}, \text{Open})$ from Construction 3.2.2.
Parameters: Field size p and parameters $\text{pp} \leftarrow \text{CS.Setup}(p)$.

1. Party P_i randomly samples $r \in_{\text{R}} \mathbb{Z}_p$.
2. Party P_i obtains $c_x \leftarrow \text{CS.Commit}_{\text{pp}}(x, r)$ and broadcasts it.

Figure 4.4: The input commitment protocol.

Protocol Π_{COIN}

Input: No input is provided by any party.
Output: Party P_i receives unbiased randomness $u \in \mathbb{Z}_p$ and blinding factor $r \in \mathbb{Z}_p$, and all parties P_j receive commitment $c_u \in \mathbb{Z}_p$.
Primitives: $\text{CS} = (\text{Setup}, \text{Commit}, \text{Open})$ from Construction 3.2.2.
Parameters: Field size p and parameters $\text{pp} \leftarrow \text{CS.Setup}(p)$.

Commitment phase:

1. For $j \in [n]$, party P_j randomly samples $u_j, r_j \in_{\text{R}} \mathbb{Z}_p$.
2. For $j \in [n]$, party P_j obtains $c_{u_j} \leftarrow \text{CS.Commit}_{\text{pp}}(u_j, r_j)$ and broadcasts it.
3. After all parties receive c_{u_j} for $j \in [n]$, they construct $c_u \leftarrow \prod_{j \in [n]} c_{u_i}$.

Sharing phase:

4. For $j \in [n] \setminus \{i\}$, party P_j sends (u_j, r_j) to P_i.
5. Party P_i constructs $u \leftarrow \sum_{j \in [n]} u_j$ and $r \leftarrow \sum_{j \in [n]} r_j$.

Figure 4.5: The augmented coin-tossing protocol.

Protocol Π^R_{ZK}

Input: Party P_i provides ZK-statement $(\mathbf{x}_i; \mathbf{w}_i) \in \mathbb{Z}_p^{N_x} \times \mathbb{Z}_p^{N_w}$ while all other parties P_j provide instance $\mathbf{x}_j \in \mathbb{Z}_p^{N_x}$.
Output: Party P_j receives unbiased randomness $x \in \mathbb{Z}_p^m$ and blinding factors $r \in \mathbb{Z}_p^m$, and all parties P_i receive commitments $c_x \in \mathbb{Z}_p^m$.
Primitives: $\text{NIZK} = (\text{Setup}, \text{Prove}, \text{Verify})$ from Definition 3.4.5.
Parameters: Field size p, ZK-statement R, and parameters $\text{pp} \leftarrow \text{NIZK.Setup}(p, R)$.

1. Party P_i obtains $\text{proof}_i \leftarrow \text{NIZK.Prove}_{\text{pp}}((\mathbf{x}_i; \mathbf{w}_i), R)$ and broadcasts it.
2. For $j \in [n] \setminus \{i\}$, party P_j obtains $V \leftarrow \text{NIZK.Verify}_{\text{pp}}(\mathbf{x}_j, \text{proof}_i, R)$.

Figure 4.6: The multi-party ZK protocol.

With all the functionalities in place, we can construct the GMW-style active-security compiler for protocols in the broadcast-only and initial point-to-point models, shown in Figure 4.7.

Protocol Π_{ACTIVE}

Input: Parties $P_1, \ldots, P_n$ provide private inputs $x_1, \ldots, x_n \in \mathbb{Z}_p$, respectively.
Output: All parties receive $f(x_1, \ldots, x_n) \in \mathbb{Z}_p$.
Primitives: Passively secure protocol $\Pi_f = \Pi_f^{k-1} \diamond \Pi_f^{k-2} \diamond \cdots \diamond \Pi_f^0$, and functionalities $\mathcal{F}_{\text{NIZK}}^R$, $\mathcal{F}_{\text{COIN}}$ and $\mathcal{F}_{\text{COMMIT}}$.
Parameters: Field size p, amount of randomness m, and ZK-statements $R_{auth}^-(\Pi_f^0)$ and $\{R_{auth}^+(\Pi_f^\ell)\}_{\ell=0}^{k-1}$.

Engagement phase:

1. For each $\ell \in [m]$ and $i \in [n]$, all parties interact with $\mathcal{F}_{\text{COIN}}$ and P_i receives randomness $u_{i\ell}$ and blinding factor $r_{u_{i\ell}}$ while all parties receive commitment $c_{u_{i\ell}}$. They each store the outbound vectors $\mathbf{u}_i \leftarrow (u_{i\ell})_{\ell \in [m]}$, $\mathbf{r}_{\mathbf{u}_i} \leftarrow (r_{u_{i\ell}})_{\ell \in [m]}$ and $\mathbf{c}_{\mathbf{u}_i} \leftarrow (c_{u_{i\ell}})_{\ell \in [m]}$.
2. Based on the mode of communication of Π_f^0, do the following:
   (a) If a broadcast channel is used:
      i. For each $i \in [n]$, P_i computes $y_i^0 \leftarrow \Pi_f^0(x_i, \mathbf{u}_i)$ and broadcasts it.
      ii. For each $i \in [n]$, P_i inputs y_i^0 in $\mathcal{F}_{\text{COMMIT}}$ and receives blinding factor $r_{y_i^0}$ while all parties receive commitment $c_{y_i^0}$.
      iii. For each $i \in [n]$, P_i inputs $(c_{y_i^0}, \mathbf{c}_{\mathbf{u}_i}, y_i^0 \ ; \ r_{y_i^0}, \mathbf{r}_{\mathbf{u}_i}, x_i, \mathbf{u}_i)$ in $\mathcal{F}_{\text{ZK}}^{R_{auth}^+(\Pi_f^0)}$ and all parties except P_i receive a verification V and abort with $\perp$ if $V = 0$.
   (b) If point-to-point channels are used:
      i. For each $i \in [n]$, P_i computes $\mathbf{y}_i^0 \leftarrow \Pi_f^0(x_i, \mathbf{u}_i)$.
      ii. For each $i \in [n]$, P_i inputs x_i in $\mathcal{F}_{\text{COMMIT}}$ and receives blinding factor r_{x_i} while all parties receive commitment c_{x_i}.
      iii. For each $i \in [n]$ and $j \in [n]$, P_i inputs y_{ij}^0 in $\mathcal{F}_{\text{COMMIT}}$ and receives blinding factor $r_{y_{ij}^0}$ while all parties receive commitment $c_{y_{ij}^0}$. They each store the outbound vectors $\mathbf{r}_{\mathbf{y}_i^0} \leftarrow (r_{y_{ij}^0})_{j \in [n]}$ and $\mathbf{c}_{\mathbf{y}_i^0} \leftarrow (c_{y_{ij}^0})_{j \in [n]}$.
      iv. For each $i \in [n]$, P_i sends $(y_{ij}^0, r_{y_{ij}^0})$ to P_j for all $j \in [n] \setminus \{i\}$.
      v. For each $i \in [n]$, P_i inputs $(c_{x_i}, \mathbf{c}_{\mathbf{u}_i}, \mathbf{c}_{\mathbf{y}_i^0} \ ; \ r_{x_i}, \mathbf{r}_{\mathbf{u}_i}, \mathbf{r}_{\mathbf{y}_i^0}, x_i, \mathbf{u}_i)$ in $\mathcal{F}_{\text{ZK}}^{R_{auth}^-(\Pi_f^0)}$ and all parties except P_i receive a verification V and abort with $\perp$ if $V = 0$.
3. For each $i \in [n]$, P_i stores the inbound vectors $\mathbf{r}_{\mathbf{y}_i^0} \leftarrow (r_{y_{ji}^0})_{j \in [n]}$, $\mathbf{c}_{\mathbf{y}_i^0} \leftarrow (c_{y_{ji}^0})_{j \in [n]}$.

Emulation phase:

4. For each $\ell \in [k-1]$ and $i \in [n]$, P_i stores the inbound vector $\mathbf{y}_i^{\ell-1} \leftarrow (y_{ji}^{\ell-1})_{j \in [n]}$ and proceeds as follows:
   (a) For each $i \in [n]$, P_i computes $y_i^\ell \leftarrow \Pi_f^\ell(\mathbf{y}_i^{\ell-1}, \mathbf{u}_i)$ and broadcasts it.
   (b) For each $i \in [n]$, P_i inputs $(\mathbf{c}_{\mathbf{y}_i^0}, \mathbf{c}_{\mathbf{u}_i}, y_i^\ell \ ; \ \mathbf{r}_{\mathbf{y}_i^0}, \mathbf{r}_{\mathbf{u}_i}, \mathbf{y}_i^{\ell-1}, \mathbf{u}_i)$ in $\mathcal{F}_{\text{ZK}}^{R_{auth}^+(\Pi_f^\ell)}$ and all parties except P_i receive a verification V and abort with $\perp$ if $V = 0$.

Figure 4.7: The active-security compiler for passively-secure MPC protocols in the broadcast-only and initial point-to-point communication model.

Compared to the original GMW compiler (Figure 4.1), we merge the input-commitment and coin-tossing phases into a single engagement phase. During this phase, we also emulate the initial sub-protocol Π_f^0 using the R_{auth}^- ZK-statement in the point-to-point model or R_{auth}^+ in the broadcast model. We note that the term *outbound* vector refers to values sent by party P_i to all other parties P_j, while *inbound* vectors are values received by P_i from all P_j.

Note on Security

Since our construction mostly aligns with the provably secure original GMW compiler, we will not provide a proof of security of our compiler from Figure 4.7. Instead, we provide an intuition behind the changes we have introduced. First, when a party proves that the private point-to-point messages were correctly computed using R^{-}_{auth}, the receiver does not learn the output directly but instead verifies its correctness against the commitment of the output.

For the individual functionality constructions, their security relies on the primitives used: The protocol Π_{COMMIT} (Figure 4.4) is actively secure due to the Pedersen commitment being secure for inputs and randomness of the same length. Similarly, Π_{COIN} (Figure 4.5) is secure as all receiving parties locally compute the commitment to unbiased randomness of the sending party through the additive homomorphic property. Finally, Π_{ZK} (Figure 4.6) is secure because the ZKP primitive is a NIZK scheme, ensuring that all proving and verification occur locally.

4.3 Practical Components

We described the active-security compiler as a cryptographic protocol Π_{ACTIVE} that relies on the sub-protocols $\Pi_{\mathrm{COMMIT}}, \Pi_{\mathrm{COIN}},$ and Π^{R}_{ZK}. While the cryptographic primitives for Π_{COMMIT} and Π_{COIN} are based on the Pedersen commitment scheme from Construction 3.2.2, the NIZK schemes used in Π^{R}_{ZK} have not yet been addressed. Consequently, the ZK-statements $R^{+}_{auth}(\cdot)$ and $R^{-}_{auth}(\cdot)$ also need to be concretely defined for the corresponding NIZK schemes.

In this section, we will examine the ZKP systems for the active-security compiler, including the concrete construction of ZK-statements, and explore more efficient methods for constructing a ZKP friendly commitment scheme.

4.3.1 Choice of ZKP Systems

Before evaluating suitable ZKP systems, we note that there are two primary computation models based on circuits. The *application-specific integrated circuits* (ASICs) correspond to *non-uniform circuits* specifically defined for fixed-length programs. The *random-access machines* (RAMs) correspond to *uniform circuits* that can emulate any variable-sized program by storing and accessing data from registers via pointers.

Typically, most general-purpose ZKP systems such as SNARKs and Bulletproofs are designed to handle ZK-statements based on non-uniform circuits. However, a small subset, including STARKs, are designed for ZK-statements based on uniform circuits. Given that the fan-in 2 fan-out 1 arithmetic circuits from Definition 3.3.7 can describe any MPC function, we will focus on ZKP systems that utilize non-uniform circuits. Furthermore, it is reasonable to think that constructing protocol authentication ZK-statements as non-uniform circuits will generally result in smaller sizes compared to using uniform circuits.

Regarding specific choices of ZKP systems, we will consider the Groth16 SNARK [Gro16] and the original Bulletproofs system [BBB^{+}18], as their constructions are well-established and have been utilized in numerous real-world projects. Since Groth16 requires a trusted setup (being secure in the CRS model), a trusted party must generate and distribute the public parameters for the NIZK. Alternatively, a trusted setup could be achieved through MPC, known as a *trusted setup ceremony*, but this is typically only secure when there are many participants. In contrast, Bulletproofs have a transparent setup, as they are secure in

the RO model, meaning public parameters need only be established once and for all for the underlying cryptographic hash function.

4.3.2 Constructing ZK-Statements

In general-purpose ZKP systems that utilize non-uniform circuits, a ZK-statement is a relation defined for a *constraint satisfaction problem*, which is related to the NP-complete circuit satisfiability relation $R_C = \{(\mathbf{x}; \mathbf{w}) \mid C(\mathbf{x}, \mathbf{w}) = 1\}$. Essentially, this means that any NP relation, such as the ones of Definitions 4.2.2 and 4.2.3, can be reduced to a particular instance of a constraint satisfaction problem.

There are various types of constraint satisfaction problems, such as *quadratic arithmetic programs* [GGPR13] and *quadratic constraint problems* [HKR19]. However, many of these, including the aforementioned, can be reduced to from the so-called *rank-1 constraint system* (R1CS) problems [BCG$^+$13], making R1CS an attractive choice.

Definition 4.3.1 (Rank-1 quadratic system, adapted from [BCG$^+$13])**.** A system of rank-1 quadratic equations over $\mathbb{Z}_p$ is a tuple $S = ((\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j)_{j\in[N_g]}, n)$, where $\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j \in \mathbb{Z}_p^{1+N_w}$ and $n \leq N_w$ with N_g being the number of constraints, N_w the number of variables and n the number of input wires. S is satisfiable with public input $\mathbf{x} \in \mathbb{Z}_p^n$ if there is a witness $\mathbf{w} \in \mathbb{Z}_p^{N_w+1}$ with $w_0 = 1$ such that:

1. $\mathbf{x} = (w_1, \ldots, w_n)$,
2. $(\sum_i w_i a_{ij}) \cdot (\sum_i w_i b_{ij}) - \sum_i w_i c_{ij} = 0$ for all $j \in [N_g]$.

We write $S(\mathbf{x}, \mathbf{w}) = 0$ to denote this.

R1CS ZK-statements are expressed as $R_S = \{(\mathbf{x}; \mathbf{w}) \in \mathbb{Z}_p^n \times \mathbb{Z}_p^{N_w} \ : \ S(\mathbf{x}; \mathbf{w}) = 0\}$. Naturally, any fan-in 2 fan-out 1 arithmetic circuit can be converted into an instance of R1CS.

The strength of R1CS lies in its ability to emulate many programming language constructs through specific arithmetic circuit *gadgets*, which can be represented in R1CS. The process of converting programs into an arithmetic circuit is called *arithmetization*. While this is straightforward for simple programming constructs[5], it becomes increasingly complex for more advanced constructs. In Section 5.2, we will design a compiler that arithmetizes programs that define an MPC protocol.

4.3.3 ZKP Friendly Pedersen Commitments

The final consideration is how to reduce the Pedersen commitment to an R1CS instance with minimal constraints. In Construction 3.2.2, a commitment requires two cyclic group generators g and h, and is formed by the product of exponentiations $g^x \cdot h^r$. If we use the multiplicative group $\mathbb{Z}_p^*$, computing this over an arithmetic field would involve multiplying two exponential functions, which cannot be directly[6] described as a polynomial. To address this, the commitment can instead be computed over an additive group, such as an elliptic curve $E(\mathbb{Z}_p, +)$. Here, the generators are random points G and H, and the commitment is given by $[x]G + [y]H$, which itself is a polynomial, and relies on the hardness of the *elliptic curve discrete logarithm* problem. We note that the multiplication $[x]G$ in an additive group is the equivalent of the exponentiation G^x in a multiplicative group.

[5] See [But16] for a tutorial on R1CS arithmetization of simple programs.

[6] A workaround would involve creating a gadget for exponentiation through repeated multiplication of fixed-length integers, which would result in a very costly R1CS instance.

While conceptually straightforward, implementing this in practice requires elliptic curve arithmetic to be integrated into an arithmetic circuit. This requires the curve to be a subgroup of $E(\mathbb{Z}_p)$ where $\mathbb{Z}_p$ is the prime field of the arithmetic circuit. Fortunately, there are subgroups for curves over prime fields commonly used in ZKP systems, which we will discuss later in Section 5.3.3.

A powerful construction of a ZKP friendly Pedersen commitment scheme, first popularized by ZCash, is through the *windowed Pedersen commitment*. The underlying construction utilizes the Pedersen hash function [HBHW22], which is described in Figure 4.8. The idea

Note: $c = \left\lfloor \frac{1}{4} \log_2 \left(\frac{15 \cdot (r-1)}{8} + 1 \right) \right\rfloor$ where r is the order of the subgroup of $E(\mathbb{Z}_p)$.

Encode(S_i: bytearray)

/ split segment into chunk of 3 bits
$m[1..c] \leftarrow \mathsf{split}(S_i, c)$
for $j = 1$ **to** c **do**
  $s_0^j, s_1^j, s_2^j \leftarrow \mathsf{split}(m[j], 3)$
  $\mathsf{enc}(m[j]) \leftarrow (1 - 2 \cdot s_2^j) \cdot (1 + s_0^j + 2 \cdot s_1^j)$
return $\sum_{j=1}^{c} \mathsf{enc}(m[j]) \cdot 2^{4 \cdot (j-1)}$

PedersenHash(D: bytearray, M: bytearray)

$M' \leftarrow \mathsf{pad}(M, 0)$ **until** $M' \,\%\, 3 = 0$
$n \leftarrow \lceil \mathsf{length}(M')/(3 \cdot c) \rceil$
/ split message into segment of $3 \cdot c$ bits
$S[1..n] \leftarrow \mathsf{split}(M', n)$
return $\sum_{i=1}^{n} \mathsf{Encode}(S[i]) \cdot \mathsf{FindGroupHash}(D, i-1)$

Figure 4.8: The Pedersen hash algorithm [HBHW22, Section 5.4.1.7]

behind the Pedersen hash is to provide an efficient way to map an integer, represented as a byte array, to some unique point on a curve. This is done by segmenting the byte array M into segments $M_1, \ldots, M_n$, and computing the linear combination

$$\mathsf{PedersenHash}((G_1, \ldots, G_n), M) = \sum_{i=1}^{n} \mathsf{Encode}(M_i) \cdot G_i,$$

where $G_1, \ldots, G_n \in E(\mathbb{Z}_p)$ are pre-computed points, and $\mathsf{Encode}(S_i) \cdot G_i$ is the multiplication of a point by a constant obtained through the Encode function. The pre-computed points are uncorrelated generator points derived using the FindGroupHash function as described in [HBHW22, Section 5.4.9.5], which are obtained from a pseudorandom generator and a single generator point $G \in E(\mathbb{Z}_p)$. This approach of mapping an integer to a point on the curve is also referred to as *windowed* point multiplication, which gives rise to the *windowed* Pedersen commitment.

Construction 4.3.1 (Windowed Pedersen, modified [HBHW22, Section 5.4.8.2])**.** Given the algorithm PedersenHash from Figure 4.8, define the windowed Pedersen commitment:

$$\mathrm{commit}(x, r) = \mathsf{PedersenHash}(G, r) + \mathsf{PedersenHash}(H, x).$$

Unfortunately, the windowed Pedersen commitment scheme is not additively homomorphic. To obtain a homomorphic variant using pre-computed points, we can use:

$$\mathrm{commit}(x, r) = [x]\mathsf{FindGroupHash}(G, \alpha) + [r]\mathsf{PedersenHash}(H, \beta),$$

where α and β are fixed. Given that x and r are variables and the points are fixed, we can represent the variables in a fixed base and perform windowed multiplication as described in [HBHW22, Section A.3.3.7].

5 Implementation

In the previous chapter, we provided and discussed the conceptual GMW-style active-security compiler design, and precisely outlined and constructed the underlying building blocks. This exploration effectively addressed research question RQ1 and set the stage for RQ2, listed in Section 1.2.

In this chapter, our focus shifts to consolidating the construction developed in the previous chapter and going through the necessary steps to create a proof-of-concept for the active security compiler. In Section 5.1, we revisit the previously defined building blocks, presenting a modular design for the implementation of the active security compiler. Section 5.2 delves into the central component of this system, the ZKP compiler, which is the main driver of the active security compiler. We design and implement the ZKP compiler through a recent arithmetic circuit compiler infrastructure [OBW22]. Finally, in Section 5.3, we design a toolkit that integrates the compiler and the proving systems into a protocol runtime environment, and provide a basic library that contains and facilitates the development of relevant ZKP gadgets.

5.1 Software Design of the Active-Security Compiler

We already provided a partial answer to the question posed in RQ2 by enhancing the conceptual GMW compiler, incorporating the necessary details that allow it to work with modern ZKP systems. This adaptation also introduces distinct *engagement* and *emulation* phases: the *engagement* phase is where the initial communication round takes place through private point-to-point channels, except for ZKPs and other public information that are broadcasted; and the *emulation* phase is where all communication occurs over a public broadcast channel. This design decision allows for compilation of passively secure protocols that include a pre-processing phase or otherwise require initial point-to-point communication, without introducing additional functionalities for the compiler.

The next step to answering the question requires an implementation design of the active security compiler. In order to achieve this, we identify how the practical constructions fit into the design of the conceptual compiler, and unify in an architectural design. We then explore further sub-questions that are relevant to the design, and tackle these in the remainder of the chapter.

5.1.1 Bridging Theory and Practice

The conceptual active-security compiler, as described in Figure 4.7, relies on sub-protocols that implement the functionalities $\mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{COIN}}, and \mathcal{F}_{\text{ZK}}^{R}$ (where R is a ZK-statement). Practical constructions for these functionalities were explored in Section 4.3, with a particular focus on concepts such as verifiable computing and arithmetization schemes. It should be stressed that these concepts are necessary, as the required ZK-statements are dynamic

and dependent on the implementation of the passively secure protocol. We also noted that verifiable computing primarily depends on the choice of the ZKP proving system, which in turn requires an appropriate arithmetization scheme.

Based on the exploration in Section 4.3, we can summarize the overall requirements for a practical construction of the compiler:

- **Sequential protocol composition**: A passively-secure protocol has to be expressed as a sequential composition of deterministic one-round subprotocols. The use of sub-protocols is required for enforcing semi-honest behavior between the communication steps, and determinism is a necessary condition for obtaining a state-free arithmetic circuit.
- **ZK-statements**: The ZK-statements that are used in the conceptual compiler use the protocol authentication template (Definition 4.2.3). We concluded that all such statements can be embedded in simple protocol descriptions that allow for verifiable computing with respect to the input commitment.
- **ZKP proving systems**: Given that ZK-statements are to be represented in R1CS, we also require proving systems that support these statements and specifically are capable of creating NIZK proofs. In addition, we determined that for achieving malicious security with abort, such systems should either have a transparent setup or distributed trusted setup among all parties. In order to gather more insights about the active security compiler, we selected the following two candidates:
  - **With trusted setup**: The popular SNARK from [Gro16], commonly referred to as Groth16.
  - **Without trusted setup**: The ZKP system from [BBB+18], also known as Bulletproofs.
- **ZKP compilation**: Following from the two previous requirements, it is necessary for protocols to be expressible as an instance of R1CS. If we make the assumption that MPC functionalities can be described as arithmetic circuits, then it is possible to arithmetize all sub-protocols from the sequential composition. For this, we require a ZKP compiler that can automatically construct R1CS instances from a protocol specification. This compiler will depend on the protocol runtime environment.
- **Commitment scheme**: Since commitments are used in the ZK-statements given in Definition 4.2.3, we determined that the windowed Pedersen commitment from Construction 4.3.1 is the most suitable.
- **Coin tossing**: The most important requirement for coin tossing is that it is compatible with the commitment scheme used throughout the active-security compiler. For this, we would require the homomorphic variant of the windowed Pedersen commitment.

While these requirements provide guidelines on the implementation of individual parts, it remains imperative to explore how these parts interconnect and fit into the conceptual design.

5.1.2 Overview of the Compiler Structure

In our pursuit to design the compiler at the implementation level, we begin our descent into software design, and model the compiler as an interconnected system of components. An effective means of illustrating this is through the *unified modeling language* (UML) component diagram, as depicted in Figure 5.1. Here, we show that the compiler consists of five subsystems, which we describe as follows:

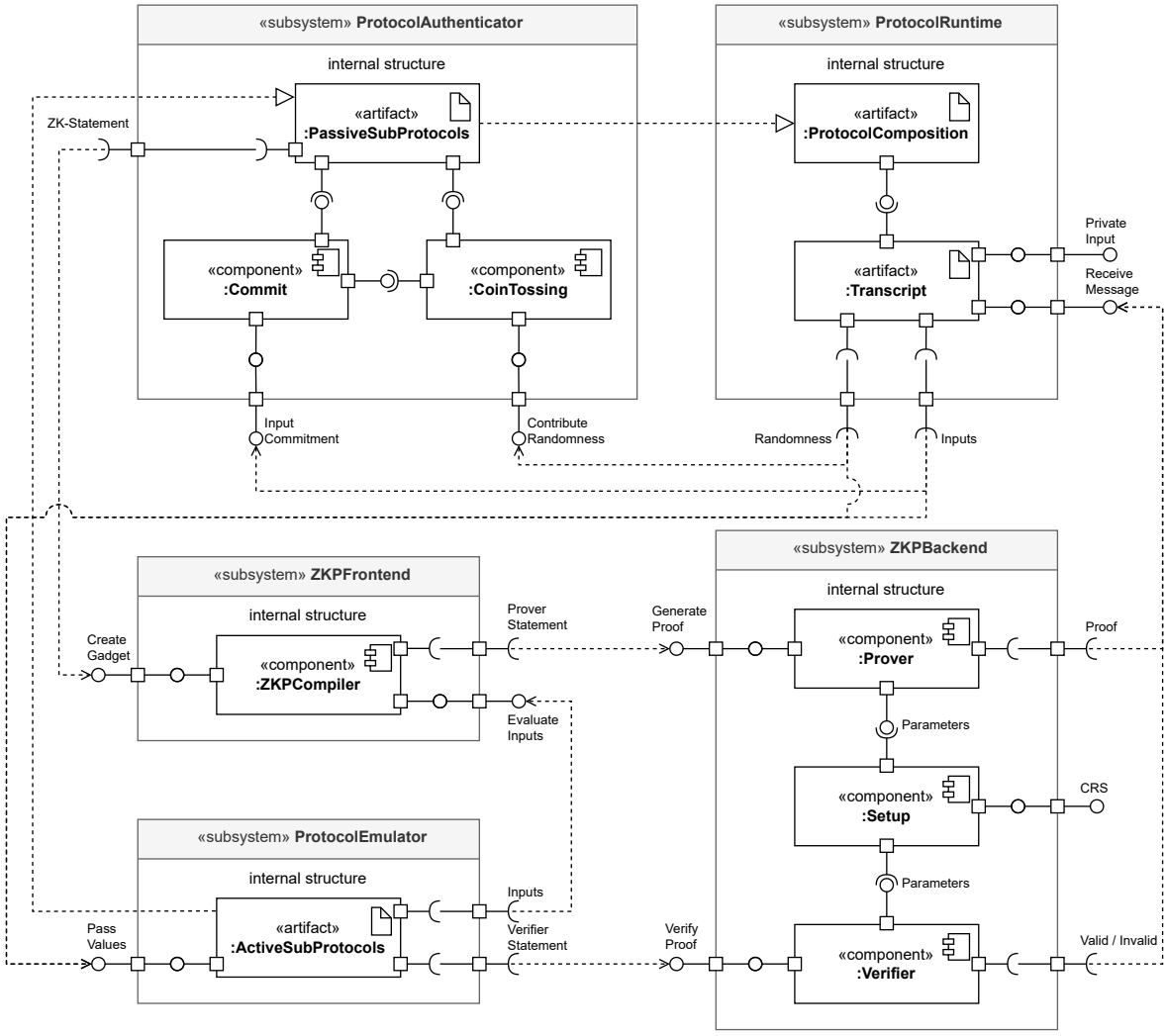


Figure 5.1: UML component diagram of the active security compiler.

- **ProtocolRuntime**: Any MPC protocol requires a system in which the protocol is executed. This is typically called a *protocol runtime*, and it serves as the driver that manages the protocol workflow, such as the handling of communication and the execution of instructions. This system should exist for passively-secure protocols, and as an entry point for the active security compiler. It is comprised of two entities that serve as artifacts:
  - **ProtocolComposition**: This should be a sequential composition of the passively-secure protocol, provided in the form of source code.
  - **Transcript**: Rather than the transcript being just a communication log, we consider it as a stateful manager that interprets and runs the protocol and deals with input/output operations.

  The resulting outputs go to the **ProtocolAuthenticator** and **ProtocolEmulator**.
- **ProtocolAuthenticator**: The protocol authenticator serves two purposes. The first purpose is to allow for execution of the commitment and coin tossing protocols, which are run in the *engagement* phase of the compiler. The second purpose is to facilitate the creation of the ZK-statements described in Section 5.1.1, which are represented as ZK-compilable source code and handed over to the **ZKPFrontend**.

- **ProtocolEmulator**: In the protocol emulator, the passively-secure subprotocols are executed and then authenticated. The latter happens by passing the inputs and output of the subprotocol to the **ZKPFrontend**, which is then responsible for creating the associated witness and instance variable assignments. These are also called the *prover* and *verifier statement* respectively.
- **ZKPFrontend**: The ZKP front-end is multifaceted, and has the responsibility of compiling ZK-statements into circuits, as well as evaluating the gadget for both the prover and verifier. Its sole component is the **ZKPCompiler**, which carries out all these tasks.
- **ZKPBackend**: The ZKP back-end is also multifaceted, and has the responsibility of generating proofs from a prover statement, verifying proofs from a proof and verifier statement, and when a trusted setup is necessary, generating the parameters from a CRS. The components are self-explanatory.

With our compiler structure in place, we now have a high-level understanding of the compiler implementation. This alone clearly does not answer RQ2, since we still do not know what facilitates the links between the subsystems and the components, and how such subsystems are realized in practice. We restate the research question with the following subquestions:

RQ2: *How can we build an active security compiler with minimal assumptions, capable of transforming a passively secure protocol, independent of its construction?*

a. *How can the required ZK-statements be automatically generated given a passively secure protocol?*

b. *How can ZKPs be used in a protocol runtime environment?*

c. *How can a proof of concept be built?*

We first address RQ2c in the remainder of this section, as it allows us to concretely specify additional requirements needed for RQ2a and RQ2b. The latter two topics are discussed in Sections 5.2 and 5.3, respectively.

5.1.3 Proof-of-Concept

Before going into the proof-of-concept implementation, it is crucial to make considerations for the **ProtocolRuntime** sub-system, as it has implications for all other components. To be specific, we have to choose a runtime environment, which could be any language that gets compiled to machine code or runs on a virtual machine. It may even be a more restricted language that only provides the programming constructs that are necessary for MPC. For implementing the compiler, and for providing support for custom passively-secure protocol implementations, we consider the popular programming language Python, version 3.10[1] (and above). One reason for this is that Python is becoming an increasingly popular language for the development of MPC protocols, such as those developed in the TNO PET lab [TNO21], and through the MPyC framework [Sch18]. Another (more important) reason is that Python provides excellent support for incorporating executable programs and external libraries, which will be necessary for integrating the ZKP front-end and back-end.

To keep the implementation simple, we prioritize the parts of the compiler that deal with automated proof generation and utilization of ZKPs. We believe that this has the highest priority for being studied due to its complexity and the fundamental role that it plays. By opting to

[1] The reason for using this specific version is that type-hint checking is coincidentally less strict than in older versions (for an unknown reason), which turns out to be useful when building the ZKP compiler.

use the non-homomorphic windowed Pedersen commitment from Construction 4.3.1we will omit the coin-tossing protocol and assume that all parties generate their randomness honestly. However, this may be compensated by the fact that not all passively-secure protocols require unbiased randomness, and that the compiler can still provide a lot of insight through experimentation and analysis.

Design

The design of the proof-of-concept implementation is illustrated in the UML class diagram in Figure 5.2. We briefly explain what this design shows, and how it is used:

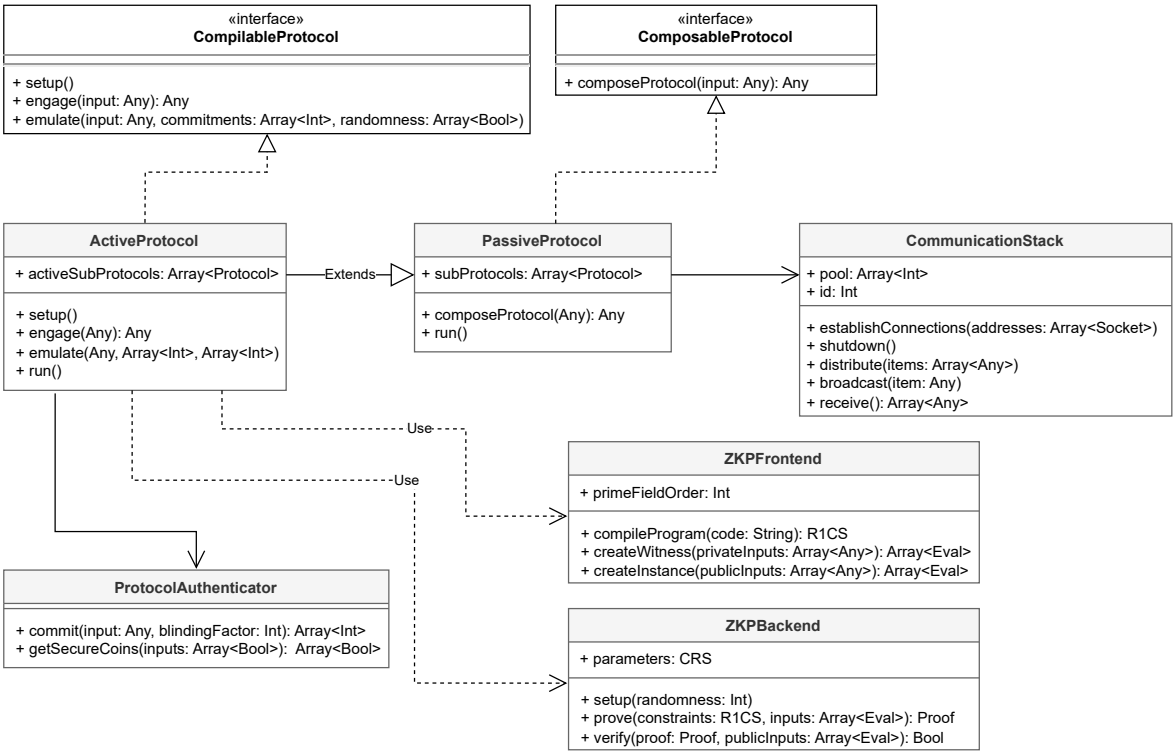


Figure 5.2: UML class diagram of the active security compiler.

- **PassiveProtocol**: This class forms the starting point and needs to be implemented by the end-user. It is required to implement **ComposableProtocol**, meaning that the *composeProtocol* method should iterate through the provided *subProtocols* and carry out the necessary communication between each round.
- **ActiveProtocol**: This class must implement **CompilableProtocol**, which means that the end-user is required to implement its methods together with the *activeSubProtocols* collection, in addition to providing the *subProtocols* collection. This is mostly a triviality and can be achieved by modifying some boilerplate code. Moreover, we require that the *activeSubProtocols* are defined using the methods of the **ProtocolAuthenticator**, and that their proofs and verifications are carried out through an interface provided by the **ZKPFrontend** and **ZKPBackend** classes.

It should be stressed that, at the current stage, the active security compiler is not fully automated; rather, it more closely resembles a framework. Nevertheless, it is plausible that a tool could be developed for automatically generating the implementation of **ActiveProtocol**, given an implementation of **PassiveProtocol**.

Implementation

The implementation can be found in [Rot24a], and follows the design from Figure 5.2. Some additional considerations are outlined as follows:

- The communication stack is built on top of the TNO communications library [TNO23].
- Since the communication stack implements asynchronous functions, we require the *setup*, *engage* and *emulate* methods to be asynchronous as well.
- Both the ZKP front-end and back-end are packaged in a toolkit, which we call *zkpy-toolkit*, described in more detail in Section 5.3.2.
- The *commit* method is provided by the toolkit's standard library, and the *getSecureCoins* method is based on a random number generator instead of coin-tossing.

The compiler's entry-point is the `run.py` script, where the name of the module that implements the MPC protocol should be specified, together with the following options:

- `--name`: The name of the protocol class (both active and passive).
- `--parties`: The total number of participating parties.
- `--idx`: The index or id of the party that is running the protocol.
- `--value`: The secret input to the protocol.
- `--security-level`: Either "passive" or "active".
- `--backend`: The backend to be used in case of running the active security compiler. Either "groth16" or "bulletproofs".

In order to utilize the compiler, the MPC protocol should be implemented as a Python package according to the structure specified in Figure 5.3.

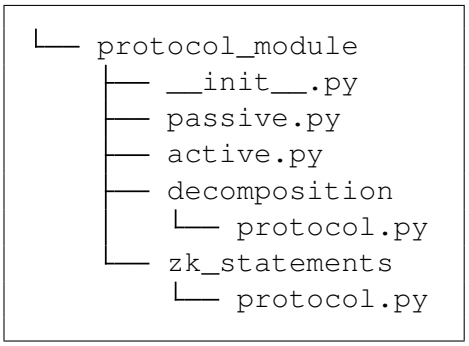


Figure 5.3: Package structure of an MPC protocol, compatible with the active security compiler.

The passively-secure protocol should be implemented in two steps:

1. The decomposition should be provided in `decomposition/protocol.py` as a collection of function definitions that are ZK-compilable, i.e., functions that can be compiled into an arithmetic circuit.
2. The protocol should be implemented in `passive.py`.

It is important to note that the decomposition should be ZKP-compilable, meaning that its implementation should be in accordance with the restricted Python subset specified in Section 5.2. We show how both steps are carried out in the following example:

Example 5.1.1. Suppose that a passively-secure 3-party protocol decomposition, based on secret sharing, consists of the subprotocols `protocol_0`, and `protocol_1`, and that the initial subprotocol is in the point-to-point communication model. The decomposition would be implemented as follows:

```
from zkpytoolkit.types import Private, Public, Array, field #ZK_IGNORE

def protocol_0(secret: Private[field]) -> Array[field, 3]:
    out: Array[field, 3] = [field(0) for _ in range(3)]
    # implement the protocol here
    ...
    return out

def protocol_1(shares: Private[Array[field, 3]]) -> field:
    out: field = field(0)
    # implement the protocol here
    ...
    return out
```

The protocol composition would then be implemented as follows:

```
from zkpytoolkit.types import field
from active_security_mpc.template.protocol import PassiveProtocol
from .decomposition.protocol import protocol_0, protocol_1

class Sum(PassiveProtocol):
    def __init__(self, local_idx, local_port, parties, enable_stats):
        super().__init__(local_idx, local_port, parties, enable_stats,
            field_type=field)

    async def compose_protocol(self, secret):
        N = self.parties

        outputs_0 = protocol_0(secret)
        inputs_1 = await self.communicate(outputs_0, "distribute", \
          "protocol_0", "field")

        output_1 = protocol_1(inputs_1)
        final_values = await self.communicate(output_1, "broadcast", \
          "protocol_1", "field")

        assert(all(output == output_1 for output in final_values))
        print(output_1)
```

To turn this passively-secure protocol into an actively-secure protocol, another two steps should be carried out:

1. The ZK-statements for each corresponding subprotocol, according to the design in Figure 4.7, must be defined.

2. The *setup*, *engage* and *emulate* phases should be implemented in `active.py`.

Although the design of the active security compiler simplifies this process, it still requires a substantial amount of boilerplate code. Additionally, the implementation of the ZK-statements must be done in a restricted subset of Python. While there is potential for automation, we do not consider it necessary for the proof-of-concept. The following example demonstrates the first step, the implementation of the ZK-statements. The rest of the implementation can be found in Section A.1 of the appendix.

Example 5.1.2. Suppose that the passively-secure protocol from Example 5.1.1 is implemented, and that we intend to use Bulletproofs as the ZKP back-end. We implement the ZK-statements in `zk_statements/protocol.py` as follows:

```
from zkpytoolkit import ZKP  #ZK_IGNORE
from zkpytoolkit.types import Private, Public, Array, field  #ZK_IGNORE
from zkpytoolkit.stdlib.commitment.pedersen.ristretto255.commit import
    commit_field as commit  #ZK_IGNORE

from ..decomposition.protocol import protocol_0, protocol_1

N: int = 3 # Hardcoded number of parties

def auth_protocol_0(
    secret: Private[field],
    blindings: Private[Array[field, 6]],
) -> Array[Array[int, 8], 3]:
    """Private protocol authentication for protocol 0"""
    commitments: Array[Array[int, 8], 3] = [
        [0 for _ in range(8)] for _ in range(N)
    ]
    outputs: Array[field, 3] = protocol_0(secret)

    for i in range(N):
        commitments[i] = commit(outputs[i], blindings[2*i:2*i+2])

    return commitments

def auth_protocol_1(
    shares: Private[Array[field, 3]],
    blindings: Private[Array[field, 6]],
    commitments: Public[Array[Array[int, 8], 3]],
) -> field:
    """Public protocol authentication for protocol 1"""
    for i in range(N):
        assert(commitments[i] == commit(shares[i], blindings[2*i:2*i+2])),
            "Invalid commitment"

    return protocol_1(shares)
```

In conclusion, it is important to highlight that the footprint for the proof-of-concept, along with the required steps for packaging the MPC protocol, is relatively small. This efficiency is primarily due to the ZKP compiler, which is crucial for enabling proof generation for custom protocol implementations. In the following section, we will delve into the design and implementation of this essential component, shedding light on its inner workings.

5.2 Zero-Knowledge Proof Compiler Implementation

ZKP compilers have been around and utilized for a considerable period; they come in different shapes and sizes, and have been developed for different purposes. For example, one of the first ZKP compilers to ever have been designed and implemented was for the automatic generation of Σ-protocols [BBH+10], which we explored at a foundational level in Section 3.4. A unifying language for more generic non-interactive implementations was then proposed in [MEK+10].

While protocol-oriented ZKP compilers certainly close the gap between theoretical and practical protocol design, they were not designed with the intention of mediating proof-based verifiable computing, which ultimately is the backbone for the active security compiler. The earliest work in this domain is attributed to the Pinocchio system [PHGR13], which introduced a compiler that compiles a restricted and stateless subset of C into a set of constraints that is then used by their SNARK. This work was succeeded by [BFR+13] and [BCG+13], who both designed compilers that compile stateful programs into constraints for stateless circuits. Both works are a hallmark for the way in which many ZKP compilers have since been designed and implemented. Notable examples are ZoKrates [ET18], Leo [CWC+21] and Circom [BIM+23].

The common thread among the previously listed compilers is that they are all defined for a *domain-specific language* (DSL), which is usually intended for specific applications. The ZKP compiler that is necessary for answering research question RQ2a must also be designed for a DSL, but specifically for one that is compatible with Python. This is possible with the help of the recent circuit compiler infrastructure: CirC [OBW22], which is motivated and described in the upcoming subsection.

5.2.1 Compiler Infrastructure: CirC

The CirC infrastructure is in many regards an analogue of the famous LLVM compiler infrastructure for traditional programming languages. In a similar way, it provides a framework for compiling programming languages to a semantically rich *intermediate representation* (IR), which can then be compiled to target-specific code. The key difference is that the LLVM-IR is an abstraction of a CPU architecture, whereas the CirC-IR is an abstraction of a so-called *existentially quantified circuit* (EQC). These types of circuits consist of wires, and constraints imposed on wires and stateless gates (similar to boolean and arithmetic circuits), and additionally support two types of inputs:

1. **Explicit inputs**: these correspond to the initial wires of the circuit.
2. **Existentially-quantified inputs**: these are variables that satisfy the circuit constraints.

To clarify why this abstraction is useful for a ZKP compiler, consider that in first-order logic an EQC would be given as $\exists B.P(A, B)$, where P is a quantifier-free predicate, A is an explicit variable, and B is an existentially-quantified variable. These types of formulas precisely match the definition of a ZK-statement given in Definition 3.4.1, and also generalize to other NP problems. Other such problems that CirC supports as compiler targets are the *linear integer programming* problem and the more general *satisfiability modulo theories* (SMT) problem. This cross-compilation capability turns out to be a major strength, since EQCs can be optimized using an SMT solver, which in addition can also be used to solve various compile-time problems [OBW22, Section 5].

While this abstraction presents a powerful compiler IR, it is important to consider its computation model, as it imposes certain restrictions on a DSL. Specifically, these restrictions include the following:

1. **Non-deterministic**: This means that for a given explicit input, there may exist more than one existentially-quantified input.
2. **Non-uniform**: The size of the input determines the size of the circuit.

A final reason that consolidates the use of CirC is supported by evidence that confirms the security of CirC's target-specific ZK-statements. This is presented in [OWBB23] through a formal verification of key compiler constructs.

Overview for Constructing a ZKP Compiler

We provided a brief overview of the strengths and the abstraction employed by CirC to facilitate the development of a ZKP compiler. Through the use of an IR, the compiler design adopts a modular structure, which allows for a streamlined development process. This is described in Figure 5.4, where the compiler pipeline is shown to consist of a front-end, middle-end and back-end.

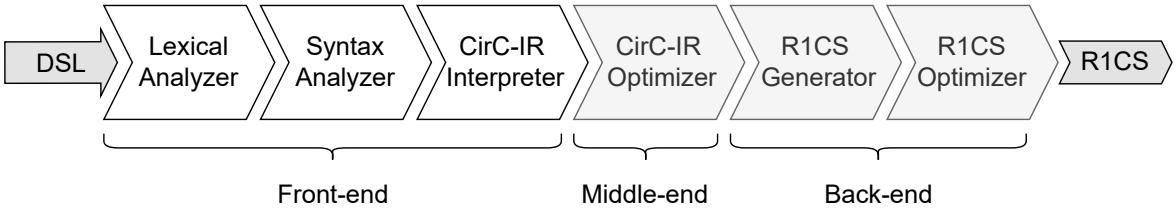


Figure 5.4: ZKP Compiler pipeline when using CirC.

Since a middle-end and back-end for targeting R1CS are already provided, our attention shifts to the implementation of the compiler front-end. The first two steps in the pipeline, i.e. the lexical and syntax analyzers, essentially correspond to a parser for the DSL that outputs an *abstract syntax tree* (AST). The third step, i.e. the CirC-IR interpreter, is where the CirC-IR is constructed by traversing the AST. This is arguably the most involved process of the entire pipeline, as appropriate transformations have to be considered for every possible node in the tree. Another challenge that arises at this stage is the required mitigation of a stateful DSL into a stateless IR. Thankfully, the infrastructure is equipped with a library called Circify, which provides the necessary machinery that abstract away stateful operations.

Technical Concepts

In order to carry out the implementation of the compiler front-end, we map out a high-level structure based on the case-study implementations[2] of ZoKrates and C, provided in the CirC code repository. Since the implementations are done in Rust, we provide an adapted UML class diagram in Figure 5.5 with the necessary level of detail that pertains to the language. The compiler front-end is based on three submodules, which are described as follows:

- **parser.rs**: In this module, a custom or predefined parser is utilized for the front-end. The Circify library requires an implementation for the **Loader** trait, which is responsible for parsing source code and identifying any dependencies that it may have. The implementation may also contain other associated functions; typically this would include a function for recursively compiling source code and their dependencies into ASTs, and providing support for standard library implementations.
- **term.rs**: In this module, the machinery that pertains to constructing the CirC-IR data structure from the DSL's AST is defined. We note that the IR is also represented as an

[2]The case-study front-end implementations pertain to commitment a586f7f of the repository in [Ozd23].

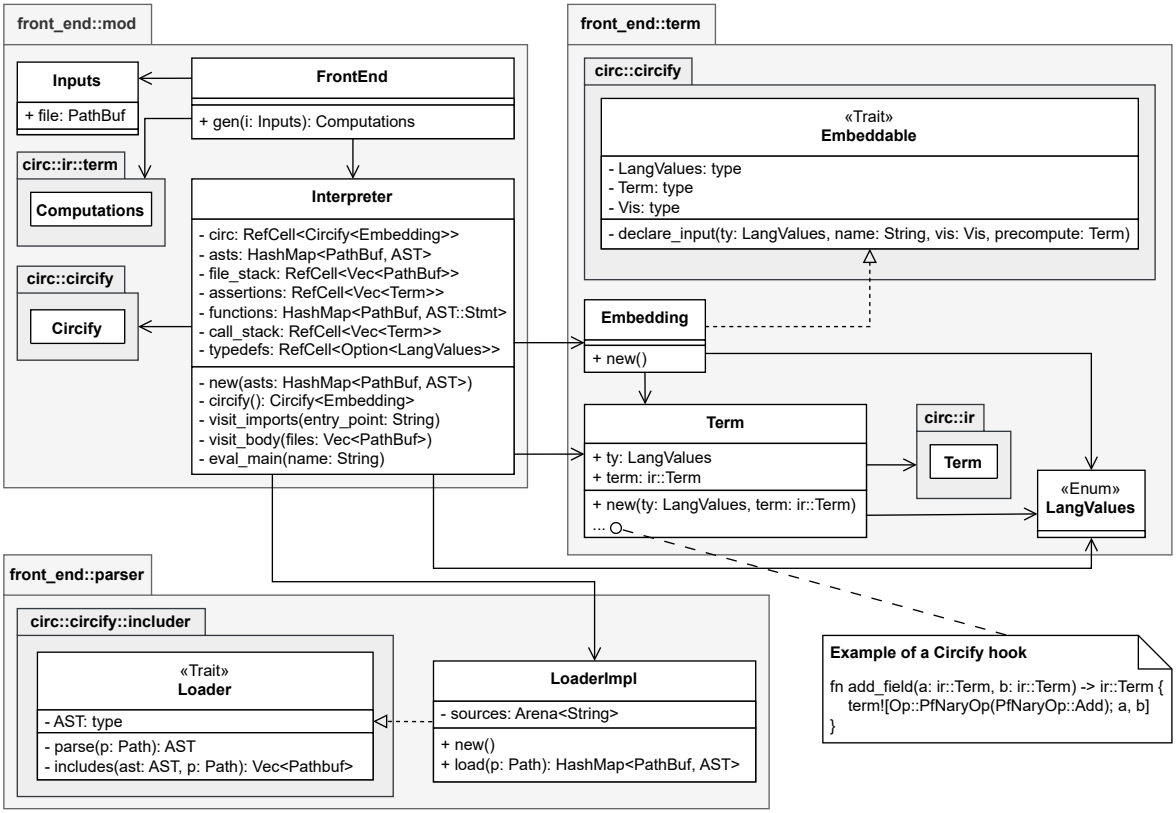


Figure 5.5: UML class diagram of a typical ZKP compiler front-end for CirC.

AST, which consists of internal nodes, referred to as *terms*, and meta-data that defines the term's datatype. The IR natively supports bit-vectors, booleans, prime fields, field-indexed arrays and tuples as its datatypes, and also comes with associated functions and macros for performing all necessary operations. Given this, one should should proceed in the following steps:

1. Define a mapping of DSL types into CirC-IR types, also referred to as *language values*, by creating a **LangValues** enum.
2. Construct a **Term** struct for the interpreter that pertains to the DSL. This is where one defines associated functions that abstract away low-level operations involved in constructing IR terms, also referred to as *hooks*. One can think of DSL terms as a composition of IR terms, so translations can be seen as production rules. This is also where Circify associated functions are invoked whenever stateful operations take place, such invocation of an array access operator, or conditional branching.
3. Finally, since the CirC-IR state is managed by Circify, it must be embedded into an instance of the state manager. Circify therefore requires an implementation of the **Embeddable** trait, which also provides an interface for processing prover and verifier inputs (based on their visibility) through the `declare_input` function. This is where transformations for typed DSL inputs to IR terms should be defined, also referred to as *precompute* terms.

- **mod.rs**: In this module, everything comes together to define the CirC-IR interpreter. An implementation would greatly vary between DSLs, but a typical implementation would hold a collection of ASTs and files, program assertions (due to the way Circify processes them), function and type definitions, and a function call stack. It is imperative to define functions for instantiating the interpreter and the Circify state manager,

for visiting the source code dependencies and their respective ASTs, and finally for building the IR starting from the main function. The **FrontEnd** struct provides an interface to the compiler front-end by implementing the `gen` function. There is a lot of freedom in how this function may be implemented, but typically it goes through the following stages:

1. **Setup**: First, an instance of the **LoaderImpl** struct is instantiated, and the source code and its dependencies are recursively loaded and compiled into an AST. Then it proceeds to traverse the ASTs, during which all function definitions, constants and other statements from the main body get stored in memory. It should be noted that an AST traversal should be implemented following the *visitor design pattern*, which in Rust is achieved via pattern matching [rus23].

2. **Semantic Analysis**: The next step is to interpret the AST and construct the associated CirC-IR. Formally, this involves a semantic analysis, where the type consistency and other rules are checked during the traversal. Since an AST typically consists of a body, a collection of statements and collection of expressions, all cases need to be handled using the appropriate hooks and Circify library functions. An overview of these functions is provided in Table 5.1. Since the interpreter begins by traversing the main function, or any other specified entry point, the Circify function `enter_fn` is initially invoked, followed by `exit_fn` once the tree has been fully traversed. Any invalid or unsupported operations would also have to be handled, and an appropriate error message should be returned.

3. **IR generation**: Once the interpreter successfully finishes, the state manager releases the final CirC-IR embedding and returns it as a **Computations** object.

The final step in building the compiler would be to join together the front, middle and backends, and to provide an interface for compiling into and evaluating R1CS instances. For compilation, this essentially boils down to the following four steps:

1. Initialize the compiler by globally setting a configuration using the `circ::cfg::set` function on an instance of `circ_opt::CircOpt`.

2. Provide a path to the source code inside of an instance of **Inputs**, and obtain an IR from calling the `FrontEnd::gen` function.

3. Optimize the IR by calling the `circ::ir::opt::opt` function on the computations and a vector of optimization strategies[3].

4. Lower the IR to R1CS by calling the `circ::target::r1cs::to_r1cs` function on the computations and the configuration. Optimize the R1CS instance by calling the `circ::target::r1cs::opt::reduce_linearities` function, and finally serialize and export the constraints and computations.

For evaluating the R1CS, the implementation depends on the proving system that is used. In all cases, a proving system requires witness and instance variable assignments, which can be obtained using functions from the `circ::target::r1cs::wit_comp` module. It should be noted that an input for the initial evaluation stage is declared in a Lisp dialect of the form: `(let ((var`$_1$ `literal) ... (var`$_n$ `literal)) t)`, where `let` is a binding for the value map with an associated value `t`. The associated value can be set to any boolean as it does not affect the value map that gets assigned to the precompute terms.

[3]For an example that applies to R1CS, see the code in `examples/circ.rs` in [Ozd23].

Table 5.1: Overview of Circify library functions used for building a CirC-IR.

Function	Description
`declare_input`	Declares an input to a computation
`declare_uninit`	Declares a new variable instantiation in the current lexical scope
`declare_init`	The same as `declare_uninit`, but with value initialization
`assign`	Assigns a value to a variable corresponding to a given location
`enter_breakable`	Emits a signal to enter a breakable block with an assigned name
`exit_breakable`	Emits a signal to exit the current breakable block
`break_`	Emits a break statement for a breakable block given its name
`enter_scope`	Emits a signal to enter a lexical scope
`exit_scope`	Emits a signal to exit a lexical scope
`enter_condition`	Emits a signal to enter a condition given a Boolean term
`exit_condition`	Emits a signal to exit the current condition
`condition`	Returns the current path condition
`enter_fn`	Emits a signal to enter a function given the name and return type
`exit_fn`	Emits a signal to exit the current function
`return_`	Emits a return statement
`assert`	Emits an assert statement
`get_value`	Returns the value given a variable location
`zero_allocate`	Emits a signal to instantiate an array of zeros and returns an identifier
`load`	Returns the term of the value stored in an array at a given index
`store`	Writes a term in an array at a given index
`replace`	Replaces a term in an array at a given index
`in_bounds`	Returns a term that indicates whether an array index is in bounds
`get_size`	Returns the size of an array given its identifier

5.2.2 Language Requirements and Design

With a clear understanding of the compiler infrastructure in place, it is time to proceed with the design specification of our own DSL. We recall that the goal is to design a language that is compatible with Python. Ideally, the language would be Python itself, but that is impossible due to the stateless nature of R1CS. We described in Section 5.2.1 that it is possible to mimic and reduce stateful language constructs to stateless circuits, wherever a stateful operation does not depend directly on an explicit input. Because of this and the fact that the language must be suitable for the Python runtime, we consider a proper subset of Python. We start with a discussion of the requirements that are imposed by CirC as well as the active security compiler, and follow this by a design of the language subset. We refer to this subset as the *zero-knowledge python compiler* (ZKPyC) language, for which we provide an implementation in [Rot24b].

Required Features

We briefly discuss the key requirements based on the previous sections, and consider what features are present in the superset language to facilitate that. We list the requirements as follows:

- **Type system**: The language must be typed and it must be expressible in the CirC-IR type system.
- **State and control flow**: By the non-uniformity condition, control flow may not depend on the size of an input. For iterative control flow, this means that loop bounds must

be fixed. For conditional control flow, this means that all cases must be handled for a condition. With regards to state, this means that conditions may not depend on mutable variables, unless the underlying expression can be solved at compile time.

- **Input specification**: By the non-determinism condition, all arguments of the main function must be marked as explicit or existential, and variables defined in the function body are considered to be existential. For ZKPs, this corresponds to the input being marked as public or private respectively.

To satisfy the first requirement, we rule out that the language should at least depend on Python 3.5 due to the introduction of type-hints in PEP 484 [vLL15]. The type-hint checking machinery has since developed a lot, and it has reached a point where its use can be customized for various type checking purposes. We will select Python 3.10 as the minimal version, since the runtime no longer performs a strict check for correct usage of type-hints in function definitions. We use this to our advantage to include literals as quantifiers in our syntax, without needing to wrap them in Python's `typing.Literal` object. Based on this, and the CirC-IR type system, we provide annotations for the following types:

- **Bool**: This is a primitive data type that corresponds to the bool type in CirC.
- **Int**: This can be represented as a bit-vector. For simplicity, it will be 32 bits large.
- **Field**: CirC provides native support for fields. In Python, these would first have to be implemented as a custom class.
- **Array**: While arrays are native to CirC, these can be understood as a list with an additional *size* property.
- **Class**: In CirC, this can be implemented as a tuple where every field stores data of a different type.

To satisfy the second requirement, we must semantically restrict the language such that any mutation on an explicit variable would be a violation. As an aside, it is common for languages to support variable-sized constants at compile-time through the use of generics. For example, in ZoKrates it is possible to define arrays and for-loops specified by a generic constant to avoid code re-use. While a similar syntax has been introduced in Python 3.12 through PEP 695 [Tv22], it is not possible to use generics as constants.

Lastly, the third requirement can be satisfied by providing type-hints that correspond to private and public types, referred to as *access specifiers*.

Language Specification

Based on the overall language requirements, we provide a formal specification of the ZKPyC language in two stages. First, we specify the type system and input accessibility through a *backus-naur form* (BNF) grammar in Figure 5.6. This is an extension to the Python 3.10 syn-

Note: the ⟨STRING⟩ and ⟨NUMBER⟩ non-terminal symbols are defined in the Python grammar.

⟨access_specifier⟩ ::= "Private" "[" ⟨type⟩ "]" | "Public" "[" ⟨type⟩ "]"
⟨type⟩ ::= ⟨atomic_type⟩ | ⟨class_type⟩ | ⟨array_type⟩
⟨atomic_type⟩ ::= "field" | "bool" | "int"
⟨array_type⟩ ::= "Array" "[" ⟨type⟩ "," ⟨NUMBER⟩ "]"
⟨class_type⟩ ::= ⟨STRING⟩

Figure 5.6: BNF grammar of supported types in the ZKPyC language.

tax grammar [Pyt21b]. To make it clear how this syntax is realized, we look at the following example:

Example 5.2.1. The following types would be valid:

- `Private[bool]`: This would be used to declare a private function argument to be a boolean.
- `Public[int]`: This would declare a public function argument to be an int.
- `Array[Array[int, 4], 2]`: This would declare a function argument, variable or constant to be a two-dimensional integer array of size 2×4.

As a remark, we note that the type signature for (nested) arrays is specified in a manner analogous to lists.

Next, we specify the language constructs that are possible in the ZKPyC language through a grammar in the *abstract syntax description language* (ASDL) [WAKS97]. This is the same language that is used to describe the AST of Python 3.10 in [Pyt21a], which simplifies the task of defining the ZKPyC language by simply removing and modifying constructs in accordance with language requirements. We briefly summarize how ASDL production rules are defined in the following example:

Example 5.2.2 (ASDL production rules)**.** We exemplify the ASDL production rules through the following grammar:

type	→	**Constructor**(*type*$_1$ field$_1$, …, *type*$_n$ field$_n$)	A constructor is a node with typed field names
	|	**AnotherConstructor$_1$**(*identifier* built_in_type)	Identifier is a builtin type
	|	**AnotherConstructor$_2$**(*string* built_in_type)	String is a builtin type
	|	**AnotherConstructor$_3$**(*int* built_in_type)	Integer is a builtin type
	|	**AnotherConstructor$_4$**(*constant* built_in_type)	Constant is a builtin type
type$_i$	→	**Node$_i$**(*type$_\ell$** zero_or_more_fields)	Product specifier * denotes zero or more fields
type$_j$	→	**Node$_j$**(*type$_\ell$*? optional_fields)	Option specifier ? denotes zero or one field
type$_k$	→	**Node$_1$** | ⋯ | **Node$_n$**	Constructors without fields are enumerations
type$_l$	→	(*type$_n$* attribute$_1$, …, *type$_n$* attribute$_n$)	No node means type is an attribute collection

We formally define the abstract syntax of the ZKPyC language in Figure 5.7, and provide a justification of the design choices in Section B.1 of the appendix.

All things considered, the subset language covers approximately 50% of all programming constructs in Python 3.10. This makes it a considerably extensive language with expressive capabilities that are well-suited for the active security compiler.

5.2.3 Implementation Using CirC

This subsection describes some of the technical details of the ZKPyC implementation. In Section 5.2.1 we saw what it takes to implement a compiler using the CirC compiler infrastructure. More specifically, we described the architecture for the front-end of a generic DSL in Figure 5.5. We will describe the process in a similar fashion and provide additional explanation where necessary. Moreover, to facillitate the implementation process, we will repurpose parts of the ZoKrates compiler front-end for CirC, due to the many similarities it shares with Python.

Note: the types *identifier, int, string* and *constant* are intrinsic to the ASDL

mod → **Module**(stmt* body)
stmt → **FunctionDef**(*identifier* name, *arguments* args, *stmt** body, *expr* returns)
| **ClassDef**(*identifier* name, *stmt** body)
| **Return**(*expr* value)
| **Assign**(*expr* targets, *expr* value)
| **AugAssign**(*expr* target, *operator* op, *expr* value)
| **AnnAssign**(*expr* target, *type* signature, *expr*? value, *int* simple)
| **For**(*expr* target, *expr* iter, *stmt** body)
| **Assert**(*expr* test, *expr*? msg)
| **Import**(*alias** names)
| **ImportFrom**(*identifier*? module, *alias** names, *int*? level)
expr → **BoolOp**(*boolop* op, *expr** values)
| **BinOp**(*expr* left, *operator* op, *expr* right)
| **UnaryOp**(*unaryop* op, *expr* operand)
| **IfExp**(*expr* test, *expr* body, *expr* orelse)
| **ListComp**(*expr* elt, *comprehension* generator)
| **Compare**(*expr* left, *cmpop** ops, *expr** comparators)
| **Call**(*expr* func, *expr** args, *keyword** keywords)
| **Constant**(*constant* value, *string*? kind)
| **Attribute**(*expr* value, *identifier* attr, *expr_context* ctx)
| **Subscript**(*expr* value, *expr* slice, *expr_context* ctx)
| **Starred**(*expr* value, *expr_context* ctx)
| **Name**(*identifier* id, *expr_context* ctx)
| **List**(*expr** elts, *expr_context* ctx)
| **Slice**(*expr*? lower, *expr*? upper, *expr*? step)
access → **Private**(*type* signature) | **Public**(*type* signature) | (*type* signature)
type → **Field** | **Bool** | **Int** | **Array**(*type* signature, *int* size) | (*identifier* class_name)
expr_context → **Load** | **Store**
boolop → **And** | **Or**
operator → **Add** | **Sub** | **Mult** | **Div** | **Mod** | **Pow**
| **LShift** | **RShift** | **BitOr** | **BitXor** | **BitAnd**
unaryop → **Invert** | **Not** | **UAdd** | **USub**
cmpop → **Eq** | **NotEq** | **Lt** | **LtE** | **Gt** | **GtE** | **Is** | **IsNot**
comprehension → (*expr* target)
arguments → (*arg** posonlyargs)
arg → (*identifier* arg, *access* visibility)
keyword → (*identifier*? arg, *expr* value)
alias → (*identifier* name, *identifier*? asname)

Figure 5.7: Abstract syntax of the ZKPyC language.

Implementing the Parser

In order for the language to be compatible with Python 3.10, it is imperative for the parser to rely on the same grammar. To keep things simple and error-free, we make use the Rust crate `rustpython-parser` (version `0.3.0`), which provides a `parse` function that can parse any Python module source code.

First, we define the structs **PyGadget**, consisting of a path to the directory that will house a relevant standard library, referred to as *stdlib*, and **PyLoad**, which consists of an stdlib

instance and provides a space for storing source code paths. The **PyGadget** struct then implements a `new` function for reading the path from an environment variable, or resorting to a pre-defined path. It also implements a function for converting submodules (from the source code directory or stdlib) to absolute paths. The **PyLoad** struct implements the Circify **Loader** trait using the already implemented parser, and also implements a `load` function for recursively parsing all dependencies.

The last feature that is implemented is a compile-time macro for specifying whether a line should be ignored by the compiler, and is handled by the function `filter_out_zk_ignore`. It filters out any lines matched by the regular expression: `(?i)#\s*zk_ignore\s*$`

Representing Language Values and Implementing Hooks

We recall from Section 5.5 that a front-end should include an enum that defines the DSL's language values, a struct for representing the DSL's terms, and hooks that transform DSL terms into CirC-IR terms. We begin be defining the language values as the following enum:

```
pub enum Ty {
    Field,
    Bool,
    Uint(usize),
    DataClass(String, FieldList<Ty>),
    Array(usize, Box<Ty>),
}
```

followed by the following ZKPyC language terms:

```
pub struct PyTerm {
    pub ty: Ty,
    pub term: Term,
}
```

For the **Ty** enum, we implement the associated function

```
fn sort(&self) -> circ::ir::term::Sort
```

to specify how each language value maps into a CirC-IR type. For the **PyTerm** struct, we implement public associated functions for unwrapping nested terms as well as initializing terms of every type. Next, we implement all the hooks that will be utilized by the interpreter. To illustrate this, we provide two examples of hooks for two different types of AST nodes.

Example 5.2.3 (Operators)**.** For the **Add**, **Sub** and **Mult** operators, we provide hooks by implementing the functions:

```
pub fn add(a: PyTerm, b: PyTerm) -> Result<PyTerm, String>
pub fn sub(a: PyTerm, b: PyTerm) -> Result<PyTerm, String>
pub fn mul(a: PyTerm, b: PyTerm) -> Result<PyTerm, String>
```

where the operation is then handled for each language value. For example, for the function `add` we handle cases for **Uint** and **Field** by implementing:

```
fn add_uint(a: Term, b: Term) -> Term {
    term![Op::BvNaryOp(BvNaryOp::Add); a, b]
}

fn add_field(a: Term, b: Term) -> Term {
    term![Op::PfNaryOp(PfNaryOp::Add); a, b]
}
```

Example 5.2.4 (Expressions)**.** For the **IfExp** and **Slice** expressions, we implement the following respective hooks:

```
pub fn cond(
    c: PyTerm,
    a: PyTerm,
    b: PyTerm,
) -> Result<PyTerm, String>

pub fn slice(
    arr: PyTerm,
    start: Option<usize>,
    end: Option<usize>,
    step: Option<isize>
) -> Result<PyTerm, String>
```

where `cond` and `slice` both depend on more primitive hooks. For example, `cond` uses the if-then-else hook, implemented as:

```
fn ite(c: Term, a: PyTerm, b: PyTerm) -> Result<PyTerm, String> {
    if a.ty != b.ty {
        Err(format!("Cannot perform ITE on {a} and {b}"))
    } else {
        Ok(
            PyTerm::new(a.ty.clone(),
            term![Op::Ite; c, a.term, b.term]),
        )
    }
}
```

and relies on the variable `c` being a term of type `Ty::Bool`.

The final step is the implementation of the language embedding that is required by the Circify state manager. For this, we define a public struct named **Python** and implement the **Embeddable** trait. This consists of associated functions for declaring the inputs to the main function, if-then-else branching, instantiating language values, and creating a term for the final return value. These functions depend on the earlier implemented hooks, and are instrumental for tracking mutations and combining terms accordingly.

Implementing the Interpreter

We now arrive at the final and most intricate component of the compiler front-end – the interpreter. In accordance with the general front-end design, we define the **Inputs** struct to denote the input to the compiler. This consists of a `file` field that indicates the path to the source code, and an `entry_point` field to denote the name of the main function. We implement the **FrontEnd** trait as described in Section 5.2.1. This involves implementing the `gen` function, which invokes the **PyLoad** recursive AST loader, runs the interpreter and finally extracts and returns the resulting IR circuit. The part of this sequence that deserves to be demystified, is how the interpreter is "run". This can be described in four steps:

1. During the initialization, a couple of data structures are created. Firstly, a reference to the Circify state manager is created with respect to the language embedding. We then create empty stacks for storing information that pertains to a function[4], such as its module path, return type, and also constant variables. A hashmap of the ASTs and stdlib are stored, and an empty hashmap is created for storing import statements, function and class definitions and constant expressions, which are populated in the next step.

[4]The call stack itself is implicit due to the visitor pattern, and is further handled by the Circify state manager.

2. To obtain all function definitions, class definitions and constant expressions that are required by the main function (specified by the entry point), the interpreter must scan and order its module dependencies. This is resolved through a *dependency resolution*, which is achieved by traversing the AST import statements, generating a directed acyclic graph of module dependencies, and applying a topological sorting in the order of last to first visit. Finally, the AST of each ordered module is visited and the corresponding hashmaps are populated.

3. With all function and class definitions in place, and constant terms having been interpreted and stored, the interpreter begins the IR construction process. Initially, a stack frame is constructed by calling the Circify `enter_fn` function given the name of the entry point and its return type. This is followed by an invocation of the `declare_input` function for each argument given its name, type and visibility. Then it proceeds to visit every statement in the function body by calling the `stmt_impl_` function on the function node of the AST, which in turn visits other statements and expressions according to the grammar in Figure 5.7. Finally, the `exit_fn` function is called, followed by an invocation of `declare_input` on the return value and `assert` on the conjunction of all assert statement conditions.

4. Lastly, the CirC-IR is obtained from the state manager and returned.

We note that step three constitutes the semantic analysis stage, which consists of type checking (and returning an error message upon a violation) and delegating AST statements and expression to Circify using hooks and other functions. Elaborating on each case is beyond the scope of this chapter and therefore deferred to the implementation source code [Rot24a].

Final remarks

All the previously described parts make up the implementation of the front-end for ZKPyC. As a proof of concept, the full compiler pipeline is implemented in a separate module called `zkpyc.rs` in accordance with Section 5.2.1, and provides a *command-line interface* (CLI) that outputs the R1CS meta-data for generating the constraints and witnesses. Another module called `zk.rs` implements a CLI for generating a witness and circuit description given the output of the compiler and a set of inputs to the main function. Examples thereof can be found in Section B.2 of the appendix, and more information on the usage of these CLIs are described in the source code repository.

This concludes the section on the ZKP compiler and provides a complete description of the component that aims to answer sub-question RQ2a. The remaining question RQ2b specifically addresses the utilization of this component within the active security compiler, and will be explored in the next section.

5.3 Zero-Knowledge Proof Toolkit with Standard Library

As previously discussed in Section 5.1.2, a practical ZKP system for the active security compiler consists of both a front-end and a back-end subsystem. The ZKP front-end is responsible for compiling any custom protocol to R1CS, while the NIZK back-end generates efficient proofs and verifies them non-interactively.

To this end, we developed a ZKP compiler, from which we obtain a front-end that is compatible with the protocol runtime environment described in Section 5.1.3. Additionally, we explored and selected suitable ZKP proving systems that can serve as a back-end, namely

Groth16 and Bulletproofs. In addressing sub-question RQ2b while aligning with the design decisions made thus far, we make the following considerations:

- We avoid re-implementing the complex schemes that are used by the back-end, and aim to facilitate support for drop-in replacements.
- Both subsystems must be interoperable.
- Integration of both subsystems into the Python runtime is necessary.

In light of these considerations, we develop a unifying library called the *zero-knowledge python toolkit* (ZKPyToolkit), and its implementation is provided in [Rot24c].

5.3.1 Front-end and Back-end Integration

We will now describe how this library manages to integrate both subsystems. To set the stage, we recall that the ZKP compiler was implemented in Rust. This means that the library has to provide a bridge for the Python runtime and the Rust implementation. One approach is to delegate the inputs and outputs to the CLIs described in Section 5.2.3. This approach would rely on inter-process communication, which introduces a new set of challenges, such as platform dependence, communication overhead, and potential issues with fault tolerance. We must also consider how the ZKP back-end is implemented, and what kind of bridge is required there. We already ruled out the option of providing a custom implementation due to their staggering complexities, aligning with the well-known principle: *"never roll your own cryptography"*. For that reason, we will make use of the community-accepted implementations for Groth16 and Bulletproofs, referred to as the Bellman implementation in [BG20] and the Dalek implementation in [dYA21], respectively. Both implementations make use of specific elliptic curves, namely:

- The Bellman back-end makes use of *bls12-381* elliptic curve.
- The Dalek back-end makes use of *ristretto255*, which is a prime order group built on the *curve25519* elliptic curve.

Since ZKPyC provides support for both curves, as well as *bn254*, we encode their scalar fields[5] directly into ZKPyToolkit, which are defined as:

$$q_{bls} = 52435875175126190479447740508185965837690552500527637822603658699938581184513,$$
$$q_{bn} = 21888242871839275222246405745257275088548364400416034343698204186575808495617,$$
$$q_{ris} = 72370055773322622139731865630429942408571163593799076060019509382854542509 89.$$

Implementation Overview

Since both the front-end and back-end components are implemented in Rust, it is reasonable to integrate them into ZKPyToolkit via Rust bindings. To achieve this, we build extension modules in Rust using the `PyO3` crate. The implementation details will not be explained here and are instead deferred to the code repository. Instead an overview of the structure is given in Figure 5.8. We note that there are two parts to the implementation:

1. **Rust bindings**: The first step is to implement Rust bindings, which serve as an interface between the Python module and the Rust implementations. These are provided in `src/rust`, and depend on the ZKPyC, Bellman and Dalek implementations.
2. **Python package**: The implementation for the toolkit is provided in `src/zkpytoolkit`, where `zkp.py` implements the overall system and `__init__.py` exposes the stdlib

[5] See the standard curve database [JS20] for a full specification of the elliptic curves.

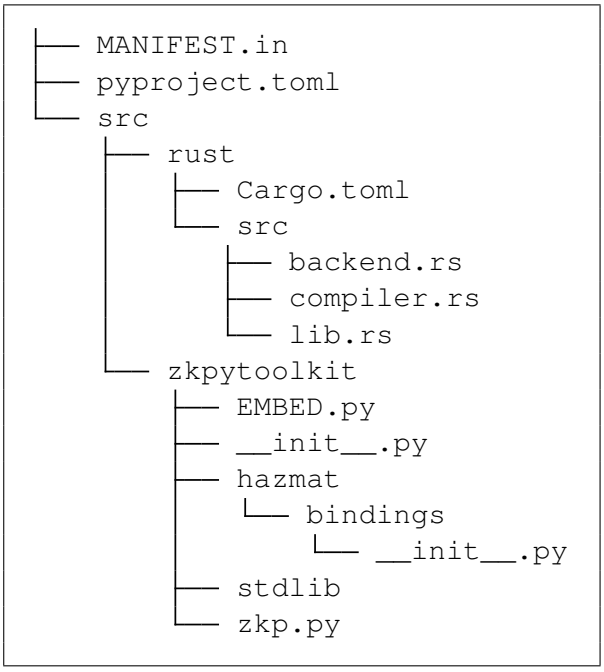


Figure 5.8: Implementation structure of ZKPyToolkit.

to ZKPyC. The package also consists of `EMBED.py`, which implements the ZKPyC built-in functions so that they can be integrated into the python runtime.

Interoperability of the Subsystems

To allow for the ZKP back-end to receive inputs from the front-end, and to support seamless drop-in replacements, a standardized format for the R1CS instance and associated prover and verifier inputs is essential. Fortunately, such a standard exists and is detailed in the zkInterface community proposal [BGK+20], officially accepted by the ZKProof standards committee.

The toolkit leverages zkInterface, where the compiler stores constraints and prover/verifier inputs in a dedicated serialized file. Subsequently, the back-end reads from these files and converts the information into a format specific to the ZKP proving system.

5.3.2 Usage of the Toolkit

The toolkit fully integrates into the Python runtime, and makes it extremely easy to construct and verify ZKPs for a given function. Specifics examples are described in the code repository and the toolkit is applicable to any of the examples described in Section B.2. For starters, a **ZKP** singleton from `zkpytoolkit` should be instantiated at the beginning of a Python module, provided with the keywords:

- `modulus`: Either “bls12_381", “bn256" or “curve25519".
- `backend` (optional): Either “groth16" or “bulletproofs”.
- `id` (optional): An integer for identifying the ZKP meta-data.
- `module` (optional): The name of the module in which the ZKPs are created.

We stress that the modulus and back-end *must* correspond.

The ZKPyC types and access specifiers are defined in the `types` module, and should only be imported after instantiating the toolkit.

Overview of Functions

It is worth describing the toolkit functionality in the follow manner:

Front-end: This consist of methods that pertain only to the ZKP front-end, i.e. `compile`, `prepare_proof` and `prepare_verification`. All outputs are cached under `cache_{ID}`.

Back-end: These consist of methods for the ZKP back-end, i.e. `generate_crs`, `run_prover` and `run_verifier`. The CRS is returned as a bytestring, whereas the other outputs are cached.

Hybrid: To make the toolkit more streamlined and intuitive, the methods `prove` and `verify` are provided, which return a bytestring and boolean respectively. Moreover, the methods `store_crs` and `store_proof` replace the currently cached items, and `cleanup` empties the cache.

5.3.3 Standard Library for Creating Gadgets

In Section 4.3.3we outlined an efficient construction of the Pedersen commitment, which is essential for the active security compiler. This implementation can be regarded as a ZKP gadget, as it represents the commitment scheme as an efficient circuit that can be reduced to R1CS. We settled for the windowed Pedersen commitment (Construction 4.3.1) due to its widespread adoption in other ZKP front-ends, such as ZoKrates and Arkworks. The requirements for this are:

- An implementation of an embedded elliptic curve.
- The windowed Pedersen hash algorithm.
- Utility functions for unpacking/packing finite field elements into/from a boolean array, and multiplexing of windowed scalars from a set of generators.

Since a lot of different components go into it, it is natural to provide a package for the commitment scheme and make it available for re-use via the standard library. Thankfully, an implementation for this exists in the ZoKrates standard library [ZoK23], which due to its Python-like syntax, can be ported over with relative ease. Since the implementation is only compatible with the bn254 curve, we discuss how this is extended for the other curves.

Elliptic Curves for Pedersen Commitment

We recall that the toolkit supports the bls12-381, bn254 and curve25519/ristretto255 elliptic curves, hence we implement associated subgroups for each based on common Edwards cruve parameterizations (Definition 3.1.4). For the bls12-381 and bn254 curves, we consider the subgroups *Jubjub* [HBHW22, Section 5.4.9.3] and *Baby Jubjub* [WBB20] as they are widely used in gadgets for pairing-based ZKP systems and satisfy the security criteria standardized by the SafeCurves project [BL17]. For the ristretto255 group, we consider the Doppio subgroup [dY19] due to its integration into the Dalek Bulletproofs implementation, but we note that it does not fulfil the safe curve criteria so its security remains unclear. For convenience, the subgroup parameters are given in Table 5.2.

Since the windowed Pedersen commitment requires a set of generator points as opposed to a single generator, we pre-compute two sets of uncorrelated generators, described by the

Table 5.2: Edwards Curves Parameters for Embedded Elliptic Curves

Parameter	Jubjub	Baby Jubjub	Doppio
A	-1	168700	1
D	$-10240/10241$	168696	-63071
q	q_{bls}	q_{bn}	q_{ris}
co-factor	8	8	4

FindGroupHash function, from the points G and H using a tool[6] from the ZKPyToolkit code. The implementation of the Pedersen commitment is based on the Pedersen hash, described in Figure 4.8. A part of the gadget implementation looks as follows:

Example 5.3.1 (Implementation of Pedersen hash)**.** The ZKP compilable implementation of the Pedersen hash is as follows:

```
from zkpytoolkit.types import Array, field # zk_ignore
from zkpytoolkit.stdlib.utils.multiplexer.lookup3bitSigned import sel3s
from zkpytoolkit.stdlib.utils.multiplexer.lookup2bit import lookup as sel2
from zkpytoolkit.stdlib.ecc.edwardsAdd import add
from zkpytoolkit.stdlib.ecc.jubjubParams import JUBJUB_PARAMS

def pedersen_no_compress(
    inputs: Array[bool, 512],
    generator: Array[Array[Array[field, 2], 4], 171]
) -> Array[field, 2]:
    e: Array[bool, 513] = [
        *inputs,
        False
    ]

    a: Array[field, 2] = JUBJUB_PARAMS.INFINITY # Infinity
    cx: field = field(0)
    cy: field = field(0)

    for i in range(0, 171):
        cx = sel3s([e[3*i], e[3*i + 1], e[3*i + 2]], [generator[i][0][0],
            generator[i][1][0], generator[i][2][0], generator[i][3][0]])
        cy = sel2([e[3*i], e[3*i + 1]], [generator[i][0][1], generator[i
            ][1][1], generator[i][2][1], generator[i][3][1]])
        a = add(a, [cx, cy], JUBJUB_PARAMS)

    return a
```

[6]The tool can be found under `extra/pedersen_generators` in [Rot24c] and is based on the ZoKrates pycrypto library in [ZoK23].

6 Evaluation

This chapter focuses on assessing how practical the active-security compiler is, and aims to give an answer to research question RQ3. In Section 6.1 we examine the measurable aspects of performance within the context of this work and explore the methods to assess them. Considering that the compiler, in its conceptual form, is entirely built on cryptographic protocols, we make assertions about its general performance through an asymptotic analysis in Section 6.2. Finally, we evaluate the utility, usability and the practical performance via an empirical analysis of the proof-of-concept implementation in Section 6.3.

6.1 Measuring Performance

The question of how 'practical' the active security compiler is depends on its utility and usability, which may be measured qualitatively, as well as on its efficiency, which can be measured quantitatively. We evaluate the compiler's utility and usability via a case study in Section 6.3.2, and begin with an exploration of the compiler's efficiency in the current section. Such an exploration warrants a selection of appropriate efficiency measures for the conceptual compiler. While its construction may not include certain components found in the proof-of-concept implementation, and vice versa, a theoretical analysis will nevertheless shed light on its behavior across a number of variables. This behavior can be modeled by a measure of *complexity* pertaining to some resource that directly affects the compiler's performance. These complexity measures naturally give rise to hypotheses that can be verified empirically for the proof-of-concept implementation. With these goals in mind, we specify the sub-questions of RQ3 as follows:

RQ3: *How practical is the active security compiler, and how can it be further improved?*

a. *What is the theoretical performance of a compiled protocol?*

b. *What empirical methods can be used to test the compiler with respect to its theoretical performance?*

c. *What insights do the empirical findings yield in terms of possible improvements?*

In order to answer question RQ3a, we must consider all measurable aspects of the compiler construction given in Figure 4.7. We note that in this construction, the compiler introduces additional overhead to the passively-secure protocol from the moment that all parties engage using their private input until the moment that all parties learn the final output. This means that the performance of the actively-secure protocol can be taken as the sum of the performance of the passively-secure protocol and the performance of the compiler. It follows that the compiled protocol can therefore only be as good as the original protocol. Since the compiler itself is a composition of cryptographic protocols, it is reasonable to define its overall performance as the sum of the performance of its parts.

To better understand how the theoretical performance of a compiled protocol can be measured, we turn to the work of [BGIN21], which describes their active security compiler for MPC protocols in the preprocessing model with respect to two efficiency measures. These are the asymptotic upper bounds of the communication cost and of the correlated randomness required during the protocol's online phase. The reason that their work makes no assertions about computation time as a measure of efficiency, is that all computationally heavy operations take place in the offline phase before the parties interact, resulting in computationally inexpensive operations during the online phase. While this is true for MPC protocols in the preprocessing model, it cannot be assumed for our compiler construction due to its extensive usage of ZKPs and their known computational overhead. Moreover, our compiler makes no specific assumptions for correlated randomness, since the passively-secure protocol is not necessary secret-sharing based. We therefore evaluate the performance of the active-security compiler based on the following two measures applied to the compiled protocol:

1. Asymptotic upper bound of communication cost, i.e., communication complexity.
2. Asymptotic upper bound of computational cost, i.e., computational (time) complexity.

Both complexity measures will be described in big O notation as a function of the number of parties, number of communication rounds in the passively-secure protocol, and the amount of required random elements. Since all inputs and random elements are assumed to be over a prime field, we quantify the latter variable by the number of prime field elements, and also consider it as the unit for the communication complexity measure. On the other hand, we consider the computational complexity measure to have an arbitrary unit of time. We explore this in depth in Section 6.2.

After carrying out an asymptotic analysis with respect to the two complexity measures, we aim to answer question RQ3b and carry out an empirical analysis on observable measurements. This question is discussed in detail in Section 6.3, but we can already foreshadow that its respective methodology is quite straightforward. However, the following caveats should be noted:

- The proof-of-concept implementation may incur an additional overhead from certain components.
- The observed measurements may be of a different unit as opposed to their respective complexity measures.
- An empirical analysis may require further assumptions and constraints due to the limitations of the proof-of-concept implementation.

These considerations are discussed in detail in the empirical methodology in Section 6.3.1.

6.2 Asymptotic Analysis

Let us recall that the active security compiler Π_{ACTIVE} (Figure 4.7) is a composition of the protocols Π_{COMMIT} (Figure 4.4), Π_{COIN} (Figure 4.5) and Π_{ZK}^{R} (Figures 4.6). We note that the compiler depends on the number of parties n, the number of random field elements m and communication rounds k required by the passively-secure protocol, and the ZK-statement R (which is generated from the protocol source code). One could additionally consider the security parameter as a variable that influences the compiler's performance, but we will assume it to be fixed. Furthermore, we assume that the compiler is run sequentially and that all parties repeatedly interact via the aforementioned protocols until each party received their required outputs. We therefore describe the overhead introduced by the compiler as

a sum of iterated protocol executions in Table 6.1 according to the two supported communication models. By sequential composability, this overhead calculation is applicable to both the communication and computational costs, and does not include the overhead of the passively-secure protocol.

Table 6.1: Active security compiler overhead as a sum of iterated compiler components, based on the communication model of the passively-secure protocol.

Broadcast-only model	Initial point-to-point model
$m \cdot \Pi_{\text{COIN}}$	$m \cdot \Pi_{\text{COIN}}$
$+ \Pi_{\text{COMMIT}}$	$+ (n+1) \cdot \Pi_{\text{COMMIT}}$
$+ \sum_{\ell=0}^{k-1} \Pi_{\text{ZK}}^{R^+_{auth}(\Pi^\ell)}$	$+ \Pi_{\text{ZK}}^{R^-_{auth}(\Pi^0)} + \sum_{\ell=1}^{k-1} \Pi_{\text{ZK}}^{R^+_{auth}(\Pi^\ell)}$

Before delving into the asymptotic analysis of the compiler, we recall that Π_{ZK}^{R} requires access to an arithmetic circuit that corresponds to the ZK-statement R and depends on the size of the input due to the non-uniformity property. We define the size of an arithmetic circuit as follows:

Definition 6.2.1 (Circuit Size)**.** Let C_f be an arithmetic circuit for a function $f : \mathbb{Z}_p^q \to \mathbb{Z}_p^r$. We denote the circuit size by $|C_f(q)|$ as the sum of all multiplication and addition gates, given the size of the input q.

Since the ZK-statements used in the active security compiler are constraint satisfiability problems, they indirectly correspond to some function that admits an arithmetic circuit. We may therefore measure the complexity of a statement R by $|C_R(q)|$. This can be used to conveniently reason about the asymptotic complexities of the ZKP systems that are used in the construction of Π_{ZK}^{R}, with respect to the input size of the arithmetic circuit. Since we make use of the Groth16 and Bulletproofs systems, we describe and outline the complexity measures of their relevant components in Table 6.2 for later use.

Complexity Measures of ZKP Systems

Generally a ZKP system aims to create proofs that are small in size, and performs computationally well in proof generation and verification time. We gather insights in the form of complexity measures with respect to the input size of the circuit for Groth16 and Bulletproofs. To remain consistent with our choice of variable names, we denote by N the size of the input to the circuit, and by M the number of multiplication gates. Naturally, $M = O(|C(N)|)$.

Groth16 [Gro16]: The proof size is always equal to 3 group elements, so it is of the order $O(1)$. The proof generation time is computationally bounded by the number of exponentiations: $N + 3M - \ell$, where ℓ is the number of field elements in the prover statement. By definition of R1CS (Definition 4.3.1), we have $\ell = N + N_w$ where N_w is the number of witness variables and $N_w \geq N$. It is easy to see that the lower bound is $\ell = \Omega(N)$, and it is reasonable to assume that the number of witness variables is at most equal to the number of gates in the circuit, i.e. $\ell = O(|C(N)|)$. Putting it all together, the proof generation time is of the order $O(|C(N)|)$. Finally, verification time is computationally bounded by the number of exponentiations ℓ, so it is also of the order $O(|C(N)|)$.

Bulletproofs [BBB$^+$18]: The proof size is equal to $2 \log M + 13$ group elements, so it is of the order $O(\log |C(N)|)$. The proof generation and verification times both depend on a number of group exponentiations and scalar multiplications with respect to M. While

the work proposes and implements optimizations in which exponentiations are reduced to only few multi-exponentiations that run in $O(M/\log M)$, the time complexity is ultimately dominated by the number of scalar operations, which is of the order $O(M)$. Therefore both the proof generation and verification time are of the order $O(|C(N)|)$.

Table 6.2: Asymptotic complexities of the ZKP systems used in the active security compiler. R is some ZK-statement, C_R is its corresponding arithmetic circuit, N is the size of the input to the circuit and $|C_R(N)|$ denotes the circuit size.

ZKP System	Proof Size	Computation Time	
		Prover	Verifier
Groth16	$O(1)$	$O(\|C_R(N)\|)$	$O(\|C_R(N)\|)$
Bulletproofs	$O(\log \|C_R(N)\|)$	$O(\|C_R(N)\|)$	$O(\|C_R(N)\|)$

6.2.1 Communication Complexity

To find the communication complexity of the protocol that is obtained using the active security compiler, we begin by describing it concretely in terms of its communication cost. The communication cost is the total number of field elements that are exchanged over a communication channel by any given party during the protocol execution. For ease of reference, we describe it as a function of n, m and k, and decompose it into the original communication cost and the additional overhead introduced by the compiler, i.e.

$$C(n, m, k) = C_0(n, m, k) + C_{\Pi_{\text{ACTIVE}}}(n, m, k).$$

We assume $C_0(n, m, k) = \sum_{\ell=0}^{k-1} C_{\Pi^\ell}(n, m)$ to be known and divert our attention to finding $C_{\Pi_{\text{ACTIVE}}}(n, m, k)$. This is done by substituting the communication costs $C_{\Pi_{\text{COMMIT}}}(n)$, $C_{\Pi_{\text{COIN}}}(n)$ and $C_{\Pi^R_{\text{ZK}}}(n, m)$ into the overheads provided in Table 6.1. We note that the three protocols are actually run n times by each party, where in one instance a party participates as P_i and at every other instance as P_j, i.e.

$$C_\Pi(n, m) = C_\Pi^{(P_i)}(n, m) + \sum_{j \in [n] \setminus \{i\}} C_\Pi^{(P_j)}(n, m).$$

We consider this to be the cumulative communication cost of each party running protocol Π, which accounts for the asymmetric interaction of the parties, as described in the protocol functionalities in Section 4.2.3. Finally, we assume that transmitting information over a broadcast channel is equivalent to transmitting over n individual point-to-point channels, which we from now on refer to as *message passing*. This way, we count the cost of message passing as the total number of field elements transmitted via each protocol, and provide a summary in Table 6.3. We note that in the case of $C_{\Pi^R_{\text{ZK}}}(n, m)$, the cost depends on the proof

Table 6.3: Communication costs of the protocols required by the active security compiler.

	$C_{\Pi_{\text{COMMIT}}}(n)$	$C_{\Pi_{\text{COIN}}}(n)$	$C_{\Pi^R_{\text{ZK}}}(n, m)$
Party P_i	$n - 1$	$4 \cdot (n - 1)$	$(n - 1) \cdot C_{\text{proof}_R}(n + m)$
Party $P_{j \neq i}$	1	$2 \cdot n$	$C_{\text{proof}_R}(n + m)$
Cumulative Cost	$2 \cdot (n - 1)$	$2 \cdot (2 + n) \cdot (n - 1)$	$2 \cdot (n - 1) \cdot C_{\text{proof}_R}(n + m)$

size for ZK-statement R, which is calculated based on the input size of the corresponding

circuit. More specifically, for R^+_{auth} the input size is $N = 3n + 3m + 1$, and for R^-_{auth} it is $N = 2n + 3m + 3$, which are both $O(n + m)$.

Asymptotic Analysis of Communication Cost

Given the generalized compiler overhead in Table 6.1, the communication costs of compiler components in Table 6.3 and the complexity measures of the ZKP systems in Table 6.2, we have all the tools to describe the communication complexity of the active security compiler.

In big O notation, it is easy to see that $C_{\Pi_{\text{COMMIT}}}(n) = O(n)$ and $C_{\Pi_{\text{COIN}}}(n) = O(n^2)$. However, the communication complexity $C_{\Pi_{\text{COMMIT}}}(n) = O(nC_{\text{proof}_R}(n + m))$ depends on the ZKP system, as well as the circuit size given the size of the input and the ZK-statement R. We already established that the circuit size input $N = O(n + m)$, but we still have a dependence on R. By making use of Lemma C.1.1 in the appendix, we have $|C_R(N)| = \text{poly}(N)$. Moreover, we consider $O(\log(\text{poly}(N))$ to be $O(\log(N))$ by the additive property of logarithms. Putting everything together in Table 6.4, we obtain two communication complexities for $C_{\Pi_{\text{ACTIVE}}}(n, m, k)$ using either Groth16 or Bulletproofs.

Table 6.4: Communication complexity of the active security compiler and its components. A distinction is made between a compiler that uses the Groth16 or Bulletproofs ZKP system.

Protocol	Communication Complexity	
	Groth16	Bulletproofs
$C_{\Pi_{\text{COMMIT}}}$	$O(n)$	$O(n)$
$C_{\Pi_{\text{COIN}}}$	$O(n^2)$	$O(n^2)$
$C_{\Pi^R_{\text{ZK}}}$	$O(n)$	$O(n \log(n + m))$
$C_{\Pi_{\text{ACTIVE}}}$	$O(n^2m + nk)$	$O(n^2m + nk \log(n + m))$

Based on these results, we can describe the communication complexity of the resulting actively-secure protocol through the following proposition:

Proposition 6.2.1 (Communication Complexity)**.** Let Π_0 be an n-party k-round passively secure protocol that requires m random field elements and has a communication cost of $C_0(n, m, k)$. Moreover, let Π be the resulting actively-secure protocol obtained using Π_{ACTIVE}. When Π_0 is in either the broadcast-only or initial point-to-point communication model, Π has communication complexity:

$$C(n, m, k) = C_0(n, m, k) + \begin{cases} O(n^2m + nk) & \text{if Groth16 is used} \\ O(n^2m + nk \log(n + m)) & \text{if Bulletproofs is used.} \end{cases}$$

If we apply this to protocols where k is some constant and $m = O(n)$, the compiler-induced communication complexity becomes $O(n^3)$ per party when using either Groth16 and Bulletproofs.

6.2.2 Computational Complexity

To find the computational complexity of the protocol that is obtained using the active security compiler, we proceed in a similar manner as Section 6.2.1. We first describe the computational cost as a function of n, m, k, which is decomposed into the original computational cost

and the additional overhead from the compiler, i.e.,

$$T(n,m,k) = T_0(n,m,k) + T_{\Pi_{\text{ACTIVE}}}(n,m,k).$$

Since the passively-secure protocol is in the synchronous model, we assume $T_0(n,m,k) = \sum_{\ell=0}^{k-1} T_{\Pi^\ell}(n,m)$ to be known, and focus on finding $T_{\Pi_{\text{ACTIVE}}}(n,m,k)$. Similar to how we obtained the communication cost, the goal is to first obtain the computational costs $T_{\Pi_{\text{COMMIT}}}(n)$, $T_{\Pi_{\text{COIN}}}(n)$ and $T_{\Pi^R_{\text{ZK}}}(n,m)$. In order to do this, we make two important considerations:

(a) All operations incur a cost: we assume that the costs related to message passing, i.e., T_{send} and T_{recv} are negligible, whereas the costs involving primefield arithmetics, i.e., T_{commit}, T_{mult}, T_{add} and ZKP operations, i.e., T_{prove_R} and T_{verify_R} are costly.

(b) Message passing dictates how the cumulative cost is calculated: since cryptographic protocols are a class of distributed algorithms, the time it takes for a protocol execution to complete depends on the slowest party, i.e.,

$$\sum_{\substack{i\in[n]\\ i\neq j_1\neq\cdots\neq j_{n-1}}} \max\left\{T_\Pi^{(P_i)}, T_\Pi^{(P_{j_1})}, \ldots, T_\Pi^{(P_{j_{n-1}})}\right\}.$$

Moreover, since the protocol functionalities interact with either P_i and $P_{j\neq i}$, it is not necessary that the parties have to run the protocol n times. For example, when only P_i sends something and all $P_{j\neq i}$ receive it, the cumulative cost is reduced to a single instance in which all parties concurrently run the steps of P_i, communicate it in a synchronous manner, and then run the steps of $P_{j\neq i}$.

Given these considerations, we count the computational costs as the sum of all significant operations and provide a summary in Table 6.5. We note that in the case of $T_{\Pi_{\text{COMMIT}}}(n)$ and $T_{\Pi^R_{\text{ZK}}}(n,m)$, party $P_{j\neq i}$ receives $n-1$ items but does not send any back, which results in the cumulative cost depending on only a single instance of the protocol execution. This is not the case with $T_{\Pi_{\text{COIN}}}(n)$, hence the cumulative cost has a multiplicative factor of n.

Table 6.5: Computational costs of the protocols required by the active security compiler.

	$T_{\Pi_{\text{COMMIT}}}(n)$	$T_{\Pi_{\text{COIN}}}(n)$	$T_{\Pi^R_{\text{ZK}}}(n,m)$
Party P_i	$T_{\text{commit}}(1) + T_{\text{send}}(n-1)$	$T_{\text{commit}}(1) + T_{\text{send}}(n-1) + T_{\text{mult}}(n) + 2\cdot T_{\text{add}}(n) + 2\cdot T_{\text{recv}}(n-1)$	$T_{\text{prove}_R}(n+m) + T_{\text{send}}(n-1)$
Party $P_{j\neq i}$	$T_{\text{recv}}(n-1)$	$T_{\text{commit}}(1) + T_{\text{send}}(n-1) + T_{\text{mult}}(n) + T_{\text{send}}(1) + T_{\text{recv}}(n-1)$	$(n-1)\cdot T_{\text{verify}_R}(n+m) + T_{\text{recv}}(n-1)$
Cumulative Cost	$\max_{j\in[n]}\{T^{(P_j)}_{\Pi_{\text{COMMIT}}}(n)\}$	$n\cdot\max_{j\in[n]}\{T^{(P_j)}_{\Pi_{\text{COIN}}}(n)\}$	$\max_{j\in[n]}\{T^{(P_j)}_{\Pi^R_{\text{ZK}}}(n,m)\}$

Asymptotic Analysis of Computational Cost

Similar as in Section 6.2.1, we use the generalized compiler overhead in Table 6.1, the computational costs of the compiler components in Table 6.5 and the complexity measures of the ZKP systems in Table 6.2 to obtain the computational complexity.

First we consider the computational complexities of the different operations. Assuming that message passing is computationally negligible, we have $T_{\text{send}}(n) = O(1)$ and $T_{\text{recv}}(n) = O(1)$. We recall that T_{commit} depends on group exponentiations and multiplications, which could be performed in sublinear time [Pip80]. Luckily, we only count instances of $T_{\text{commit}}(1) = O(1)$. Moreover, we consider that $T_{\text{mult}}(n) = O(n)$ and $T_{\text{add}}(n) = O(n)$. Finally, we have that T_{prove_R} and T_{verify_R} depend on the ZKP system. Since proof generation and verification are asymptotically linear in the circuit size for Groth16 and Bulletproofs, and by Lemma C.1.1, we have $T_{\text{prove}_R}(n+m) = O(\text{poly}(n+m))$ and $T_{\text{verify}_R}(n+m) = O(\text{poly}(n+m))$.

Next, we evaluate the computational complexities of the protocols: We get $T_{\Pi_{\text{COMMIT}}}(n) = O(1)$, $T_{\Pi_{\text{COIN}}}(n) = O(n^2)$ and $T_{\Pi^R_{\text{ZK}}}(n,m) = O(n\,\text{poly}(n+m))$. We put everything together in Table 6.6 and obtain a single computational complexity for $T_{\Pi_{\text{ACTIVE}}}(n,m,k)$ using either Groth16 or Bulletproofs.

Table 6.6: Computational complexity of the active security compiler and its components, based on the Groth16 and Bulletproofs ZKP systems.

Protocol	Computational Complexity
$T_{\Pi_{\text{COMMIT}}}$	$O(1)$
$T_{\Pi_{\text{COIN}}}$	$O(n^2)$
$T_{\Pi^R_{\text{ZK}}}$	$O(n\,\text{poly}(n+m))$
$T_{\Pi_{\text{ACTIVE}}}$	$O(n^2m + nk\,\text{poly}(n+m))$

The computational complexity of the resulting actively-secure protocol is thus given as the following proposition:

Proposition 6.2.2 (Computational Complexity)**.** Let Π_0 be an n-party k-round passively secure protocol that requires m random field elements and has a computational cost of $T_0(n,m,k)$. Moreover, let Π be the resulting actively-secure protocol obtained using Π_{ACTIVE}. When Π_0 is in either the broadcast-only or initial point-to-point communication model, Π has computational complexity:

$$T(n,m,k) = T_0(n,m,k) + O(n^2m + nk\,\text{poly}(n+m)).$$

If we apply this to protocols where k is some constant and $m = O(n)$, the compiler-induced computational complexity becomes $O(n\,\text{poly}(n))$ per party when using either Groth16 and Bulletproofs.

6.3 Empirical Analysis

In the previous section we devised two complexity measures for the active security compiler. Specifically, we described the communication and computational complexities of the resulting actively-secure protocol, which are comprised of the passively-secure protocol performance and the additional compiler overhead. Both measures describe the asymptotic behavior with respect to the number of parties, amount of randomness and number of communication rounds involved in the protocol. The goal of this section is to empirically verify the two measures in order to gain a better understanding of the true performance of the compiler, and to understand what the compiler's practical utility is.

6.3.1 Methodology

To answer research question RQ3b, we discuss a methodology for carrying out the empirical analysis.

Variables

First, we clearly state the variables that define the causal relation of the performance measures:

- **Independent variable:** The number of parties n.
- **Controlled variables:** The amount of randomness m, the number of communication rounds k involved in the passively-secure protocol, and the field size p.
- **Dependent variables:** The communication cost and the computational cost of the resulting actively-secure protocol.

The variables m and k are controlled variables since they depend on the MPC protocol. In fact, for secret-sharing based MPC the use of a (m, n)-threshold scheme means that $1 \leq m \leq n$, therefore $m = O(n)$. While it is possible to consider an experimental setup for a dummy protocol where m and k are independent variables, doing so would not demonstrate the compiler's utility for real-world protocols. Lastly, the field size p is controlled because it depends on the (supported) choice of elliptic curve used in the ZKP system.

Choice of Concrete Protocol and Compiler Considerations

The next step would be to consider a concrete passively-secure MPC protocol that fits into the framework of the active security compiler. We consider a specific construction of the multi-party sum protocol, decomposed as $\Pi_{\text{SUM}} = \Pi^2_{\text{SUM}} \diamond \Pi^1_{\text{SUM}} \diamond \Pi^0_{\text{SUM}}$, and describe its instantiation as a case study for the compiler, explained in detail in Section 6.3.2. We note that each sub-protocol has a circuit complexity of $O(n)$, allowing us to fine-tune the complexity measures for the compiler.

Furthermore, we highlight some limitations of the proof-of-concept implementation that need to be accounted for in the analysis, initially described in Section 5.1.3. The limitations are as follows: (1) the process of obtaining unbiased randomness through secure coin tossing is skipped, (2) a passively-secure protocol must be decomposed into a sequential decomposition, and the active security transformation must be implemented. Consequently, this means that we eliminate the coin tossing complexity terms and obtain a quadratic communication and computational complexity. We also note that the second part is easily achieved when following Examples 5.1.1, 5.1.2 and the active security transformation implementation details from Section A.1.

Hypotheses

Given the considerations so far, we state hypotheses for the practical performance of the compiler given the theoretical performance as described in Propositions 6.2.1 and 6.2.2. Moreover, we also hypothesize that the compiler performs better using one of two ZKP systems. We base this on the fact that Bulletproofs are known to be slow in both proof generation and verification time for large circuits, whereas Groth16 is known to have fast proof generation and almost constant-time verification. For example, in the benchmarks provided by [XZZ+19], a 64×64 matrix multiplication ZK-statement of 2^{18} constraints results in a 10^3 seconds proof generation and 10^2 seconds verification time for Bulletproofs, vs a 30 seconds proof generation and 1.8 milliseconds verification time for Groth16. This leads us to the following hypotheses:

Hypotheses:

1. *The communication cost of the compiled multi-party sum protocol without coin tossing grows quadratically in the number of parties when using either the Groth16 or Bulletproofs ZKP system.*
2. *The computational cost of the compiled multi-party sum protocol without coin tossing grows quadratically in the number of parties when using either the Groth16 or Bulletproofs ZKP system.*
3. *Both the communication and computational costs of the compiled multi-party sum protocol are lower when using the Groth16 ZKP system as opposed to using Bulletproofs.*

Experimental Design

We test the hypotheses listed above by estimating a generalization over repeated measurements and organizing these in a way that allows for easy comparison. Statistically, a generalization can be estimated by calculating either a sample mean or sample median of the repeated measurements. We assume the presence of extreme outliers in the measurements due to compiler's reliance on many software and hardware components, making the median a suitable estimator due to its robustness to outliers.

All measurements are obtained through a benchmark of the multi-party sum protocol before and after compilation, which is analyzed in Section 6.3.3. The primary performance measures are the protocol runtime (in seconds), the communication (in bytes) and cache (in bytes). While both runtime and communication are sufficient to test all three hypotheses, their dependence on cached data necessitates a deeper analysis that includes data specific measurements. Moreover, cached data also partially represents memory usage during points of the protocol runtime, which is an important empirical performance metric. Additionally, we are also interested in measuring the cost of protocol authentication vs simple protocol evaluation in terms of the number of constraints.

For each experiment, we model a party as an individual CPU core, starting with $n = 3$ parties and increment until a reasonable pattern can be seen from the measurements. Every experiment is then repeated at least 10 times, and an initial experiment is run and discarded to warm up the CPU. The experiments are carried out as follows:

1. Each party runs an instance of the passively-secure protocol on a uniformly random input, where the total runtime and communication are measured between the start and the end of the running program.
2. Each party runs an instance of the actively-secure protocol on a uniformly random input, obtained using the compiler with the Groth16 ZKP system, where:
   (a) the total runtime, communication and cache size are measured between the start and end of the running program.
   (b) the runtime and communication are measured between the start and end of the setup, engagement and emulation phases of the compiler.
   (c) the cumulative runtimes for the ZKP compilation, CRS generation, proof generation, verification and communication components are measured.
   (d) the cumulative R1CS constraints, proof sizes and CRS sizes are measured.
3. The previous experiment is carried out using the Bulletproofs ZKP system.

4. Using the ZKP compiler with respect to the primefield required by the Groth16 and Bulletproofs ZKP system, the R1CS constraint sizes of the following ZK-statements are measured:
   (a) The protocol authentication statements $R^{-}_{auth}(\Pi^{0}_{\mathrm{SUM}})$, $R^{+}_{auth}(\Pi^{1}_{\mathrm{SUM}})$ and $R^{+}_{auth}(\Pi^{2}_{\mathrm{SUM}})$.
   (b) The protocol evaluation statements (Definition 4.2.2) $R_{eval}(\Pi^{0}_{\mathrm{SUM}})$, $R_{eval}(\Pi^{1}_{\mathrm{SUM}})$ and $R_{eval}(\Pi^{2}_{\mathrm{SUM}})$.

6.3.2 Case Study: Compiling a Multi-Party Sum Protocol

For the empirical analysis, we study a multi-party sum protocol due to its simplicity and ubiquitous presence in MPC. The first step to compiling this passively-secure protocol is to describe the protocol as a sequential decomposition, which is outlined in Figure 6.1. The decomposition requires all primitives to be defined, which in this case are based on

Protocol Decomposition $\Pi_{\mathrm{SUM}} = \Pi^{2}_{\mathrm{SUM}} \diamond \Pi^{1}_{\mathrm{SUM}} \diamond \Pi^{0}_{\mathrm{SUM}}$

Input: Parties $P_1, \ldots, P_n$ have private inputs $x_1, \ldots, x_n \in \mathbb{Z}_p$ and private randomness $(u_{1j})_{j=1}^{n-1}, \ldots, (s_{nj})_{j=1}^{n-1} \in \mathbb{Z}_p^{n-1}$, respectively.
Output: All parties obtain $\sum_{i=1}^{n} x_i$.
Parameters: The primefield size p and $\mathrm{pp} \leftarrow p$.
Functional primitives: Let $\mathrm{TSS} = (\mathrm{Gen}, \mathrm{Share}, \mathrm{Reconstruct})$ be an additive TSS scheme.

$f_{\mathrm{share}}(x_i, u_{i1}, \ldots, u_{in-1})$

$(m_j)_{j\in[n]} \leftarrow \mathrm{TSS.Share}_{\mathrm{pp}}(m, n; u_{i1}, \ldots, u_{in-1})$
$(s_{i1}, \ldots, s_{ii-1}) \leftarrow (m_1, \ldots, m_{i-1})$
$s_{ii} \leftarrow m_n$
$(s_{ii+1}, \ldots, s_{in}) \leftarrow (m_i, \ldots, m_{n-1})$
return $(s_{i1}, \ldots, s_{in})$

$f_{\mathrm{recon}}(x_1, \ldots, x_n)$

return $\mathrm{TSS.Reconstruct}_{\mathrm{pp}}(\{x_1, \ldots, x_n\})$

Sub-protocols:
Π^{0}_{SUM}: For each $i \in [n]$, P_i computes $(s_{i1}, \ldots, s_{in}) \leftarrow f_{\mathrm{share}}(x_1, u_{i1}, \ldots, u_{in-1})$ and sends s_{ij} to party P_j for $j \in [n] \setminus \{i\}$.
Π^{1}_{SUM}: For each $i \in [n]$, P_i computes $r_i \leftarrow f_{\mathrm{recon}}((s_{1i}, \ldots, s_{ni})$ and broadcasts r_i.
Π^{2}_{SUM}: For each $i \in [n]$, P_i computes $\sigma \leftarrow f_{\mathrm{recon}}(r_1, \ldots, r_n)$ and broadcasts σ.

Figure 6.1: Sequential decomposition of the multi-party sum protocol.

a deterministic version of the additive TSS from Construction 3.3.1 that takes randomness as additional input. This is followed by the construction of each sub-protocol, which simply makes use of the primitives f_{share} and f_{recon}. We recall that in the sequential decomposition, each sub-protocol must end with either a broadcast or point-to-point communication. The final broadcast is not necessary since all parties learn the output at that point, but it will be included to be consistent with the compiler construction and allows parties to carry out a final sanity check.

The second step involves implementing the protocol decomposition, as described in Section 5.1.3. This initially requires an implementation of the secret-sharing primitives, which is given in Figure 6.2. While the functions can be easily programmed in the restricted Python subset ZKPyC, we note that it is indeed restrictive in a number of ways. First, type annotations must be defined statically, so making the scheme compatible with a variable number of parties requires a separate implementation for each fixed number of parties. Moreover, we run

```
from zkpytoolkit.types import Private, Public, Array, field          # ZK_IGNORE
from zkpytoolkit.EMBED import sum

N: int = 3                                              # Hardcoded number of parties

def generate_shares(
    x: Private[field], U: Private[Array[field, 2]],
    i: Public[int], _one: Public[field]
) -> Array[field, 3]:
    s_i: field = (x-sum(U))*_one
    S: Array[field, 3] = [field(0) for _ in range(N)]
    k: int = 0
    u_end: field = U[N-2]                                # Last element is U[N-2]
    for j in range(N-1):                                 # Avoid accessing U[N-1]
        S[j] = U[k] if j != i else s_i
        k += 1 if j != i else 0
    S[N-1] = s_i if k == N-1 else u_end
    return S

def reconstruct_secret(
    S: Private[Array[field, 3]], _one: Public[field]
) -> field:
    return sum(S)*_one
```

Figure 6.2: The additive secret-sharing scheme required by the multi-party sum protocol for three parties, implemented in `decomposition/additive.py`.

into the issue that the ZKP compiler evaluates all `if-else` branches, requiring a counter-intuitive workaround that avoids accessing a non-existing array element. Lastly, we also have to provide a dummy variable named `_one` due to a bug in the ZKP compiler[1].

Next we implement the protocol decomposition, whereby each protocol calls and returns the output to the correct secret-sharing primitive, in Figure 6.3. These protocols are ZK-

```
from .additive import generate_shares
from .additive import reconstruct_secret
from zkpytoolkit.types import Private, Public, Array, field          # ZK_IGNORE

def protocol_0(
    x: Private[field], U: Private[Array[field, 2]],
    i: Public[int], _one: Public[field]
) -> Array[field, 3]:
    return generate_shares(x, U, i, _one)

def protocol_1(S: Private[Array[field, 3]], _one: Public[field]) -> field:
    return reconstruct_secret(S, _one)

def protocol_2(R: Public[Array[field, 3]], _one: Public[field]) -> field:
    return reconstruct_secret(R, _one)
```

Figure 6.3: The sequential decomposition of the multi-party sum protocol for three parties, implemented in `decomposition/protocol.py`.

compilable and can be wrapped into the ZK-statements $R^{-}_{auth}(\Pi^0_f)$, $R^{+}_{auth}(\Pi^1_f)$ and $R^{+}_{auth}(\Pi^2_f)$, as shown in Figure 6.4. It should be noted that the ZK-statement implementations deviate from Definition 4.2.3 in a couple of ways. First, the function `auth_protocol_0` does not

[1]This is described in more detail in Section B.2 of the appendix.

```
from zkpytoolkit.types import Private, Public, Array, field          # ZK_IGNORE
from zkpytoolkit.stdlib.commitment.pedersen.ristretto255.commit import
    commit_field as commit
from ..decomposition.protocol import protocol_0, protocol_1, protocol_2

N: int = 3                                          # Hardcoded number of parties

def auth_protocol_0(
    x: Private[field], U: Private[Array[field, 2]],
    i: Public[int], _one: Public[field],
    blinding_factors: Private[Array[field, 6]]
) -> Array[Array[int, 8], 3]:
    commits: Array[Array[int, 8], 3] = [[0 for _ in range(8)] for _ in range(N)]
    outputs: Array[field, 3] = protocol_0(x, U, i, _one)
    for j in range(N):
        commits[j] = commit(outputs[j], blinding_factors[2*j:2*j+2])
    return commits

def auth_protocol_1(
    S: Private[Array[field, 3]], _one: Public[field],
    blinding_factors: Private[Array[field, 6]],
    commits: Public[Array[Array[int, 8], 3]]
) -> field:
    for i in range(N):
        assert commits[i] == commit(S[i], blinding_factors[2*i:2*i+2]), "Abort"
    return protocol_1(S, _one)

def auth_protocol_2(
    R: Public[Array[field, 3]], _one: Public[field]
) -> field:
    return protocol_2(R, _one)
```

Figure 6.4: The ZK-statements for protocol authentication of each sub-protocol, implemented in `zk_statements/protocol.py`.

assert that `x` is correct with respect to its commitment. While it is expected by definition, it is not a requirement for active security since the secret shares count as the initial private input in the eyes of the recipients. Second, this function also includes a public input `i`, which denotes the caller's index and is not accounted for in the definition of the statement. In fact, this input should be considered as a private input, but since all parties know each other's index it simply serves as an explicit check. Lastly, since `protocol_2` only takes public inputs, a proof of the function `auth_protocol_2` would be a tautology, making it meaningless to check. This is also apparent from the sub-protocol itself since all parties would have access to the same public information, and could run and check the function locally without requiring a ZKP.

The implementations of the passively-secure protocol and its active security transformation follow somewhat trivially from the explanation given in Section 5.1.3 and are provided in Listings A.2 and A.3 in the appendix, respectively.

To summarize, this case study shows that compiling a passively-secure protocol using the proof-of-concept compiler is straightforward and can be achieved systematically. The downsides of the current approach are that protocol implementations can only be provided for fixed-size parties, and that ZK-statements and both the passively-secure protocol composition and active security transformation must be implemented manually[2].

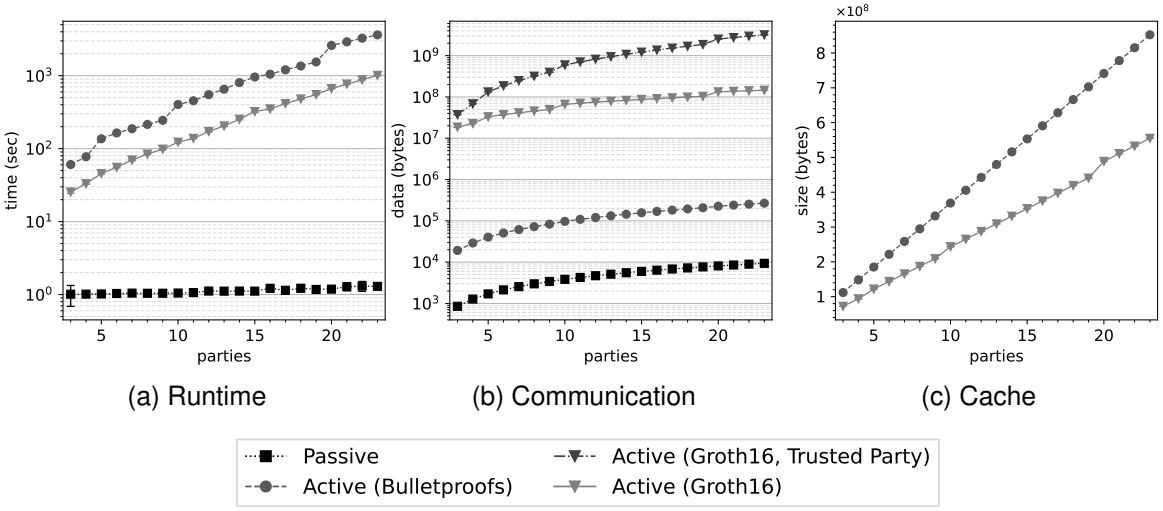
[2]The programmer adapts and uses the boilerplate code from Example 5.1.1 and Listing A.1 in the appendix.

6.3.3 Benchmark of Proof-of-Concept Implementation

Given the implementation of the passively-secure multi-party sum protocol (Figures 6.1, 6.3 and Listing A.2) and the compiled actively-secure protocol (Listing A.3), we carry out benchmarks according to the experiments listed in Section 6.3.1. All experiments are conducted for $n = 3, 4, \ldots, 23$ parties and run on separate cores distributed among 5 separate single-threaded nodes[3] running an AMD 7763 CPU with 64 GB of shared memory, clocked at 2.45 GHz. Additionally, all nodes are connected via an internal 25 Gbps network, ensuring that there is minimal runtime overhead from communication attributed to network latency.

Overall Performance

In Figure 6.5 we see semi-log plots of the median measurement over 10 repetitions for total runtime, communication and cache against a varying number of parties running the passively and actively-secure protocol. The actively-secure protocol is configured to use either



(a) Runtime (b) Communication (c) Cache

Figure 6.5: Median total runtime, communication, and cache size of the multi-party sum protocol, comparing an instance with passive security vs active security using Bulletproofs and Groth16 across an increasing number of parties.

the Groth16 or Bulletproofs ZKP system, and includes distinct measurements of communication for the trusted party. We first comment on each of the plots to gain surface-level insight into the performance of the passively-secure vs compiled protocol, and then delve into the specific details for each type of measurement.

With regards to the total runtime, is clear that the passively-secure protocol is close to constant and always less than 2 seconds, while both the Groth16 and Bulletproofs-driven actively-secure protocols exhibits superpolynomial growth going from 20 seconds to well over an hour, which would not align with Hypothesis 2. Moreover, it appears that using Bulletproofs results in a runtime that grows at a rate roughly half an order of magnitude slower than Groth16, and also appears to repeatedly jump after a specific increase in the number of parties, which confirms Hypothesis 3.

For total communication, the size appears to exhibit polynomial growth in all cases, confirming Hypothesis 1, with the passively-secure protocol consistently requiring less than 10 kilobytes of data. For Bulletproofs, the size is in the low to mid kilobytes range, while for Groth16 it is in the low megabytes to low gigabytes range, which is surprising and does not

[3] A node is an individual computer within a high-performance computing cluster.

align with Hypothesis 3. There is also a steep increase in communication by the trusted party vs everyone else.

Finally, the cache size appears to grow linearly for both types of actively-secure protocols, ranging between 0.1 and 1 gigabyte of information. It also appears that the compiled protocol that uses Groth16 caches substantially less information than Bulletproofs.

Runtime Analysis

Since the total runtime measurements of the actively-secure protocol do not agree with Hypothesis 2, it is likely that there is an additional overhead introduced during the setup phase which is not accounted for. We dissect the runtime measurements over the different compiler phases in Figure 6.6 to see this. It is clear that the setup phase, which involves the ZKP

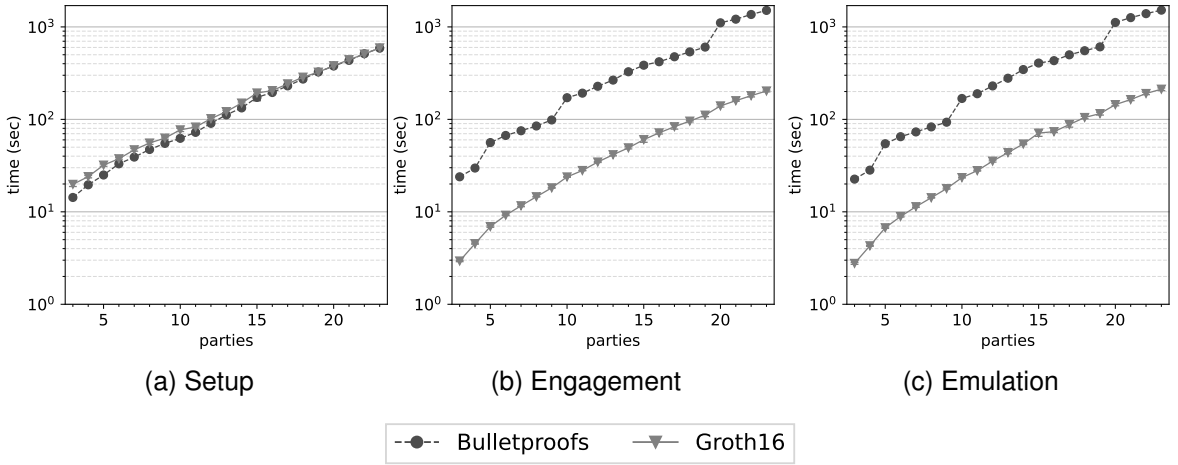


Figure 6.6: Median runtimes of the active security compiler phases using Groth16 or Bulletproofs across an increasing number of parties.

compiler and CRS generation components, appears to run in subexponential time, while the engagement and emulation phases, which involve the ZKP proof generation and verification components, run in polynomial time. Interestingly, the setup time for both Groth16 and Bulletproofs appear to converge, while in the engagement and emulation phases the runtime is roughly an order of magnitude slower in the case of Bulletproofs. Moreover, it appears that both the engagement and emulation phases have similar runtimes, which can be explained by the fact that the number of R1CS constraints involved in protocol authentication is evenly split among both phases, as can be seen in Table C.1 in the appendix.

To better understand what the runtime distributions for the different compiler phases look like using Groth16 and Bulletproofs, we take a look at the relative runtimes with respect to the median total runtime measurement in Figure 6.7. When using Groth16, the setup phase forms a major computational bottleneck, whereas for Bulletproofs this is dominated by the engagement and emulation phases. This can be further explained by the large absolute difference in runtimes when comparing the engagement and emulation phases between the two ZKP systems.

To understand why the three compiler phase runtimes grow at vastly different rates, we go one level further and compare the runtimes of the relevant components in Figure 6.8. In the setup phase we see that the ZKP compiler, when used for Bulletproofs, is consistently slower at a small constant rate when compared to Groth16. This could be explained by the different implementations of the BLS12-381 and Ristretto255 curves, where group operations are handled differently and likely less efficiently in the latter case. Nevertheless, it is

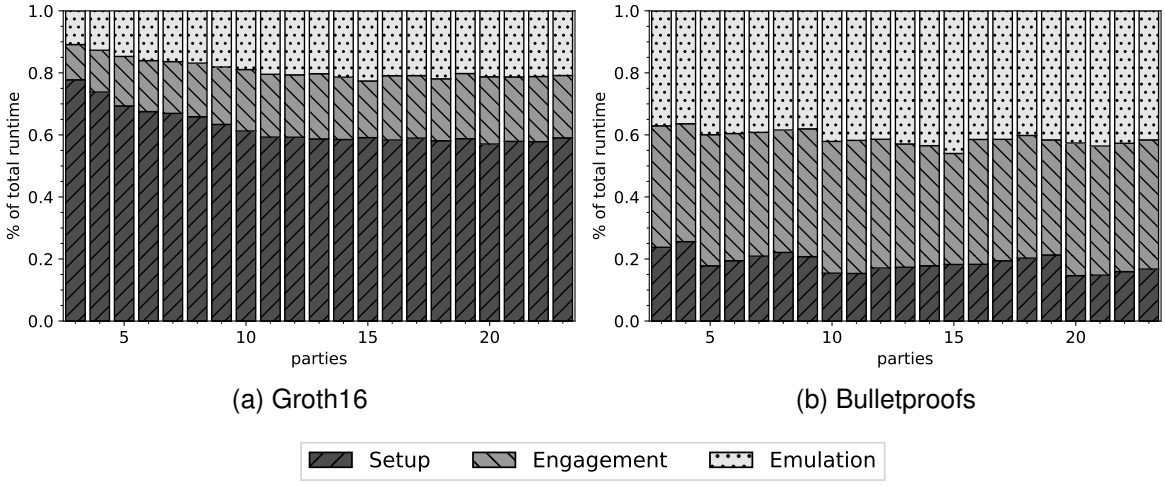


Figure 6.7: Relative median runtimes of the active security compiler phases across an increasing number of parties, comparing Groth16 and Bulletproofs.

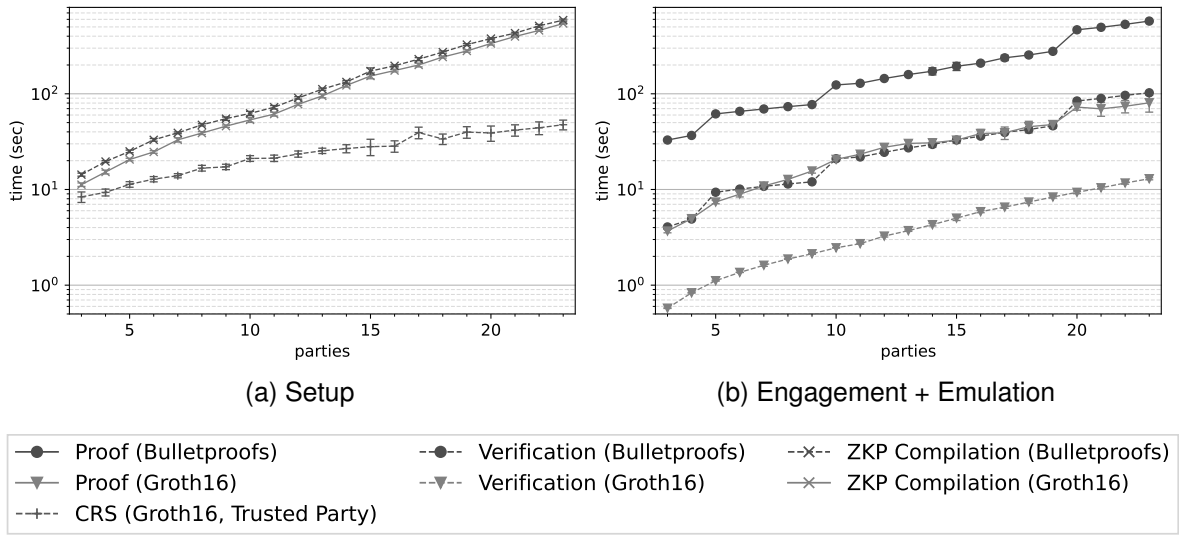


Figure 6.8: Median runtimes of the active security compiler ZKP components for Groth16 and Bulletproofs across an increasing number of parties.

clear that the ZKP compiler is responsible for the subexponential setup phase runtime, and that the CRS generation component contributes a small linear overhead. In the engagement and emulation phases, we see both the proof generation and verification runtimes for the Groth16 and Bulletproofs ZKP systems, which appear to run in polynomial time. We note that the proof generation time using Bulletproofs is far slower in comparison to the proof generation time using Groth16, which coincidentally overlaps with the average verification time per party using Bulletproofs. Surprisingly, the Groth16 verification runtime appears to run in polynomial time as opposed to being near-constant. Since the absolute runtime is relatively small (max $\sim$10 seconds), it is likely dominated by intermediate proccesses such as R1CS witness evaluation and zkInterface data generation/parsing, which are not accounted for in the computational complexity.

Lastly, we look at the runtime distributions for the different compiler components, including communication, when using Groth16 vs Bulletproofs in Figure 6.9. We note that the runtime distributions are very similar to those in Figure 6.7 due to separation of ZKP compilation and ZKP proof/verification into the setup and engagement/emulation phases, respectively.

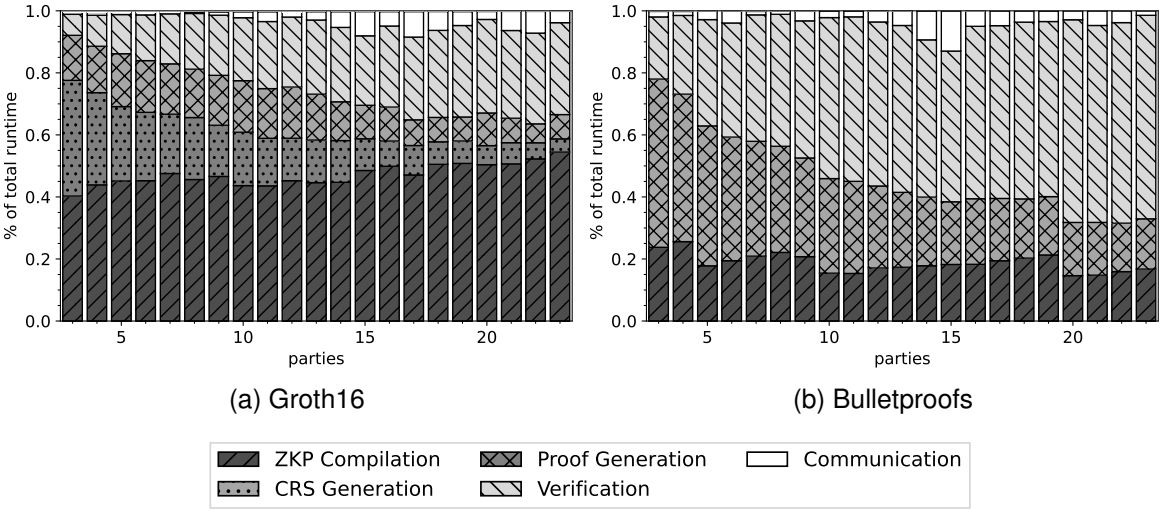


Figure 6.9: Relative median runtimes of the active security compiler components across an increasing number of parties, comparing Groth16 and Bulletproofs.

It is clear that when using Groth16, the ZKP compiler forms the major bottleneck, whereas for Bulletproofs this is mainly attributed to proof generation and verification. We see that in both cases, the total verification time overtakes the proof generation time (only after 6 parties), which can be explained by the increasing number of verifications that are required by each party. Moreover, we note that total communication, which we did not factor into the computational complexity, appears to be noticeably costly at certain points.

All in all, when separating the compiler phases into “offline” and “online”, these findings show that the ZKP compiler forms a bottleneck in the “offline” phase, while repeated verification forms a bottleneck in the “online” phase of the compiled protocol.

Communication Analysis

While the total communication measurements confirm Hypothesis 1, the discrepancy with Hypothesis 3 warrants further analysis. In Figure 6.10 we divide the communication mea-

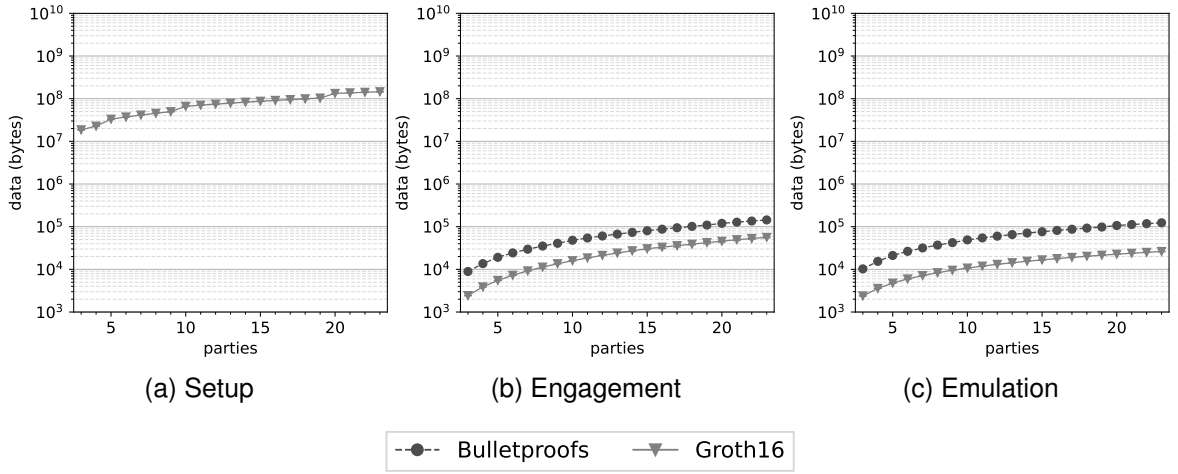


Figure 6.10: Median communications during the active security compiler phases using Groth16 and Bulletproofs across an increasing number of parties.

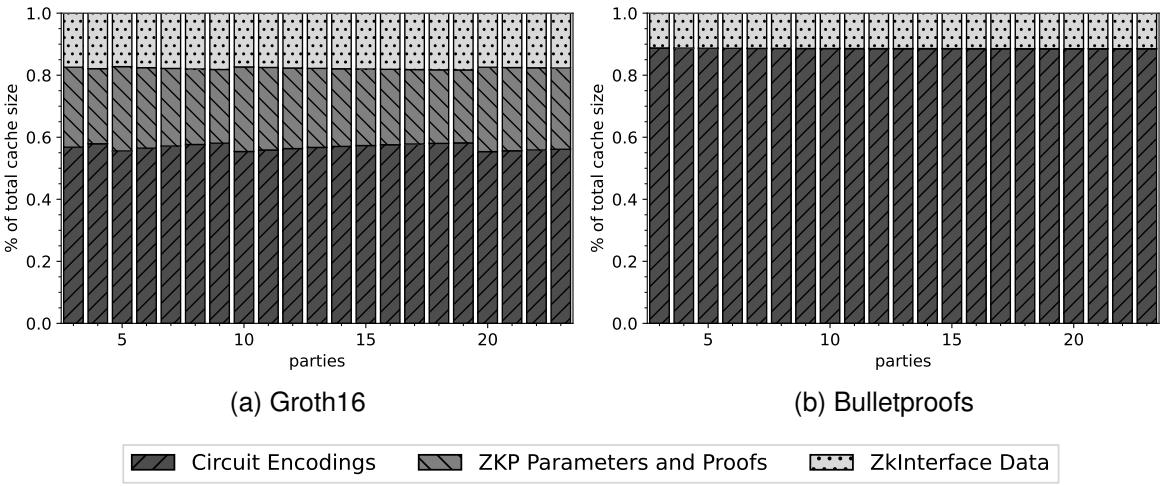
surements into their constituent compiler phases, and see that the costly communication using Groth16 can be attributed to the setup phase, and that Hypothesis 3 does indeed hold

for the engagement and emulation phases. We recall that communication is only required during the setup phase whenever a ZKP system requires a trusted setup, which suggests that the communication of CRS data forms the main bottleneck.

ZKP Component Analysis

From the runtime and communication analysis we were able to infer that while the empirical measurements do confirm the hypotheses when excluding the setup phase, the active security compiler induces very large overheads when run using either of the ZKP systems, particularly for the runtime. Since the ZKP proof generation and verification algorithms dominate the computational costs, which depend on the circuit for the ZK-statement, it is likely that the compiler performance suffers as a result of the circuit sizes being too large.

We saw from Figure 6.5c that the total cache starting at 3 parties incurs a large cost ($\sim$ 0.1 gigabytes). In Figure 6.11 we see and compare the cache sizes of its three constituents for



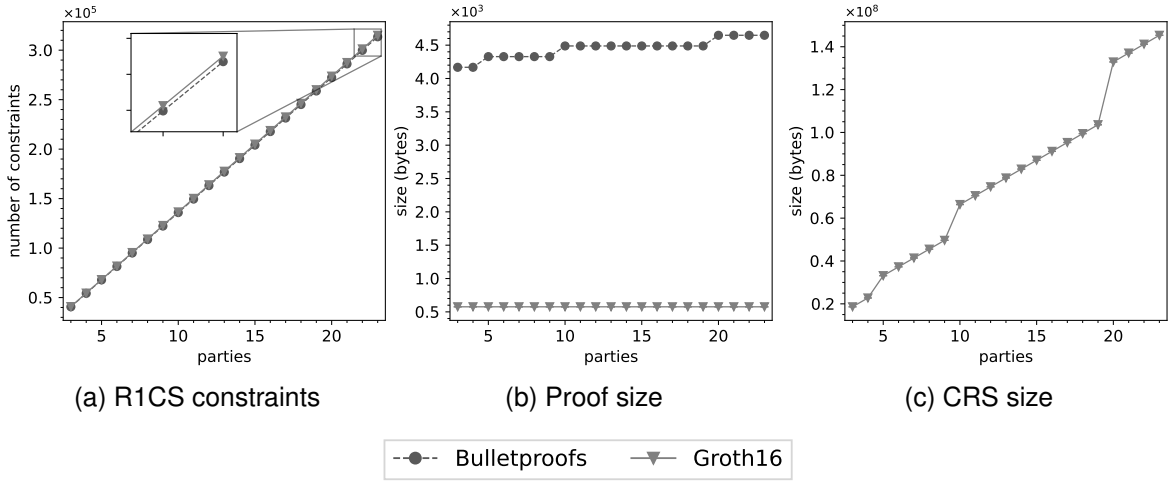
(a) Groth16 (b) Bulletproofs

Figure 6.11: Relative median cache size of the active security compiler serializations, of the compiled multi-party sum protocol using Groth16 and Bulletproofs across an increasing number of parties.

Groth16 and Bulletproofs: The circuit encodings that store the raw R1CS constraints and witness evaluation information, the ZKP parameters and proofs required by the ZKP systems, and the zkInterface data that allows the ZKP back-end and front-end to communicate. Since the circuit encodings take up more space than the zkInterface data, this suggests that the circuit for evaluating the witnesses may be significantly larger[4] than the resulting R1CS constraints. We also note that the circuit encodings for Bulletproofs are significantly larger than Groth16, which is not the case for the zkInterface data. This is likely due to the more 'efficient' implementation of the BLS12-381 primefield vs the one for Ristretto255, as is also reflected by the runtimes in Figure 6.8a.

We measure the sizes of the cached ZKP components, which are the total number of R1CS constraints, the resulting proof size and the CRS data sizes in Figure 6.12. We note that the number constraints ranges between 4×10^4 and 3.2×10^5 and increases linearly in the number of parties. The large size is explained by the relatively expensive subcircuits that encode the individual Pedersen commitments, which are required for each input and hence increases the total number of constraints linearly. We note that this number is slightly larger for Groth16 than it is for Bulletproofs, which is likely explained by field size of the *Jubjub* group ($\sim$ 255 bits) being larger than the *Doppio* group ($\sim$ 252 bits).

[4] An R1CS constraint is expressed as a linear combination of multiple addition and multiplication gates.



(a) R1CS constraints (b) Proof size (c) CRS size

Figure 6.12: Cached ZKP data sizes of the compiled protocol across an increasing number of parties.

Next, we see that the total proof sizes for Groth16 are always constant at 576 bytes, whereas Bulletproofs is over 7 times larger, starting at 4168 bytes and increasing by 160 bytes at 5, 10 and 20 parties. This is clearly reflective of the proof size complexity of both ZKP systems (Table 6.2), where the Bulletproofs logarithmic growth resembles that of a discrete logarithm, and therefore also results in characteristic jumps in the runtime measurements.

Finally, we see that the CRS size ranges between 20 and 150 megabytes, and increasingly increments at the aforementioned points, which explains the large communication seen in Figure 6.10a when using Groth16.

Measuring Constraints

We determined that the large number of constraints are the primary cause for the slow runtimes seen in Figure 6.5a, and speculated that the expensive Pedersen commitments in the protocol authentication ZK-statements are the culprit. We can infer the overhead from commitments in the number of constraints by comparing protocol evaluation (Definition 4.2.2) to protocol authentication (Definition 4.2.3). A detailed comparison is provided in Table C.1 in the appendix, but the key results are summarized in Table 6.7. We note that eliminating Ped-

Table 6.7: Number of constraints for Π_{SUM} with and without commitments.

Protocol	Protocol Evaluation					Protocol Authentication				
Parties	3	4	5	...	23	3	4	5	...	23
Π^0_{SUM}	46	91	144	...	2103	20455	27303	34157	...	158572
Π^1_{SUM}	1	1	1	...	1	20410	27213	34016	...	156470
Π^2_{SUM}	1	1	1	...	1	1	1	1	...	1
Π_{SUM}	48	93	146	...	2105	40866	54517	68174	...	315043

ersen commitments from the ZK-statement results in the number of constraints being orders of magnitudes smaller, which suggests that utilizing more efficient commitment schemes or ideally eliminating them from the ZK-statement would vastly improve the performance.

7 Conclusions

7.1 Conclusion

The challenge of transforming a passively secure MPC protocol into an actively secure one fundamentally involves preventing any malicious adversary from deviating from the protocol. This objective can be achieved by appending a ZKP to every message based on the initially agreed input, a transformation achieved by the GMW compiler. This thesis has demonstrated that, with certain modifications, the GMW compiler provides practical benefits beyond its theoretical significance, particularly for special-purpose protocols, which would require a tailored approach to achieve active security.

Conceptually, our compiler design closely mirrors the original GMW compiler, which is well-known for providing active security with abort. We have enhanced the GMW compiler to support initial point-to-point communication, thereby allowing the compilation of secret-sharing-based protocols without emulating point-to-point communication via a secure broadcast channel. This enhancement removes the necessity for assumptions such as the existence of a PKI. Operating within the synchronous communication model, we framed the problem of proving the correctness of the next-message function as a proof of valid sub-protocol execution at the corresponding round. This approach makes the proofs compatible with a variety of general-purpose ZKP systems, demonstrating practical interoperability. Moreover, our construction of the input-commitment and coin-tossing protocols requires only a single round of communication without requiring proofs of correct execution, further streamlining the compiler.

From a practical perspective, the implementation of the compiler involved creating a software design that automatically generates the necessary ZKPs and integrates them into a protocol runtime. The proof-of-concept was developed to support a substantial subset of Python 3.10, showcasing the potential of incorporating the GMW paradigm into existing MPC frameworks. Through zkInterface, we have demonstrated how easy it is to utilize two popular yet vastly different ZKP systems, Groth16 and Bulletproofs, to power the active-security compiler. Our benchmarks indicate that while the compiler is effective for small-scale MPC, its concrete runtime efficiency diminishes as the number of parties increases. However, with some adjustment of the ZK-statements and a better selection of ZKP systems, the compiler's performance could be significantly enhanced, suggesting a promising step towards a practical active-security compiler for special-purpose protocols.

7.2 Discussion

In this section, we discuss some limitations and areas for improvement in this work.

Communication model. The current communication model is restricted to the broadcast-only and initial point-to-point communication model. While it better aligns with the original

GMW compiler, it is not compatible with all MPC protocols. Investigating the use of multi-prover ZKPs could be a starting point for lifting the GMW paradigm into the point-to-point communication model.

Expresiveness of the ZKPyC language. Considering non-uniform circuits allows for creating more efficient ZK-statements, particularly as an instance of R1CS. However, it limits the expressiveness of programming languages for implementing MPC protocols. While it is in theory sufficient for MPC, it would require end-users to rewrite code to avoid variable-bounded statements. Considering uniform circuits would make it possible to prove stateful MPC protocols, thus requiring less code refactoring.

ZK-statements. The evaluation showed that the generated ZK-statements are very large, which form a culprit in the proof and verification runtimes. Eliminating proof of commitment opening in the ZK-statement would in many cases reduce the number of constraints and thus improve the runtime by orders of magnitude. This could likely be achieved using a commit-and-prove style ZKP system, such as LegoSNARKs.

Proof of concept. The current implementation uses the non-homomorphic windowed Pedersen commitment scheme and lacks a coin-tossing protocol. Implementing the homomorphic variant and implementing a robust coin-tossing protocol would give better insight into the practical utility of our active-security compiler.

ZK-statement generation. ZK-statement generation and a part of the active-security compiler currently require manual programming, albeit based on boilerplate code. Automating these processes is possible and would make the compiler more user-friendly and therefore more practical.

Communication overhead and runtime. There is a trade-off between communication overhead and runtime when using different ZKP systems. Groth16 has an acceptable runtime but an unacceptable communication overhead due to the trusted setup. Bulletproofs offer an acceptable communication overhead but have an unacceptable runtime. By extending the compiler to other ZKP systems, it should be possible to find a suitable system that offers an optimal overhead.

7.3 Future Work

This thesis has laid a foundation for the development of an active-security compiler for MPC protocols, particularly special-purpose ones, but there are several directions for future research that could enhance both the conceptual and practical aspects of this work.

7.3.1 Conceptual Directions

Commit-and-prove techniques. Utilizing a commit-and-prove style ZKP system could significantly reduce the number of R1CS constraints, leading to more efficient proofs. Additionally, it would enable the creation of a UC-secure variant of the active-security compiler, allowing for the compilation of passively secure protocols in the asynchronous communication model. This would broaden the applicability of the compiler and enhance its security guarantees.

Proof-carrying data. Designing a GMW-style compiler that leverages proof-carrying data could achieve more efficient ZKPs. Proof-carrying data naturally integrates MPC protocols with ZKPs, and reduces proof sizes and verification times due to their recursive-compositional structure.

Covert security. Exploring the relaxation of active security to covert security could lead to a more efficient compiler. Covert security, which assumes that adversaries are unlikely to cheat if there is a chance of being detected, may allow for more efficient ZKP constructions in the GMW paradigm.

7.3.2 Practical Directions

Public auditability. Integrating a NIZK system with a transparent setup into the active-security compiler should, in theory, allow for public auditability. Future research should focus on designing and implementing a concrete and practical version of such a compiler. Public auditability would allow third parties to verify the correctness of the computations without compromising the security of the underlying data.

Batch verification. Although we did not initially consider this, Bulletproofs support batch verification, which allows for more efficient verification of multiple prover statements with respect to the same ZK-statement. Investigating the speedup achieved by applying batch verification in the active-security compiler could yield a significant improvement in performance when the number of parties is large.

Computer-aided cryptography. The CirC infrastructure, used for developing the ZKP compiler, can utilize an SMT solver for circuit optimization. SMT solvers could bridge the gap between practical MPC and computer-aided cryptography. Logically determining if a protocol is passively secure before running the compiler is akin to carrying out a syntax check on a program's source code. This approach would not only guarantee active security, but also ensure that the input, i.e., the passively secure protocol, is correct.

Bibliography

[AC20] Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. (12172), August 2020. doi:10.1007/978-3-030-56877-1_18.

[ACC+18] Abdelrahaman Aly, Kelong Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE–MAMBA. COSIC KU Leuven, 2018. URL: https://nigelsmart.github.io/SCALE/Documentation.pdf.

[ACGJ18] Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Round-Optimal Secure Multiparty Computation with Honest Majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 395–424, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96881-0_14.

[ADEL22] Thomas Attema, Vincent Dunning, Maarten Everts, and Peter Langenkamp. Efficient Compiler to Covert Security with Public Verifiability for Honest Majority MPC, 2022. URL: https://eprint.iacr.org/2022/454.

[Adl79] Leonard Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*, pages 55–60, 1979. doi:10.1109/SFCS.1979.2.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2087–2104, New York, NY, USA, October 2017. Association for Computing Machinery. doi:10.1145/3133956.3134104.

[AKP22] Benny Applebaum, Eliran Kachlon, and Arpita Patra. Round-Optimal Honest-Majority MPC in Minicrypt and with Everlasting Security. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography*, pages 103–120, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-22365-5_4.

[AKP23] Benny Applebaum, Eliran Kachlon, and Arpita Patra. The Round Complexity of Statistical MPC with Optimal Resiliency. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 1527–1536, New York, NY, USA, June 2023. Association for Computing Machinery. doi:10.1145/3564246.3585228.

[AO12] Gilad Asharov and Claudio Orlandi. Calling Out Cheaters: Covert Security with Public Verifiability. In Xiaoyun Wang and Kazue Sako, editors, *Advances*

in Cryptology – ASIACRYPT 2012, pages 681–698, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-34961-4_41.

[BBB+18] Benedikt Bunz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, San Francisco, CA, May 2018. IEEE. doi:10.1109/SP.2018.00020.

[BBH+10] Endre Bangerter, Thomas Briner, Wilko Henecka, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. Automatic Generation of Sigma-Protocols. In Fabio Martinelli and Bart Preneel, editors, *Public Key Infrastructures, Services and Applications*, Lecture Notes in Computer Science, pages 67–82, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-16441-5_5.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable Zero Knowledge with No Trusted Setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 701–732, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-26954-8_23.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, October 1988. doi:10.1016/0022-0000(88)90005-0.

[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 Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03549-4_20.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-Carrying Data Without Succinct Arguments. In *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I*, pages 681–710, Berlin, Heidelberg, August 2021. Springer-Verlag. doi:10.1007/978-3-030-84242-0_24.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly Auditable Secure Multi-Party Computation. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 175–196, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10879-7_11.

[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*, pages 169–188, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-20465-4_11.

[Bea92] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer. doi:10.1007/3-540-46766-1_34.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958, pages 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/11745853_14.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing - STOC '88*, pages 103–112, Chicago, Illinois, United States, 1988. ACM Press. doi:10.1145/62212.62222.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357, Farminton Pennsylvania, November 2013. ACM. doi:10.1145/2517349.2522733.

[BG92] Mihir Bellare and Oded Goldreich. On Defining Proofs of Knowledge. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 390–420, Berlin, Heidelberg, 1992. Springer. doi:10.1007/3-540-48071-4_28.

[BG20] Sean Bowe and Jack Grigg. Bellman: Zk-SNARK library. GitHub, November 2020. URL: https://github.com/dalek-cryptography/bulletproofs.

[BGIN21] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear GMW-Style Compiler for MPC with Preprocessing. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, Lecture Notes in Computer Science, pages 457–485, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-84245-1_16.

[BGK+20] Daniel Benarroch, Kobi Gurkan, Ron Kahat, Aurélien Nicolas, and Eran Tromer. Community Proposal: zkInterface, a standard tool for zero-knowledge interoperability, February 2020. URL: https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-zkinterface.pdf.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, January 1988. Association for Computing Machinery. doi:10.1145/62212.62213.

[BIM+23] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6):4733–4751, November 2023. doi:10.1109/TDSC.2022.3232813.

[BL17] Daniel J. Bernstein and Tanja Lange. SafeCurves: Choosing safe curves for elliptic-curve cryptography., January 2017. URL: https://safecurves.cr.yp.to/.

[Blu81] Manuel Blum. Coin flipping by telephone. In *Advances in Cryptology: A Report on CRYPTO 81*, pages 11–15, 1981. URL: /archive/crypto81/11_blum.pdf.

[BM89] Mihir Bellare and Silvio Micali. Non-Interactive Oblivious Transfer and Applications. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 547–557, New York, NY, 1989. Springer. doi:10.1007/0-387-34805-0_48.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 92–122, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-84259-8_4.

[BMRS23] Carsten Baum, Nikolas Melissaris, Rahul Rachuri, and Peter Scholl. Cheater Identification on a Budget: MPC with Identifiable Abort from Pairwise MACs, 2023. URL: https://eprint.iacr.org/2023/1548.

[BMY24] Marina Blanton, Dennis Murphy, and Chen Yuan. Efficiently Compiling Secure Computation Protocols From Passive to Active Security: Beyond Arithmetic Circuits. *Proceedings on Privacy Enhancing Technologies*, 2024(1):74–97, January 2024. doi:10.56553/popets-2024-0006.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security - CCS '93*, pages 62–73, Fairfax, Virginia, United States, 1993. ACM Press. doi:10.1145/168588.168596.

[But16] Vitalik Buterin. Quadratic Arithmetic Programs: From Zero to Hero, December 2016. URL: https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, October 2001. doi:10.1109/SFCS.2001.959888.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, New York, NY, USA, January 1988. Association for Computing Machinery. doi:10.1145/62212.62214.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 280–300, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-44987-6_18.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2075–2092, New York, NY, USA, November 2019. Association for Computing Machinery. doi:10.1145/3319535.3339820.

[CHI+23] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Daniel Genkin, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious

Adversaries. *Journal of Cryptology*, 36(3), April 2023. doi:10.1007/s00145-023-09453-7.

[CKMZ14] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient Three-Party Computation from Cut-and-Choose. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 513–530, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-44381-1_29.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 494–503, New York, NY, USA, May 2002. Association for Computing Machinery. doi:10.1145/509907.509980.

[CWC+21] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications, 2021. URL: https://eprint.iacr.org/2021/651.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 418–430, Berlin, Heidelberg, 2000. Springer-Verlag.

[DI05] Ivan Damgård and Yuval Ishai. Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 378–394, Berlin, Heidelberg, 2005. Springer. doi:10.1007/11535218_23.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 558–576, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-14623-7_30.

[DOS18] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*, pages 799–829, Berlin, Heidelberg, August 2018. Springer-Verlag. doi:10.1007/978-3-319-96881-0_27.

[DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-Box Transformations from Passive to Covert Security with Public Verifiability. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 647–676, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-56880-1_23.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-32009-5_38.

[DR22] Thomas Debris-Alazard and Nicolas Resch. Worst and Average Case Hardness of Decoding via Smoothing Bounds, 2022. URL: https://eprint.iacr.org/2022/1744.

[DS83] D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983. doi:10.1137/0212045.

[dY19] Henry de Valence and Cathie Yun. The Doppio Group, 2019. URL: https://doppio.group/index.html.

[dYA21] Henry de Valence, Cathie Yun, and Oleg Andreev. Bulletproofs: A pure-Rust implementation of Bulletproofs using Ristretto. GitHub, March 2021. URL: https://github.com/dalek-cryptography/bulletproofs.

[Edw07] Harold Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, April 2007. doi:10.1090/S0273-0979-07-01153-6.

[EKRN24] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. Bulletproofs++: Next Generation Confidential Transactions via Reciprocal Set Membership Arguments. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 249–279, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-58740-5_9.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi:10.1109/TIT.1985.1057074.

[ET18] Jacob Eberhardt and Stefan Tai. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, July 2018. doi:10.1109/Cybermatics_2018.2018.00199.

[FHKS21] Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic Compiler for Publicly Verifiable Covert Multi-Party Computation. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 782–811, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-77886-6_27.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[Gal62] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962. doi:10.1109/TIT.1962.1057683.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure com-

putation. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 495–504, New York, NY, USA, May 2014. Association for Computing Machinery. doi:10.1145/2591796.2591861.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient Multi-party Computation: From Passive to Active Security via Secure SIMD Circuits. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9216, pages 721–741. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. doi:10.1007/978-3-662-48000-7_35.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing - STOC '82*, pages 365–377, San Francisco, California, United States, 1982. ACM Press. doi:10.1145/800070.802212.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 1069–1083, USA, 2016. USENIX Association.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. arXiv:https://doi.org/10.1137/0218012, doi:10.1137/0218012.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)*, pages 174–187, October 1986. doi:10.1109/SFCS.1986.47.

[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 V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987. doi:10.1145/28395.28420.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, England, 2001.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, England, 2004.

[Gro16] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, Lecture Notes in Computer Science, pages 305–326, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49896-5_11.

[GTK16] Shafi Goldwasser and Yael Tauman Kalai. Cryptographic Assumptions: A Position Paper. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 505–522, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49096-9_21.

[HBHW22] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2022. URL: https://zips.z.cash/protocol/protocol.pdf.

[HKR19] Max Hoffmann, Michael Klooß, and Andy Rupp. Efficient zero-knowledge arguments in the discrete log setting, revisited, 2019. URL: https://eprint.iacr.org/2019/944.

[HM00] Martin Hirt and Ueli Maurer. Player Simulation and General Adversary Structures in Perfect Multiparty Computation. *Journal of Cryptology*, 13(1):31–60, January 2000. doi:10.1007/s001459910003.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, pages 21–30, San Diego California USA, June 2007. ACM. doi:10.1145/1250790.1250794.

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147, 1995. doi:10.1109/SCT.1995.514853.

[IOS12] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying Cheaters without an Honest Majority. In Ronald Cramer, editor, *Theory of Cryptography*, pages 21–38, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-28914-9_2.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure Multi-Party Computation with Identifiable Abort. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 369–386, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-44381-1_21.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding Cryptography on Oblivious Transfer – Efficiently. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 572–591, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-85174-5_32.

[JS20] Jan Jancar and Vladimir Sedlacek. Standard curve database, 2020. URL: https://neuromancer.sk/std/.

[Kel20] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1575–1590, New York, NY, USA, November 2020. Association for Computing Machinery. doi:10.1145/3372297.3417872.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 549–560, New York, NY, USA, November 2013. Association for Computing Machinery. doi:10.1145/2508859.2516744.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and Robust Multi-party Computation Using a Global Transaction Ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 705–734, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49896-5_25.

[Lin17] Yehuda Lindell. How to simulate it – a tutorial on the simulation proof technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-57048-8_6.

[Lin21] Yehuda Lindell. Secure multiparty computation. *Communications of the ACM*, 64(1):86–96, January 2021. doi:10.1145/3387108.

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS Compiler: Optimizations, Variants and Concrete Efficiency. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science, pages 259–276, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22792-9_15.

[LP07] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, pages 52–78, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-72540-4_4.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-13190-5_1.

[McE78] Robert J. McEliece. A public key cryptosystem based on algebraic coding theory. 1978.

[MEK+10] Sarah Meiklejohn, C. Chris Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 13, USA, 2010. USENIX Association.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266, San Francisco, CA, USA, May 2022. IEEE. doi:10.1109/SP46214.2022.9833782.

[OWBB23] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 154–175, Cham, 2023. Springer Nature Switzerland.

[Ozd23] Alex Ozdemir. CirC: The Circuit Compiler. GitHub, 2023. URL: https://github.com/circify/circ.

[Ped92] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer. doi:10.1007/3-540-46766-1_9.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, May 2013. doi:10.1109/SP.2013.47.

[Pip80] Nicholas Pippenger. On the Evaluation of Powers and Monomials. *SIAM Journal on Computing*, 9(2):230–250, May 1980. doi:10.1137/0209022.

[Pyt21a] Python Software Foundation. Ast — Abstract Syntax Trees, 2021. URL: https://docs.python.org/3.10/library/ast.html.

[Pyt21b] Python Software Foundation. Full Grammar specification, 2021. URL: https://docs.python.org/3.10/reference/grammar.html.

[Rot24a] Lorenzo Rota. ActiveSecurityMPC. GitHub, January 2024. URL: https://github.com/lorenzorota/activesecuritympc/.

[Rot24b] Lorenzo Rota. ZKPyC: The Zero-Knowledge Proof Compiler for Python. GitHub, January 2024. URL: https://github.com/lorenzorota/zkpyc.

[Rot24c] Lorenzo Rota. ZKPyToolkit: The Zero-Knowledge Proof Toolkit for Python. GitHub, January 2024. URL: https://github.com/lorenzorota/zkpytoolkit.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. arXiv:https://doi.org/10.1137/0108018, doi:10.1137/0108018.

[rus23] rust-unofficial. Visitor. In *Rust Design Patterns*. 2023. URL: https://rust-unofficial.github.io/patterns/patterns/behavioural/visitor.html.

[RW19] Dragos Rotaru and Tim Wood. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 227–249, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-35423-7_12.

[Sch91] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. doi:10.1007/BF00196725.

[Sch18] Berry Schoenmakers. MPyC – Secure Multiparty Computation in Python., May 2018. URL: https://github.com/lschoe/mpyc/tree/v0.3.

[Sch23] Berry Schoenmakers. Lecture Notes Cryptographic Protocols. February 2023. URL: https://www.win.tue.nl/~berry/CryptographicProtocols/.

[Set20] Srinath Setty. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, Lecture Notes in Computer Science, pages 704–737, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-56877-1_25.

[SF16] Gabriele Spini and Serge Fehr. Cheater Detection in SPDZ Multiparty Computation. In Anderson C.A. Nascimento and Paulo Barreto, editors, *Information Theoretic Security*, pages 151–176, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-49175-2_8.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States, third edition, international edition edition, 2013.

[SSS22] Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty Computation with Covert Security and Public Verifiability. *LIPIcs, Volume 230, ITC 2022*, 230:8:1–8:13, 2022. doi:10.4230/LIPICS.ITC.2022.8.

[TNO21] TNO MPC Lab. TNO PET Lab - secure Multi-Party Computation (MPC). GitHub, April 2021. URL: https://github.com/TNO-MPC/.

[TNO23] TNO PET Lab. TNO PET lab - secure multi-party computation (MPC) - communication. GitHub, August 2023. URL: https://github.com/TNO-MPC/communication.

[Tv22] Eric Traut and Guido van Rossum. Type Parameter Syntax. PEP 695, 2022. URL: https://peps.python.org/pep-0695/.

[Val08] Paul Valiant. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78524-8_1.

[vDv+24] Marie Beth van Egmond, Vincent Dunning, Stefan van den Berg, Thomas Rooijakkers, Alex Sangers, Ton Poppe, and Jan Veldsink. Privacy-preserving anti-money laundering using secure multi-party computation. In *Financial Cryptography*, Lecture Notes in Computer Science. Springer, 2024.

[vLL15] Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. Type Hints. PEP 484, 2015. URL: https://peps.python.org/pep-0484/.

[WAKS97] Daniel C Wang, Andrew W Appel, Jeff L Korn, and Christopher S Serra. The Zephyr Abstract Syntax Description Language. 1997.

[WBB20] Barry WhiteHat, Marta Bellés, and Jordi Baylina. ERC-2494: Baby Jubjub Elliptic Curve, January 2020.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019:26–49, July 2019. doi:10.2478/popets-2019-0035.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1074–1091, San Francisco, CA, USA, May 2021. IEEE. doi:10.1109/SP40001.2021.00056.

[XZZ+19] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, Lecture Notes in Computer Science, pages 733–764, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-26954-8_24.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 160–164, Los Alamitos, CA, USA, November 1982. IEEE Computer Society. doi:10.1109/SFCS.1982.88.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)*, pages 162–167, 1986. doi:10.1109/SFCS.1986.25.

[ZoK23] ZoKrates. A toolbox for zkSNARKs on Ethereum. GitHub, April 2023. URL: https://github.com/Zokrates/ZoKrates/.

A

Active Security Compiler: Examples

A.1 Boilerplate Active Security Transformation Implementation

Listing A.1: Implementation of `active.py`, used in Example 5.1.2.

```
from zkpytoolkit import ZKP
from zkpytoolkit.types import field
from zkpytoolkit.stdlib.commitment.pedersen.ristretto255.commit import commit_field as commit
from active_security_mpc.template.protocol import ActiveProtocol
from .decomposition.protocol import protocol_0, protocol_1
from .zk_statements.protocol import auth_protocol_0, auth_protocol_1

zkp = ZKP._instance # defined globally across all modules at runtime
class Sum(ActiveProtocol):
    def __init__(self, local_idx, local_port, parties, enable_stats):
        super().__init__(local_idx, local_port, parties, enable_stats, field_type=field)

    async def setup(self):
        """Implementation of the setup phase"""
        N = self.parties

        # Step 0(a): Compiling the ZK-statements
        functions = [auth_protocol_0, auth_protocol_1]
        includes = [commit, N, protocol_0, protocol_1]
        self.compile_zkps(functions, includes, globals(), locals())

    async def engage(self, secret):
        """Protocol engagement"""
        i = self.local_idx
        N = self.parties

        # Step 2(b)i: Compute protocol_0 functionality
        outputs_0 = protocol_0(secret)

        # Step 2(b)ii: Compute commitments and blinding factors to outputs of protocol_0
        my_blindings = self.coin_flipping(2 * N)
        my_commitments = auth_protocol_0(secret, my_blindings)
        commitments = await self.communicate(my_commitments, "broadcast", "commitments", "int_list", flatten=True,
              unflatten=8)

        # Step 2(b)iii: Distribute outputs and blinding factors
        inputs_1 = await self.communicate(outputs_0, "distribute", "protocol_0", "field")
        blindings_1 = await self.communicate(my_blindings[0::2], "distribute", "blindings_1", "field")
        blindings_2 = await self.communicate(my_blindings[1::2], "distribute", "blindings_2", "field")
        blindings = [item for pair in zip(blindings_1, blindings_2) for item in pair]

        # Step 2(b)iv: Authenticate protocol_0
        args_prove = (secret, blindings)
        args_verify = (commitments, None, [None for _ in range(N)])
        await self.authenticate(auth_protocol_0, args_prove, args_verify)

        return inputs_1, blindings, commitments

    async def emulate(self, inputs, blindings, commitments):
        """Protocol emulation"""
        i = self.local_idx
        N = self.parties

        # re-organize commitments according to received secret shares
        incoming_commitments = [[comm[j] for comm in commitments] for j in range(N)]

        # Step 3(a): Compute and broadcast protocol_1 functionality
        output_1 = protocol_1(inputs)
        final_values = await self.communicate(output_1, "broadcast", "protocol_1", "field")

        # Step 3(b): Authenticate protocol_1
        args_prove = (input, blindings, incoming_commitments[i])
        args_verify = (final_values, [[None, None, incoming_commitments[j]] for j in range(N)])
        await self.authenticate(auth_protocol_1, args_prove, args_verify)

        assert(all(output == output_1 for output in final_values))
        print(output_1)
```

A.2 Multi-Party Sum Protocol Implementation

Listing A.2: Multi-party sum protocol as a sequential subprotocol composition

```
from zkpytoolkit.types import field
from active_security_mpc.template.protocol import PassiveProtocol
from .decomposition.protocol import protocol_0, protocol_1, protocol_2

class Sum(PassiveProtocol):
    def __init__(self, local_idx, local_port, parties, enable_stats):
        super().__init__(local_idx, local_port, parties, enable_stats, field_type=field)

    async def compose_protocol(self, secret):
        i = self.local_idx
        N = self.parties

        coins = self.coin_flipping(N - 1)

        outputs_0 = protocol_0(secret, coins, i, field(1))
        inputs_1 = await self.communicate(outputs_0, "distribute", "protocol_0", "field")

        output_1 = protocol_1(inputs_1, field(1))
        inputs_2 = await self.communicate(output_1, "broadcast", "protocol_1", "field")

        output_2 = protocol_2(inputs_2, field(1))
        final_outputs = await self.communicate(output_2, "broadcast", "protocol_2", "field")

        if all(output == output_2 for output in final_outputs):
            print(output_2)
        else:
            raise ValueError("Invalid output")
```

Listing A.3: Active security transformation of the multi-party sum protocol

```
from zkpytoolkit import ZKP
from zkpytoolkit.types import field
from zkpytoolkit.stdlib.commitment.pedersen.ristretto255.commit import commit_field as commit
from active_security_mpc.template.protocol import ActiveProtocol
from .decomposition.protocol import protocol_0, protocol_1, protocol_2
from .zk_statements.protocol import auth_protocol_0, auth_protocol_1, auth_protocol_2

zkp = ZKP._instance # defined globally across all modules at runtime
class Sum(ActiveProtocol):
    def __init__(self, local_idx, local_port, parties, enable_stats):
        super().__init__(local_idx, local_port, parties, enable_stats, field_type=field)

    async def setup(self):
        """Compiler setup"""
        N = self.parties

        # Step 0(a): Compiling the ZK-statements
        functions = [auth_protocol_0, auth_protocol_1, auth_protocol_2]
        includes = [commit, N, protocol_0, protocol_1, protocol_2]
        self.compile_zkps(functions, includes, globals(), locals())

        # Step 0(b): Performing a trusted up
        if zkp.backend in ["groth16"]:
            await self.trusted_setup(functions)

    async def engage(self, secret):
        """Protocol engagement"""
        i = self.local_idx
        N = self.parties

        # Step 1: Obtain randomness
        coins = self.secure_coin_flipping(N - 1)

        # Step 2(b)i: Compute protocol_0 functionality
        outputs_0 = protocol_0(secret, coins, i, field(1))

        # Step 2(b)ii: Compute commitments and blinding factors to outputs of protocol_0
        my_blindings = self.coin_flipping(2 * N)
        my_commitments = auth_protocol_0(secret, coins, my_blindings, i, field(1))
        commitments = await self.communicate(my_commitments, "broadcast", "protocol_0", "int_list", flatten=True,
              unflatten=8)

        # Step 2(b)iii: Distribute outputs and blinding factors
        inputs_1 = await self.communicate(outputs_0, "distribute", "engagement", "field")
        blindings_1 = await self.communicate(my_blindings[0::2], "distribute", "blindings_1", "field")
        blindings_2 = await self.communicate(my_blindings[1::2], "distribute", "blindings_2", "field")
        blindings = [item for pair in zip(blindings_1, blindings_2) for item in pair]

        # Step 2(b)iv: Authenticate protocol_0
        args_prove = (secret, coins, my_blindings, i, field(1))
        args_verify = (commitments, [[None, None, None, j, field(1)] for j in range(N)])
        await self.authenticate(auth_protocol_0, args_prove, args_verify)

        return inputs_1, blindings, commitments

    async def emulate(self, inputs, blindings, all_commitments):
        """Protocol emulation"""
        i = self.local_idx
        N = self.parties

        # re-organize commitments according to received secret shares
        incoming_commitments = [[comm[j] for comm in all_commitments] for j in range(N)]

        # Step 3(a): Compute and broadcast protocol_1 functionality
        output_1 = protocol_1(inputs, field(1))
        inputs_2 = await self.communicate(output_1, "broadcast", "protocol_1", "field")

        # Step 3(b): Authenticate protocol_1
        args_prove = (inputs, blindings, incoming_commitments[i], field(1))
        args_verify = (inputs_2, [[None, None, incoming_commitments[j], field(1)] for j in range(N)])
        await self.authenticate(auth_protocol_1, args_prove, args_verify)

        # Step 3(a): Compute and broadcast protocol_2 functionality
        output_2 = protocol_2(inputs_2, field(1))
        final_output = await self.communicate(output_2, "broadcast", "protocol_2", "field")

        # Step 3(b): Authenticate protocol_2
        args_prove = (inputs_2, field(1))
        args_verify = (final_output, [[inputs_2, field(1)] for j in range(N)])
        await self.authenticate(auth_protocol_2, args_prove, args_verify)

        # Termination
        if all(output == output_2 for output in final_output):
            print(output_2)
        else:
            raise ValueError("Invalid output")
```

B

ZKP Compiler: Design and Usage

To clarify that ZKPyC is a subset language, we will first identify the omitted nodes and provide an explanation for each omission. Next, we will describe the additions made and discuss their implications for compatibility with Python. Lastly, we will outline how certain nodes have been restricted by the compiler infrastructure while maintaining compatibility with Python.

B.1 Language Design Choices

B.1.1 Syntax Omissions

Firstly, we observe that the **Interactive**, **Expression**, and **FunctionType** nodes have been removed from the *mod* type. These constructs enable Python to function as an interactive interpreter, a feature not present in ZKPyC since its programs are intended to be compiled and thus must be parsed as a **Module**.

We omit the following constructs with the following reasons:

stmt::AsyncFunctionDef, stmt::AsyncFor, stmt::AsyncWith

Currently, it is unclear how coroutines could be compiled into arithmetic circuits, as circuit execution cannot be partially evaluated. A practical workaround is to identify which parts of an asynchronous statement can be executed synchronously and compile those instead.

stmt::Delete

Because arithmetic circuits are static structures, dynamic allocation is not possible, and consequently, objects cannot be deallocated.

stmt::While

While it may be possible to implement while loops if the conditional expression is statically bound to a variable defined in the current scope, this feature is not necessary at this time.

stmt::If

It may seem counterintuitive to exclude if statements, but there are two main reasons for this decision. Firstly, for if-else branching to occur in an arithmetic circuit, both branches must be executed as part of a single arithmetic expression. For example, given a condition P with result a and its complement $\overline{P}$ with result b, an if-else statement would be represented by the expression: $P \cdot a + \overline{P} \cdot b$. Secondly, this requires that both results be of the same type.

While it is possible to enforce proper if-else behavior, we currently achieve this more easily using the **IfExp** expression.

stmt::With

This construct necessitates the concept of a *context manager*, which requires an object to have associated start and end procedures. While it is possible to implement this for behavioral classes, currently only data classes are supported.

stmt::Match

Similar to if-else statements, match statements could be implemented but would require a uniform result type and the evaluation of all possible branches.

stmt::Raise, stmt::Try

Exceptions and exception handling could be implemented under the guise of exceptional control flow. However, true exception handling is not feasible for arithmetic circuits, so it is currently excluded.

stmt::Global, stmt::Nonlocal

Since an arithmetic circuit represents a function, the only way to access a variable outside of that function is by returning it. Consequently, variables cannot be made globally accessible outside the scope of a function.

stmt::Expr

Expression statements are not consumed by other statements, so they do not contribute to the circuit.

stmt::Pass, stmt::Break, stmt::Continue

Pass statements currently serve little purpose other than enhancing code readability. Break and pass statements are technically of the same type and could be used in **IfExpr** expressions. Additionally, they could be implemented as a form of exceptional control flow, but they are not considered essential at this time.

expr::NamedExpr

This is the so called *walrus operator* in Python, and allows expressions to be both evaluated and assigned to a new variable. The issue with this construct is that it does not admit a type signature, so the assignment would require type inference. This is achievable, but currently not a desirable feature.

expr::Lambda

This construct allows for unnamed functions to be instantiated and evaluated in place. While this is generally a desirable feature, we do not want functions to be treated as objects that can be returned. This may become feasible with the introduction of a static pointer data type, but it is not essential at the moment.

expr::Dict, expr::DictComp, expr::Set, expr::SetComp

Dictionaries are not supported due to their dynamic size, and are best replaced by using data classes as their static counterparts. While sets could be statically sized, they require extensive additional processing, resulting in expensive circuits. In most cases, arrays serve as reasonable alternatives.

expr::GeneratorExp

This construct is an abstraction of a *comprehension* over some iterable. While it should be possible to support this for statically sized iterable structures, it is not a crucial feature to have as the only currently supported iterable structure right now is an array.

expr::Await, expr::Yield, expr::YieldFrom

As discussed earlier, asynchronous functions are not currently supported.

expr::FormattedValue, expr::JoinedStr

String types currently serve no purpose in arithmetic circuits.

expr::Tuple

This may seem surprising, but tuples are currently not supported since they can be replaced by their mutable counter part: arrays. They may, however, be useful to implement in the future to support things like destructuring assignments and multiple returns.

expr_context::Delete

Currently serves no purpose.

operator::FloorDiv

The compiler infrastructure only supports division for field types.

cmpop::In, cmpop::NotIn

They are not currently essential features and can be implemented using a for loop.

B.1.2 Syntax Additions

The only necessary additions are the *access* and *type* types for reasons described earlier. They can be interpreted using the Python AST as a type annotation through the **Tuple** and **Subscript** constructs respectively.

B.1.3 Syntax Modifications

The modifications are not directly obvious, so we list them for each type.

In all nodes, we removed the fields: *type_ignore** type_ignores, *string*? type_comment and *type_param** type_params, since they serve no purpose in ZKPyC.

stmt::FunctionDef

Decorators serve no purpose for the time being so the field for it has been omitted, and a return type is always expected, so the product specifier was removed.

stmt::ClassDef

As before, decorators serve no purpose, but can still be parsed for the sake of declaring it a dataclass in Python. Bases and keywords have also been omitted since class inheritance and meta-class specification is not supported.

stmt::Return

The option specifier was removed, since a return statement is always expected.

stmt::Assign

The product specifier was removed, since multi-assignment is currently not supported.

stmt::AnnAssign

The *expr* annotation field was replaced by the *type* signature field to only allow for supported ZKPyC types.

stmt::ListComp

The product specifier was removed, since list comprehension is currently only limited to initializing single value arrays.

Lastly, we modified the remaining types as follows:

- *comprehension*: The *expr* iter, *expr** ifs, *int* is_async were ommited. The reason is that comprehensions are only used in array initialization of single values, so iterators currently serve no purpose. Consequently, **IfExpr** expressions also serve no purpose, and asynchronous statements are not supported.
- *arguments*: This type has been significantly scoped down as the *arg** args, *arg*? vararg, *arg** kwonlyargs, *expr** kw_defaults, *arg*? kwarg and *expr** defaults fields have been omitted. For now it is only crucial to support positional arguments, and the other attributes could be incorporated in the future.
- *arg*: The *expr*? annotation field was replaced by the *type* signature field, as each argument variable must be labeled with one of the supported ZKPyC types.

B.1.4 Semantic Considerations

This concludes the syntactic definition of ZKPyC, which also partly describes some semantic restrictions. While most of the language semantics are similar to Python, we should note the exceptions and state the following considerations:

Compiler pre-processing

Just like regular Python, comments are supported. Additionally, the compiler will ignore a line if it is followed by the comment: `# ZK_IGNORE` (case-insensitive and allows for any number of whitespaces). We consider this to be the *ignore directive*, which is can also be considered a *macro*.

Module-level declarations

A module is comprised of the following statements:

- Possible import statements, interpreted from **Import** and **ImportFrom** statements. These *must* be stated at the module-level.
- Constants definitions, which are interpreted from **AnnAssign** statements. These are accessible in the scopes of all class and function definitions.
- Function and class definitions, which are interpreted from **FunctionDef** and **ClassDef** statements respectively. It is currently *not* possible to define these inside of the scope of a function.

Function and class definitions

All functions *must* have typed arguments and include a return type. The return keyword *must* also be explicitly stated at the end of the function. Classes are *strictly* data classes and may only contain typed field declarations. For example:

```
class Name():
    integer: int
    array: Array[int, 3]
    boolean: bool
```

In Python, this could be declared using the `@dataclass` decorator. Moreover, recursive class definitions are not supported.

Function calls and class instantiation

A function call *must* consist of positional arguments, e.g. if a function is defined as `def func(x: int, y: field) -> field`, then `func(5, field(10))` is a valid function call, but `func(x=5, y=field(10))` is not. Moreover, recursive function calls are currently not supported. Class instantiation, on the other hand, only admits keyword arguments. For example, given the previously defined class, `Name(integer=5, array=[1,2,3], boolean=True)` is a valid class instantiation, but `Name(5, [1,2,3], True)` is not.

Literals

Just like in regular Python, integers can be represented as decimal, hexadecimal and binary numbers. Boolean and array literals are also constructed the same way. Unlike floats, field literals are not native to Python and cannot be expressed by a suffix. Instead, it can be expressed as `field(value)` where `value` is an integer literal.

Array initialization

There are two ways to initialize arrays.

1. The first way is by declaring an array literal, which is a Python list that consists of constants that are of the same type. For example:
   `arr: Array[field, 3] = [field(0), field(1), field(2)]`.
2. The other way is to use list comprehension for a fixed value literal. For example:
   `arr: Array[field, 3] = [field(0) for _ in range(3)]`.
   It is important to note that the only accepted iterable object for now is `range`, and iterator variables are not used.

Type casting

Type casting is possible from and to any of the primitive types: `int`, `field` and `bool`. First, consider the type ranks to be as follows: `bool < int < field`, where the data of a lower ranked type can be represented by a higher ranked type. This is similar to *integer promotion* in C, but here it only happens when carried out explicitly. The converse is also possible, but requires the data to be *truncated*. From `field` to `int`, this is achieved through a circuit implementation of modular arithmetic, and from `int` to `bool`, this can be achieved through a null equality check. All of this is possible through IR operations.

Array composition

Since array concatenation has not yet been defined, one way to compose multiple arrays would be to initialize a new array that contains the contents of the other arrays. This can be done through unpacking, using the asterisk operator. For example, given `int` arrays `arr1`, `arr2` and `arr3` each of size 3, we can compose them by doing `arr: Array[int, 9] = [*arr1, *arr2, *arr3]`.

Compiler entry function

For the module to be compilable, there *must* be an entry function from which the arithmetic circuit is built. By default, this would be `main`, but this can be modified through a compiler argument. Moreover, the `main` function is the only function where the access specifier *must* be used for all arguments.

Embedded module

ZKPyC, just like ZoKrates, has an embedded module called `EMBED` that consists of functions that cannot be programmed directly in the ZKPyC language. The currently implemented functions are:

- `int_to_bits(x: int) -> Array[bool, 32]`
- `int_from_bits(x: Array[bool, 32]) -> int`
- `int_to_field(x: int) -> field`
- `unpack(x: field, N: int) -> Array[bool, N]`
- `bit_array_le(x: Array[bool, N], y: Array[bool, N]) -> bool`
- `get_field_size() -> int`

Their purposes are self-evident from the names and type signatures, but a couple of things should be noted. Firstly, the function `int_to_field` is equivalent to the type-cast `field(x)` for some integer `x`. The function `unpack` maps a field element to a variable sized bit-array. Originally, this was handled using generics in ZoKrates, but in this case we specify the array size as part of the input. The function `bit_array_le` compares two equally sized bit-arrays with the `<=` operator, as if they were integers. The take-away is that this allows for comparisons of integers that are smaller or larger than 32 bits, and could be extended to leverage support for Python's dynamically sized integers.

B.2 Example Programs

Finally, we look at some examples of modules that can be compiled, and simultaneously can be interpreted and run in a Python runtime environment. We follow the syntax and semantic

considerations described in Section 5.2.2. Moreover, we also describe the requirements that are specifically imposed by the CirC compiler infrastructure for a given program.

We start with a simple example of integer addition in Figure B.1:

```
from zk_types.types import Private # zk_ignore

def main(x: Private[int]) -> int:
    return x + x + x
```

Figure B.1: Integer addition in ZKPyC language.

Note that we use the compiler ignore directive to allow for ZKPyC to ignore the import statement, as the types are built into the compiler. This allows for the Python runtime environment to import the necessary types, which are not known a priori. The compiler then outputs a representation of the R1CS constraints, together with a computation model that is used to evaluate the arithmetic circuit and compute a witness.

For generating the correct witness and instance values, the compiler expects both the prover and verifier to provide their respective private and public information in a dialect of lisp: `(let ((var_1 literal) ... (var_n literal)) t )`, where `let` is a binding for the value map with an associated value `t`. This is then interpreted as a CirC-IR term, and used directly in the circuit evaluation. Since this term is used for assigning input/output wires, the associated term value can be set to anything, since it is not used in the computation. In the case of a verifier, a (sequence of) `return` assignment(s) must be provided as part of the public information. For the prover that is not necessary since the function is evaluated from its inputs. For the integer addition function, the prover and verifier could prepare their inputs as follows: These inputs are then parsed and assigned to the input (and output) wires of

```
(let (
    (x #x00000004)
)
    false
)
```

```
(let (
    (return #x0000000c)
)
    false
)
```

Figure B.2: The respective prover and verifier inputs described as a CirC-IR value map, for integer addition in the ZKPyC language.

the computation model, and then used to produce the correctly sorted witness and instance variable assignments for both the prover and verifier respectively.

We now consider the simple example of field element addition, which cannot be carried out directly due to a shortcoming of the compiler. Unlike integer addition, it is currently not possible to carry out addition of field elements without involving a multiplication in the expression. A workaround, as suggested, involves transforming the problem of addition to that of multiplication, as shown in Figure B.3. By introducing a public variable `_one`, we can

```
from zk_types.types import Private, Public, field # zk_ignore

def main(x: Private[field], y: Private[field], _one: Public[field]) -> field:
    out: field = (x + y) * _one
    return out
```

Figure B.3: Field addition in ZKPyC language.

ensure that verifier is aware that the correct multiplicative factor is chosen, which in this case must always be 1. Since here we include a public variable, both the verifier and the prover must provide the same value as part of their input. Moreover, we also must specify the field modulus via the `set_default_modulus` binding, as is illustrated in Figure B.4.

```
(set_default_modulus
    52435875175126190479447 7...
    (let (
        (x #f3)
        (y #f4)
        (_one #f1)
    )
        false
    )
)
```

```
(set_default_modulus
    52435875175126190479447 7...
    (let (
        (_one #f1)
        (return #f7)
    )
        false
    )
)
```

Figure B.4: The respective prover and verifier inputs for field addition in the ZKPyC language.

To highlight the expressive power of ZKPyC, we show two more examples. The first example is that of matrix multiplication:

```
from zk_types.types import Private, Array, field # zk_ignore

def main(
    A: Private[Array[Array[field, 2], 2]],
    B: Private[Array[Array[field, 2], 2]]
) -> Array[Array[field, 2], 2]:
    AB: Array[Array[field, 2], 2] = [
        [field(0) for _ in range(2)] for _ in range(2)
    ]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                AB[i][j] = AB[i][j] + A[i][k] * B[k][j]
    return AB
```

Figure B.5: Matrix multiplication of 2×2 matrices in the ZKPyC language.

If, for example, a prover wanted to prove the computation:

$$\begin{bmatrix} 23 & 52 \\ 99 & 123 \end{bmatrix} \times \begin{bmatrix} 12 & 1 \\ 100 & 42 \end{bmatrix} = \begin{bmatrix} 5476 & 2207 \\ 13488 & 5265 \end{bmatrix}$$

over a large finite field, then the prover and verifier must prepare their inputs, as illustrated in Figure B.6. In the other example, we show how to utilize data classes and functions from a separate module, as well as typecasting and the declaration of constants. Consider a function that applies a horizontal shear transformation, defined by the matrix:

$$S = \begin{bmatrix} 1 & 10 \\ 0 & 1 \end{bmatrix},$$

to a point `Pt` with (x, y) coordinates as integers. First we could define the data class inside of a module named `point.py`, as illustred in Figure B.7.

Using the `Pt` class and previously defined matrix multiplication function from a module named `mm.py`, we define the point transformation function as follows:

```
(set_default_modulus
    52435875175126190479447 7...
    (let (
        (A.0.0 #f23)
        (A.0.1 #f52)
        (A.1.0 #f99)
        (A.1.1 #f123)
        (B.0.0 #f12)
        (B.0.1 #f1)
        (B.1.0 #f100)
        (B.1.1 #f42)
    )
        false
    )
)
```

```
(set_default_modulus
    52435875175126190479447 7...
    (let (
        (return.0.0 #f5476)
        (return.0.1 #f2207)
        (return.1.0 #f13488)
        (return.1.1 #f5265)
    )
        false
    )
)
```

Figure B.6: The respective prover and verifier inputs for matrix multiplication in the ZKPyC language.

```
from dataclasses import dataclass #zk_ignore

@dataclass
class Pt:
    x: int
    y: int
```

Figure B.7: Definition of a point data class in the ZKPyC language.

```
from zk_types.types import Private, Public, Array, field # zk_ignore
from mm import main as multiply
from point import Pt

S: Array[Array[field, 2], 2] = [
    [field(1), field(10)],
    [field(0), field(1)]
]

def main(pt: Private[Pt]) -> Pt:
    x: field = field(pt.x)
    y: field = field(pt.y)
    P: Array[Array[field, 2], 2] = [[x, field(0)], [y, field(0)]]
    result: Array[Array[field, 2], 2] = multiply(S, P)
    new_x: int = int(result[0][0])
    new_y: int = int(result[1][0])
    return Pt(x=new_x, y=new_y)
```

Figure B.8: Shear transformation of a point in the ZKPyC language.

If a prover and verifier wanted to prove the transformation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \mapsto \begin{bmatrix} 64 \\ 6 \end{bmatrix},$$

then they both must prepare their inputs as illustrated in Figure B.9.

```
(let (
    (pt.x #x00000004)
    (pt.y #x00000006)
)
    false
)
```

```
(let (
    (return.x #x00000040)
    (return.y #x00000006)
)
    false
)
```

Figure B.9: The respective prover and verifier inputs for the shear transformation of a point in the ZKPyC language.

C

Performance Analysis: Supplement

C.1 Proof of the Circuit Complexity Lemma

Lemma C.1.1 (Circuit Size Complexity)**.** Let $f : \mathbb{Z}_p^q \to \mathbb{Z}_p^r$ be some function and C_f be its corresponding arithmetic circuit with q input wires. The total number of gates in C_f grows in the order of $\mathsf{poly}(q)$.

Proof. The proof will have two parts: in part 1 we show that the lemma holds for any arithmetic circuit corresponding to a polynomial function $f : \mathbb{Z}_p^q \to \mathbb{Z}_p$, and in part 2 we extend the result for any multi-dimensional polynomial function. We recall that $\mathsf{poly}(q) := q^{O(1)}$. Moreover, we denote by $|\cdot| : \mathbb{Z}_p[x_1, \ldots, x_q] \to \mathbb{Z}^{\geq 0}$ the measure of *circuit size*, which counts the total number of addition and multiplication operations for a given polynomial.

Part 1: Suppose that C_f is a non-uniform arithmetic circuit for some q-variate s-degree polynomial function, which is expressed as

$$f(x_1, \ldots, x_q) = \sum_{i_1+i_2+\ldots+i_q \leq s} a_{i_1 i_2 \ldots i_q} x_1^{i_1} x_2^{i_2} \cdots x_q^{i_q} \text{ where } i_1, \ldots, i_q \in \mathbb{Z}_p^{\geq 0}.$$

We will provide a proof by induction on s to show that $|C_f| = \mathsf{poly}(q)$ for any q. It is easy to see for $s = 1$ that there are at most $q - 1$ additions and q multiplications, i.e., $|C_f| \leq 2q - 1$, which is clearly $\mathsf{poly}(q)$. If we set $s = k$, then there are $\binom{q+k}{q} = \binom{q+k}{k}$ ways to pick $i_1, \ldots, i_q$ such that $i_1 + i_2 + \ldots + i_q \leq k$, therefore the circuit size has an upper bound of

$$|C_f| \leq \binom{q+k}{k} qk.$$

Let us assume that above proposition holds for $s = k$ and call this the induction hypothesis. We express the induction step for $s = k + 1$ as follows:

$$\begin{aligned} |C_f| &\leq \binom{q+k+1}{k+1} q(k+1) \\ &= \left(\binom{q+k}{k} + \binom{q+k}{k+1} \right) q(k+1) \\ &= \underbrace{\binom{q+k}{k} q}_{(1)} + \underbrace{\binom{q+k}{k+1} q}_{(2)} + \underbrace{\binom{q+k}{k} qk}_{(3)} + \underbrace{\binom{q+k}{k+1} qk}_{(4)}. \end{aligned}$$

To prove the induction step, we make use of the fact that if $|C_f| \leq (i)$ implies $|C_f| = \mathsf{poly}(q)$ for each case i, then $|C_f| \leq \sum_{i=1}^{4} (i)$ implies $|C_f| = \mathsf{poly}(q)$. Since $(1) \leq (3)$ and $(2) \leq (4)$, it

suffices to show that the antecedent holds for cases 3 and 4. Since case 3 is the induction hypothesis, we only need show that case 4 holds. Since k is constant, there exists a constant c such that $q + k \leq q^c$, and we obtain the upper bound

$$|C_f| \leq \binom{q+k}{k+1} qk \leq \frac{(q+k)^{k+1}}{(k+1)!} qk = \frac{(q+k)^{k+1}}{(k+1)(k-1)!} q \leq q(q+k)^{k+1} \leq q^{c(k+1)+1}.$$

Since $c(k+1)+1 = O(1)$, it follows that $|C_f| = \mathsf{poly}(q)$ and the proof by induction is done.

Part 2: By extension, consider a multi-dimensional polynomial function f with $\mathrm{rank}(f) = r$ and circuit C_f as a union of subcircuits C_{f_i}, where $f_i : \mathbb{Z}_p^q \to \mathbb{Z}_p$. Assuming that the intersection of subcircuits is non-empty, we use the inclusion-exclusion principle to count the total circuit size, i.e.,

$$|C_f| = \left| \bigcup_{i=1}^{r} C_{f_i} \right| = \sum_{i=1}^{r} |C_{f_i}| + \sum_{k=2}^{r} (-1)^{k+1} \left(\sum_{1 \leq i_1 < \cdots < i_k \leq r} \left| C_{f_{i_1}} \cap \cdots \cap C_{f_{i_k}} \right| \right).$$

Since $|\cdot|$ is a measure, it follows from the property of monotonicity that $|C_{f_1} \cap \cdots \cap C_{f_r}| \leq |C_{f_i}|$ for all $i = 1, \ldots, r$. Since $|C_{f_i}| = \mathsf{poly}(q)$, it follows that $|C_f| = \mathsf{poly}(q)$.

□

C.2 Zero-Knowledge Proof Constraints Measurements

Table C.1: Comparison of the number of R1CS constraints for protocol evaluation ZK-statements (Definition 4.2.2) vs protocol authentication ZK-statements (Definition 4.2.3), for the multi-party sum protocol $\Pi_{\mathrm{SUM}} = \Pi^2_{\mathrm{SUM}} \diamond \Pi^1_{\mathrm{SUM}} \diamond \Pi^0_{\mathrm{SUM}}$ (Figure 6.1)

Parties	Subprotocol Evaluation				Subprotocol Authentication			
	0	1	2	total	0	1	2	total
BLS12-381 (Groth16)								
3	46	1	1	**48**	20455	20410	1	**40866**
4	91	1	1	**93**	27303	27213	1	**54517**
5	144	1	1	**146**	34157	34016	1	**68174**
6	202	1	1	**204**	41017	40819	1	**81837**
7	266	1	1	**268**	47887	47622	1	**95510**
8	336	1	1	**338**	54755	54425	1	**109181**
9	410	1	1	**412**	61633	61228	1	**122862**
10	494	1	1	**496**	68523	68031	1	**136555**
11	579	1	1	**581**	75414	74834	1	**150249**
12	675	1	1	**677**	82309	81637	1	**163947**
13	774	1	1	**776**	89210	88440	1	**177651**
14	882	1	1	**884**	96122	95243	1	**191366**
15	989	1	1	**991**	103039	102046	1	**205086**
16	1112	1	1	**1114**	109958	108849	1	**218808**
17	1229	1	1	**1231**	116885	115652	1	**232538**
18	1359	1	1	**1361**	123813	122455	1	**246269**
19	1499	1	1	**1501**	130752	129258	1	**260011**
20	1643	1	1	**1645**	137704	136061	1	**273766**
21	1785	1	1	**1787**	144648	142864	1	**287513**
22	1946	1	1	**1948**	151607	149667	1	**301275**
23	2103	1	1	**2105**	158572	156470	1	**315043**
Ristretto255 (Bulletproofs)								
3	46	1	1	**48**	20356	20311	1	**40668**
4	91	1	1	**93**	27171	27081	1	**54253**
5	142	1	1	**144**	33994	33851	1	**67846**
6	199	1	1	**201**	40822	40621	1	**81444**
7	266	1	1	**268**	47653	47391	1	**95045**
8	331	1	1	**333**	54491	54161	1	**108653**
9	410	1	1	**412**	61340	60931	1	**122272**
10	494	1	1	**496**	68187	67701	1	**135889**
11	578	1	1	**580**	75051	74471	1	**149523**
12	672	1	1	**674**	81915	81241	1	**163157**
13	778	1	1	**780**	88780	88011	1	**176792**
14	883	1	1	**885**	95660	94781	1	**190442**
15	994	1	1	**996**	102539	101551	1	**204091**
16	1109	1	1	**1111**	109431	108321	1	**217753**
17	1231	1	1	**1233**	116321	115091	1	**231413**
18	1367	1	1	**1369**	123223	121861	1	**245085**
19	1497	1	1	**1499**	130130	128631	1	**258762**
20	1639	1	1	**1641**	137043	135401	1	**272445**
21	1787	1	1	**1789**	143961	142171	1	**286133**
22	1947	1	1	**1949**	150885	148941	1	**299827**
23	2106	1	1	**2108**	157816	155711	1	**313528**