Developing a Multi-Party MPC Compiler with Covert Security and Public Verifiability

V.A. Dunning
M.Sc. Thesis
August 2021

Supervisors:
dr. ir. M.H. Everts (UT)
prof. dr. ir. R.M. van Rijswijk-Deij (UT)
prof. dr. A. Peter (UT)
dr. ir. P.N. Langenkamp (TNO)
T. Attema, M.Sc. (TNO)

University of Twente:
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Services & Cybersecurity Group (SCS)
TNO:
Cyber Security & Robustness (CSR)

Abstract

Multi-party computation (MPC) is a cryptographic tool that enables a number of parties to perform computations with their inputs while keeping their inputs private and ensuring correctness of the outcome. For many years, these MPC protocols were not yet efficient enough for all relevant scenarios. However, a large line of research and the increasing power of modern computers made the use of these protocols in practice feasible. Traditionally, two types of adversaries have been distinguished: *passive* adversaries who follow the protocol honestly and *active* adversaries who can arbitrarily deviate from the protocol. In general, passively secure protocols are fast but not very secure while actively secure protocols are very secure, but typically slow. As a compromise between these two notions, Aumann and Lindell [AL07] presented the notion of *covert* adversaries in 2007. Covert adversaries cheat like an active adversary but only get caught with a certain probability, called the *deterrence rate*. The idea is that this chance of being caught should be enough to deter most cheaters in practice. Later in 2012, the additional notion of *public verifiability* was introduced by Asharov and Orlandi [AO12]. In this model, an extra mechanism is added that allows the parties to generate a *certificate* that proofs the cheating to *anyone*, including third parties. This notion should discourage cheating even further. As the interest in using MPC for actual use-cases increased, another line of research to arise is that of MPC *compilers*, which provide a generic blueprint to transform a protocol with passive security into a protocol with stronger security.

In this work, we investigate the mechanisms for building MPC protocols and compilers with various security levels. After that, we present a new design for transforming protocols with passive security into protocols with covert security and public verifiability. Our compiler is based on a construction called *Publicly verifiable secret sharing*. This compiler treats the passively secure protocol in a *black-box* manner, meaning it will be applicable to any current or future protocol. Furthermore, our compiler works efficiently for MPC protocols with an arbitrary number of parties. The only known works that also achieve these properties have been presented in 2021 by Faust et al. [Fau+21] and concurrently by Scholl et al. [SSS21].

Compared to the works presented in [Fau+21; SSS21], our design is able to obtain the same (optimal) deterrence rates with much simpler building blocks by assuming the existence of an honest majority. With this assumption, we reduce the complexity of the compiler by multiple orders of magnitude compared to [Fau+21]. Furthermore, we present a proof-of-concept implementation of our design in the MPyC framework [Sch18]. To the best of our knowledge, this is the first time such a compiler has been applied in practice. By benchmarking this implementation, we further demonstrate the power of covert security. Compared to an actively secure implementation of the framework, our implementation reduces the overhead over a passively secure version by a factor 3 up to a factor 10 depending on the number of parties and the chosen deterrence rate.

Contents

Chapter 1

Introduction

1.1 Multi-Party Computation

Suppose multiple parties hold sensitive data they want to combine to extract a shared result. For example, multiple banks who together want to combat fraud more effectively by combining their respective transaction data. Due to regulations, they cannot simply share the data of their customers with each other but need to keep this sensitive customer information confidential. Hypothetically, this could be solved with an incorruptible, trusted third party where everyone sends their confidential inputs to. The trusted third party computes the desired functionality and gives everyone their outputs, which are guaranteed to be correct. However, in the real world such a trusted third party might be very costly or even impossible to find. Secure Multiparty Computation (MPC) is a cryptographic solution for this problem.

In MPC, two or more parties want to compute a joint function on their private inputs while revealing nothing but the output of the function. This means that privacy of the inputs as well as correctness of the output need to be guaranteed, even in the presence of an attacker or *adversary*. Essentially, MPC is an umbrella term for a set of techniques which 'emulate' the aforementioned trusted third party by means of a protocol the parties execute among themselves.

The idea of MPC has been around since the late 80s when Yao published his, now famous, *Millionaires' problem* [Yao82]. However for a long period, these protocols remained mainly of theoretical interest due to their complexity and impracticality. As a result of the increasing attention to the design of practical protocols and the increasing computational power of modern machines, practical implementations of MPC have started to see the light. Examples of use-cases which have been solved with MPC include auctions, anti money laundering, combining healthcare data and many more.

1.2 Security Models

To reason about the security of these protocols, the capabilities of possible adversaries need to be established. Traditionally, a distinction has been made between *passive* adversaries and *active* adversaries. A passive adversary is curious what he can find out about the other users, potentially compromising the privacy but does follow the protocol honestly. On the other hand, active adversaries try to compromise the privacy as well as the corectness of the outcome of the protocol by arbitrarily deviating from the protocol or even stop sending messages altogether. While safeguarding against active adversaries provides a very strong degree of security, this comes at a significant cost in terms of complexity of the MPC protocol. On the other hand, passively secure protocols give rise to efficient protocols but have weak security guarantees. As the focus in recent years shifted from theoretical feasibility of MPC protocols towards actual practical implementations, there came a desire for more practical adversary models. In 2007, Aumann and Lindell [AL07] proposed a new adversary model called a *covert adversary*.

The idea for this type of adversary is to compromise some security compared to active security for more efficiency. Instead of guaranteeing the detection of an active attack, now it suffices to catch this cheating behaviour only with a certain probability, called the *deterrence rate*. This is thought of to be a more natural security notion as the chance of being caught cheating and the additional loss of reputation are often enough to dissuade a possible cheater. This definition of security did result in more practical protocols, where the trade-off between security and complexity can often be tweaked by means of a parameter which determines the probability of catching the adversary.

In 2012, Asharov and Orlandi [AO12] observed that despite this natural notion, slightly stronger guarantees were actually needed in practice. Consider for example a large bank that performs MPC with its clients. If one client catches the bank cheating, the bank will lose its reputation with this client but if there is no way for the client to proof to other parties that cheating has happened, the reputation damage is minimised to only this client. As a solution to this, they introduced the extended notion of covert security with *public verifiability*. In this security model, the parties are equipped with an additional mechanism to produce a *certificate* which undeniably proves that another party has attempted to cheat in the protocol execution. This is a much stronger security guarantee in comparison to the plain covert security model.

Even though this notion looks promising for a wider use of MPC in practice, relatively little research has been done in this area. The only concrete protocols in this security model have been presented in [AO12; KM15; Hon+19].

1.3 Compilers

In the search for practical MPC protocols, another line of interest was the wish for *general* approaches for realising secure protocols. These approaches aim to provide a generic way to obtain efficient, strongly secure protocols.

This is where so-called *compilers* come in. These provide a generic approach or blueprint to transform protocols with passive security into protocols with stronger security guarantees. For example, a compiler might take a passively secure protocol as input and output an actively secure protocol that calculates the same function. Compilers are useful because if in the future a new highly efficient approach for, e.g., evaluating neural networks were to be found in the passive security model, it can automatically be compiled to obtain a covertly or even actively secure version. Furthermore, these compilers allow programmers with less expertise to implement MPC for their problems without needing to know all the non-trivial details of designing strongly secure protocols.

In this work, the techniques to obtain these generic transformations will be investigated. In particular the *player virtualisation* paradigm, which has proven to be effective and applicable to a wide range of concrete protocols. This paradigm was introduced by Hirt and Maurer in 2000 [HM00] and includes various strategies based on *simulating* a number of virtual parties who will then execute the actual protocol on behalf of the real parties. The behaviour of these virtual parties is then verified resulting in various security guarantees including covert security, the extended notion of public veriability as well as active security.

The first compilers using this paradigm were presented by Ishai, Prabhakaran and Sahai in the form of the IPS Compiler [LPS08; IPS08] and later improved in [LOP11]. These compilers transform passively secure protocols into protocols with active security. In recent years, such compilers for actively secure MPC based on this paradigm were introduced in [DOS18; Eer+20] which was optimised to be used in the three-party case and [Abs+21] for an arbitrary number of parties.

For compilers with covert security and public verifiability, little is known. The first compiler for this was introduced in 2020 in [DOS20] in the two-party setting. Here a construction is presented which makes no assumptions of the passively secure protocol, meaning it can be applied to any passively secure protocol. Furthermore, this work presented ideas on how to efficiently instantiate these compilers in practice. However, the obtained deterrence rates are fairly low, especially if this were to be extended to multi-party protocols. This is due to the fact that each party picks their watchlist individually, likely reducing the size of the set of executions that can be checked.

In 2021, improvements for arbitrary multi-party protocols were presented in [Fau+21; SSS21] by constructing a shared set of executions that all the parties check. With an overhead of roughly 2 times the passively secure protocol, they already obtain a deterrence rate of $\frac{1}{2}$, with an overhead of 3, they reach a deterrence rate of $\frac{2}{3}$, etc. While these are very promising results, some hard assumptions on the network latency in which these protocols run are made which make these constructions complex and hard to realise in practice.

In this work we will use these works and design a new construction for compilers for covert security with public verifiability in the multi-party setting. Compared to the watchlist approach in [DOS20], we obtain a much more efficient approach to reach higher deterrence rates for a larger number of parties. compared to the works of [Fau+21; SSS21], we reach the same optimal deterrence rates with less complex building blocks with the assumption of an honest majority.

1.4 Contributions

The goal of this thesis is to shrink the gap between the usability and performance of concrete MPC protocols and compilers. To this end, we aim to provide insight and present improvements in the design and implementation of MPC protocols and MPC compilers. Our first contribution is therefore a survey of the existing literature regarding the choices on security levels, guarantees and other assumptions. Furthermore, we survey the influence of these choices on the design and implementation of concrete MPC protocols and we survey the existing work on the design of MPC compilers with various security guarantees.

Our second and main contribution is a new design for two compilers for transforming passively secure protocols into protocols with covert security and public verifiability. These compilers treat the passively secure protocol in a *black-box* manner, meaning they will work for any current or future protocol. Our first compiler can be applied to input-independent protocols while the second compiler also works for protocols that do receive private inputs. These compilers build on the ideas introduced in [DOS20; Fau+21; SSS21]. Compared to [DOS20], our compilers yield much higher deterrence rates in the multi-party setting with less overhead. Compared to [Fau+21; SSS21], our compilers reach the same deterrence rates but by assuming an honest majority, we can use simpler building-blocks. These two works base their security on a construction which encrypts something in a way that costs an (approximately) fixed amount of time to unlock. This time is picked such that it is *just* long enough such that an adversary can not unlock it within a number of communication rounds but short enough to not introduce too much overhead when the honest parties or outsiders have to unlock it. Since the honest parties always have to solve the time-lock puzzle if a cheating attempt takes place, this time is set as low as possible. This predicament makes it hard to realise in practice where predefined communication rounds are very hard to realise and the asynchronous nature of the networks on which these protocols run make it hard to be secure in practice. Furthermore, for opening the passively secure executions, they require general-purpose MPC protocols with active security to implement an ideal functionality, which is unrealistic in a setting where we want to use a compiler to increase the security of a passively secure protocol.

Our approach only requires a small actively secure building block for a shared coin-toss and in general introduces a simpler construction for such compilers. With the assumption that the majority of the parties are honest, we reach the same deterrence rates as the aforementioned time-lock constructions with a less complex and much more practical approach. As we will show in this work, this is a natural assumption which has lead to some of the most promising MPC protocols.

To show this even further, our final contribution is a proof-of-concept implementation in which we apply parts of our compiler to a real-world MPC framework called MPyC [Sch18] which currently only supports passive security. We show how this design could work for arbitrary MPC protocols and how it is implemented for MPyC. This should be a good stepping stone for implementing the rest of our compiler to MPyC as well to obtain a covertly secure version of MPyC with public verifiability. We also benchmark this implementation and compare it to implementations with passive and active security.

1.5 Thesis Outline

In Chapter 2, the basic primitives used throughout this work will be introduced as well as the security models relevant for this research. In Chapter 3, an overview of the existing concrete MPC protocols will be given and compared to show the considerations regarding efficiency and security that can be made when designing concrete MPC protocols. In Chapter 4, the concepts of MPC compilers and player virtualisation will be explained. Next, in Chapter 5, our new design for compilers to transform passively security protocols into protocols with covert security and public verifiability will be demonstrated. In Chapter 6, the proof-of-concept implementation in MPyC will be elaborated. In Chapter 7, we will analyse both the complexity of our compiler from Chapter 5 as well as the practical performance of our implementation. Finally, we will summarise our conclusions and discuss this research in Chapter 8. Here, we also provide pointers towards future work.

Chapter 2

Background

In this chapter, we will look at the preliminary knowledge needed to for the rest of this work. First, in Section 2.1 will explain the cryptograhic primitives on which we will build our protocols. In Section 2.2, the fundamental techniques and concepts of multi-party computation will be explained. Finally in Section 2.3, the various adversarial models as well as other choices regarding the security of MPC protocols will be defined.

2.1 Cryptographic Primitives

2.1.1 Commitment Schemes

Commitment schemes are some of the most well-known building blocks for a lot of cryptographic protocols, including multi-party computation. A commitment scheme allows a party to *commit* to a message while keeping it hidden from other parties. Later, for example after executing a protocol, the scheme allows the party to reveal or *open* the message. Furthermore, such a scheme allows another party to verify that the message is indeed the message originally constructed by the party. A commitment scheme Π_{COM} consists of three functions and can be defined as follows [Dam98; Sma16]:

Definition 2.1.1 (Commitment scheme)**.** A commitment scheme consists of the following three algorithms:

- $(c, d) \leftarrow \texttt{Com}(m)$: The commitment algorithm $\texttt{Com}$ generates a commitment c and opening information d on a given message m.
- m or $\perp\ \leftarrow \texttt{Open}(c, d)$: The opening algorithm $\texttt{Open}$ yields a message m from a commitment c and opening information d or $\perp$ in case the opening information is invalid.
- $\{\texttt{accept, reject}\} \leftarrow \texttt{Verify}(pk, c, m)$: The verification algorithm, $\texttt{Verify}$, algorithm accepts if c is a valid commitment on message m given public key pk.

For a commitment scheme to be considered secure, it needs to be (computational/ information-theoretic) *binding* and *hiding*. The *binding* property means that an adversary can not change the message it has committed to later on. Formally, we can define the binding property as:

Definition 2.1.2 (Binding)**.** A commitment scheme is computational/ information-theoretic binding if a computationally bounded/ unbounded adversary can not create a commitment c for a message m and find another message $m'(m' \neq m)$ such that:

$$\texttt{Com}(m) = \texttt{Com}(m')$$

The hiding property ensures that given a commitment, an adversary is not able to determine what the underlying message is. Formally, this can be defined as:

Definition 2.1.3 (Hiding)**.** A commitment scheme is computational/ information-theoretic hiding if a computationally bounded/ unbounded adversary, given two messages m_1 and m_2 and commitment $c = \texttt{Com}(m_d)$, is unable to guess d better than just random guessing.

2.1.2 Signature Schemes

Digital signature schemes are a form of asymmetric encryption to provide an additional layer of security to messages communicated between parties. The idea is that a sender S "signs" a message using some private key when he sends to a receiver R. R can verify this signature, which provides three things: authenticity, integrity and non-repudiation. Authenticity ensures that R can validate the identity of S. By signing the message with a some key that only S could possibly know, R knows that if it received a valid signature, this message must indeed come from S. Integrity is obtained by knowing that if a message has been signed by S, any change of the encrypted message after the signature invalidates the signature. Therefore, if the signature is still valid upon arrival at R, R is convinced that the message is also the original message sent by S. Finally, non-repudiation is obtained by the fact that only S knows its private key. Therefore, if a valid signature was made with that private key, S must have sent the message and cannot claim otherwise. In general, a signature scheme consists of three algorithms:

Definition 2.1.4 (Signature Scheme)**.** A signature scheme consists of the following three algorithms:

- $(\text{sk}, \text{pk}) \leftarrow \texttt{Gen}()$: The Generation algorithm Gen generates a public-private key pair.
- $\sigma \leftarrow \texttt{Sign}_{\text{sk}}(m)$: The Signature algorithm Sign takes a message m and returns a signature σ under a secret key sk.
- $\{\texttt{accept}, \perp\} \leftarrow \texttt{Verify}_{\text{pk}}(m, \sigma)$: The verification algorithm Verify takes a message m and a signature σ and returns accept if σ is a valid signature on m given a public key pk. Otherwise, $\perp$ is returned.

Various security notions for these signature schemes can be found in literature. For this work, we will only consider signature schemes that provide *existential unforgeability against chosen message attacks*. *Existential unforgeability* states that it should be impossible for an adversary to construct ("forge") a valid message-signature pair (m, σ) that has not been signed by a legitimate signer. *Chosen message attack* means that this should even hold if the adversary can ask the signer for the signatures of an (unlimited) number of messages of his choice. Concretely, given a public key *pk* and a number of message-signatures pairs (m, σ) signed using the corresponding private key *sk*, it should be hard for an adversary to construct a new, valid message-signature pair (m', σ') that has nog been seen before, i.e., $(m', \sigma') \neq (m, \sigma)$ for every pair (m, σ) that was already retrieved from the signer.

2.2 Multi-Party Computation

Secure multi-party computation (MPC) is a tool for several parties to compute a joint function of their inputs without revealing the actual content of their inputs. This way a trusted third party can be emulated that might sometimes be hard or even impossible to find for real. Even though this line of research has been of interest since the beginning of the 80's, only recently these protocols have started to become practically relevant due to more efficient designs as well as increased computation power.

The first notion of *secure multiparty computation* was made in 1982 by Yao [Yao82]. He illustrated the concept of a 2-party setting in which two millionaires want to determine who of them is the richest without revealing their actual wealth. More formally, they both have some private input (their wealth) x_0 and x_1 and want to compare these by computing a function $f(x_1, x_2)$ that returns 0 if $x_0 \leq x_1$ and 1 if $x_0 > x_1$. After the protocol execution, neither party should know anything other than their own input and the outcome of the function. He proposed three specific solutions for *Yao's millionaires problem*, as it is often referred to nowadays. Four years later, he was also first to propose a 2-party MPC solution for arbitrary functions [Yao86] in which computational security against passive adversaries is guaranteed.

Based on Yao's 2-party construction, Goldreich, Micali and Wigderson proposed a general solution to the *n-party* case in 1987 [GMW87]. The GMW-protocol (or GMW-compiler) has been an important milestone in MPC since it was the first protocol to be secure against active adversaries in the computational model. Similar results were achieved almost simultaneously by Chaum, Damgård and Van de Graaf [CDG87], who did not rely on Yao's construction but solved the problem directly using zero-knowledge proofs.

These works from 1987 [GMW87; CDG87] served as the main inspiration for another milestone one year later when Ben-Or, Goldwasser and Wigderson [BGW88] as well as Chaum, Crépeau and Damgård [CCD88] presented the first *unconditionally (or information-theoretic)* secure, n-party, MPC protocols.

Since then, a lot of research has been done to make MPC protocols more secure, more efficient and more suitable to be used in practice. When looking at actual implementations for MPC protocols, an important design choice is the amount of parties the protocol supports. A distinction is made between two-party computation and multi-party computation. The possibility of having an honest majority in the multi-party case proves to have a lot of benefits for obtaining stronger security notions. The simplest model for this is a three-party computation with at most one corruption.
Looking at the research done on MPC protocols, three fundamentally different approaches can be identified: *Garbled Circuits*, *(Fully) Homomorphic Encryption* and *Secret Sharing*.

2.2.1 Garbled Circuits

Garbled Circuits (GC) are the oldest form of MPC techniques, being the underlying idea for the previously discussed protocols by Yao in the early 1980s and later formalised by Beaver, Micali and Rogaway in [BMR90].

The basic idea is the setting of two parties that together want to compute a function $f(a, b)$ where one party holds a and the other b. This function f is then represented as a binary circuit consisting only of XOR and AND gates.

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: Truth table of an AND-gate

Next, one of the parties gets the role of the generator while the other becomes the evaluator. The generator goes first and “encrypts" the binary circuit. Suppose the two parties want to compute the function $c = a \wedge b$, of which the truth table is depicted in table 2.1.

Now to encrypt (or garble) this gate, the generator assigns two keys to the two input wires and two keys to the output wire, resulting in: $k_{w,0}$ and $k_{w,1}$ where $w \in a, b, c$. These correspond to either a 0 or a 1 on the wire w. Now using a symmetric encryption scheme that takes two keys, encrypt the gate by forming the four ciphertexts:

$$\begin{aligned}
c_{0,0} &= Enc((k_{a,0}, k_{b,0}), k_{c,0}) \\
c_{0,1} &= Enc((k_{a,0}, k_{b,1}), k_{c,0}) \\
c_{1,0} &= Enc((k_{a,1}, k_{b,0}), k_{c,0}) \\
c_{1,1} &= Enc((k_{a,1}, k_{b,1}), k_{c,1})
\end{aligned}$$

These ciphertexts correspond exactly to 4 rows in the truth table of the AND gate. Finally, the rows of the truth table are permuted so that the evaluator does not know which ciphertext represents what wire.

Now, all the garbled gates as well as the wire keys corresponding to their own input are sent to the evaluator. The goal of the evaluator is then to obtain the correct wire keys corresponding to its own input. To let the evaluator obtain the correct wire keys for his input in a privacy-preserving way, the parties use a protocol called Oblivious Transfer. This allows the evaluator to obtain the correct wire keys without the generator learning the input of the evaluator.

In the final step, the evaluator can now use these inputs obtain the outcome of the end gate. For arbitrary circuits, this requires the evaluator to propagate these inputs through the circuit. For every gate, he decrypts the four ciphertexts of which (with a very high probability) only one will give a valid decryption. In the end, the evaluator will end up with the keys for the output wires that he sends to the generator, who in turn knows the mapping back to the actual output and obtains the result of the function.

This 2-party approach is similar to the protocol Yao described in 1986 [Yao86] and can be generalised to more than two parties. With certain optimisations that have been proposed this proves to be efficient enough for a low number of parties. The main advantage of the GC approach is the low round-complexity. Unlike other approaches, this technique only requires 1 round of communication between the parties. However, this approach generally scales poorly if we want to evaluate functions for a large number of parties due to the large communication overhead associated to transferring the encrypted circuit.

2.2.2 (Fully) Homomorphic Encrypion

Another, more recent technique for performing MPC is the use of homomorphic encryption schemes. These are encryption schemes where the ciphertexts contain some mathematical structure that can be exploited to perform calculations on the ciphertexts. This in combination with threshold decryption constitutes a fundamentally different technique for realising MPC functionality. Threshold decryption essentially gives every party only a subset of the decryption key, and only by combining their shares they can jointly decrypt a ciphertext.

Concretely, for a homomorphic encryption scheme there is a function Enc to encrypt plaintexts m_i and two operators $\bigoplus$ and $\bigotimes$ which can be equal but do not have to be. Due to the mathematical structure of the ciphertexts, the following identity now holds: $Enc(m_1 \bigoplus m_2) = Enc(m_1) \bigotimes Enc(m_2)$.

The types of operators and relationships between these operators vary per encryption scheme. A distinction can be made between *partially homomorphic encryption* and *fully homomorphic encryption*. In general, partially homomorphic encryption schemes only support one type of operation such as addition or multiplication where fully homomorphic encryption schemes allow for arbitrary operations and an arbitrary number thereof.

The arguably most famous partially homomorphic encryption schemes are the (textbook) RSA encryption scheme [RSA78] and the Pailler encryption scheme [Pai99]. In the RSA encryption scheme, multiplying two ciphertexts results in a multiplication of the underlying plaintexts while with Pailler, multiplying the ciphertexts corresponds to an addition of the plaintexts.

Several works use partially homomorphic encryption to realise MPC protocols. With additively homomorphic ecnryption, Franklin and Haber [FH93; FH96] introduced a passively secure MPC protocol with complexity $O(nkC)$, where C is the size of the (boolean) circuit and k a security parameter. In the active security setting, Cramer, Damgård and Nielsen also developed a protocol with the same complexity [CDN01].

A more powerful class of encryption schemes are the fully homomorphic encryption schemes, which support an arbitrary amount of arbitrary operations on the ciphertexts. The first encryption scheme supporting this was introduced by Gentry in 2009 [Gen09]. While there are various works that have been succesful in using homomorphic encryption for specific use-cases, the computational overhead of such schemes renders it impractical for many application scenarios.

2.2.3 Secret Sharing

The final technique for realising the MPC functionality is *secret sharing*. The general concept of secret sharing is to distribute a secret into a set of *shares* that can later be combined in a certain way to reconstruct the original secret. Formally, a secret-sharing scheme can be defined as follows [CDN15]:

Definition 2.2.1 (Secret-sharing Scheme)**.** A secret-sharing scheme among a group of parties $\mathcal{P} = \{P_1, \dots, P_n\}$ consists of the following two protocols:

- **Distribution:** For a certain party P_i to share a secret s, it is distributed in such a way that each party P_j obtains a share $s_j, 1 \leq j \leq n$.

- **Reconstruction** A qualified group of participants can combine their individual shares s_i to successfully reconstruct the underlying secret s.

Throughout this work, we will merely consider *threshold secret-sharing schemes*. These are secret-sharing schemes where any set of participants of size $\geq t$ is qualified to reconstruct the secret. The number of shares needed to correctly reconstruct the secret is called the *threshold* t. Such a scheme is called a *(n,t)-threshold scheme*, where t is the threshold of number of shares needed to successfully reconstruct the secret out of n secrets in total. A set of shares smaller than the threshold should perfectly hide the underlying secret.

The simplest threshold secret-sharing scheme is called *additive secret sharing*. In this scheme, a secret s is some element from a finite group G. To secret-share s among n parties, choose $n-1$ shares at random from G, $s_1, \dots, s_{n-1}$. The final share s_n will be picked such that $s = s_1 + s_2 + \cdots + s_n$. Note that s_n does not reveal any information of s due to the uniform random generation of $s_1, \dots, s_{n-1}$. Finally, to reconstruct the secret s, all the shares need to be combined. Note that any strict subset of shares does not leak any information on the secret x. Using additive secret-sharing in this way, we obtain an *(n,n)-threshold scheme* since all the secrets are needed to be able to reconstruct the original secret.

2.2.3.1 Shamir's Secret Sharing Scheme

Another widely used secret sharing scheme for MPC protocols is (variants of) Shamir secret sharing [Sha79]. This scheme is more flexible in the sense that an arbitrary reconstruction threshold $t \leq n$ can be chosen. This scheme is based on polynomial interpolation and the intuition that it takes t points to uniquely define a polynomial of degree $t-1$. For example, two points are needed to uniquely reconstruct a line, three points a parabola and so on.

This scheme works for a given threshold $t \leq n$, the number of players. Let $\mathbb{F}$ be a finite field with at least $n+1$ elements. Now, to distribute a secret $s \in \mathbb{F}$, choose $t-1$ random elements from the field $\alpha_1, \dots, \alpha_{t-1} \in \mathbb{F}$ and construct the polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}$ where a_0 is the secret x we want to share. To distribute the secret, every party receives an arbitrary non-zero point on this polynomial. Since the degree of this polynomial is at most $t-1$, the polynomial (and thus the secret) can be uniquely reconstructed via *Lagrange interpolation* if t or more parties combine their shares. On the other hand, any strict subset of the shares reveal no information on the polynomial nor the secret.

The secret can then be obtained by simply evaluating the result of $f(0) = \alpha_0 = s$. This makes the scheme flexible in the sense that not all the shares need to be combined but only a minimum amount of the chosen threshold t. This way, the secret can be recovered even if a subset (size smaller than or equal to $n-t$) of the parties do not cooperate.

2.2.3.2 Linear Secret-sharing Schemes

To construct an MPC protocol based on such a secret-sharing scheme, the *linearity* of this scheme is used. A *linear secret sharing scheme (LSSS)* allows parties to compute a secret sharing of an arbitrary linear function of the underlying secrets *without interaction*. To see how this linearity can be used to perform arbitrary computations, let's look at how this will work with Shamir's secret sharing scheme.

Let two secrets s and s' be hidden by polynomials $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{k-1}x^{k-1}$ and $g(x) = a'_0 + a'_1x + a'_2x^2 + \cdots + a'_{k-1}x^{k-1}$ and shared to the parties via $s = (s_1, s_2, \ldots, s_n)$ and $s' = (s'_1, s'_2, \ldots, s'_n)$. Furthermore, let $[s]$ denote a secret sharing of s. Now, arbitrary operations such as addition and multiplication can be performed as follows:

Addition Adding the two shares can simply be done by letting each party P_i add their shares of s and s' to obtain $[s] + [s'] = [s + s']$, which corresponds to $s + s' = s_0 + s'_0 + s_1 + s'_1 + \cdots + s_n + s'_n$ in the plaintexts. The reconstructed polynomial will indeed be of the form $f(x) + g(x) = a_0 + a'_0 + (a_1 + a'_1)x + \ldots + (a_{k-1} + a'_{k-1})x^{k-1}$, which hides $s + s'$. Multiplication with a constant factor (scalar) can be performed in the same, trivial way.

Multiplication Multiplication of two secret-shared values is more complex and often *does* require communication between the parties. Most of the computational complexity of secret-sharing based schemes often lies in the number of multiplications that need to be performed. Shamir's scheme does support multiplications, making it a *multiplicative LSSS*. By multiplying two secrets analogous to how addition is performed, a polynomial of the form $f(x) * g(x)$ is obtained. This indeed hides the secret $s * s'$ but has degree $2t$ and thus requires $2t + 1$ shares to be reconstructed.

To convert back to shares of a polynomial of degree t, after each party P_i computes their share of $s * s'$ by calculating $x_i = s_i * s'_i$, they *reshare* this local result to the other parties. Using all these shares, they can locally evaluate a linear function to get back to consistent shares on a polynomial sharing the secret $s * s'$. A concrete protocol for this approach can be found in Section 3.1.1.1.

2.2.3.3 Share-Compute-Reveal

In 2000, Cramer, Damgård and Maurer [CDM00] have shown that actually any *LSSS* can be used to perform multiplications in a similar fashion as explained for Shamir's scheme and thus supports non-linear operations. As a general blueprint for secret-sharing based MPC protocols, evaluating arbitrary functions is split into three distinct steps:

1. **Share:** First, each party *Pi* distributes its private input x^i to all the parties in the protocol.
2. **Compute:** Then, each party performs an arbitrary set of operations on its own shares. Here, a protocol can be *symmetric*, meaning the operations performed by all the parties are identical or *asymmetric*, where the operations that are performed vary per party.
3. **Reveal:** After all the parties performed the required operations, the individual shares are recombined to reveal the outcome to the correct parties.

Nowadays, most secret-sharing based MPC protocols make use of this so-called *share-compute-reveal* paradigm. The bottleneck of protocols following this blueprint mainly lies in the communication costs associated with non-linear operations, since each party needs to send $n - 1$ field elements to the other parties. This largely determines the efficiency of MPC protocols based on secret-sharing. However, the clear distinction between these three phases proves to be convenient for optimising the efficiency of such MPC protocols.

2.2.4 Publicly Verifiable Secret Sharing

The additive- and Shamir's secret sharing scheme from the previous section work with the assumption that the participants are honest, meaning they follow the protocol as they should. However, the privacy and correctness of these schemes break in case of a malicious participant. Such a participant may send incorrect shares to some of the parties or lie about the shares it has received, causing qualified sets of parties to reconstruct the wrong secret. To accommodate secret sharing with malicious participants the extended notion of *Verifiable Secret Sharing (VSS)* was introduced in 1985 by Chor et al.[Cho+85]. A VSS can be constructed from any secret sharing scheme with the additional properties that:

(i) The parties can verify that they received consistent shares from an untrusted dealer during the distribution phase;

(ii) The parties can verify that they received the correct shares from the other parties during the reconstruction phase.

On top of that, Stadler [Sta96] introduced *Publicly Verifiable Secret Sharing (PVSS)*. With publicly verifiable secret sharing, not only the participants in the protocol can verify their shares but *anyone* can verify (i) and (ii). Additionally, a *non-interactive* PVSS provide this property without the need to interact with the parties. Examples of such schemes can be found in [Sch99; Jan+20].

In general, a PVSS scheme consists of three algorithms for distribution, verification and reconstruction:

Definition 2.2.2 (PVSS Scheme)**.** A PVSS scheme with a set of players $\mathcal{P}$ consists of the following three algorithms:

- $(E_i(s_i)_{i\in\mathcal{P}}, \texttt{proof}) \leftarrow \texttt{Distribute}(s)$: The distribution algorithm takes as input a secret s and generated and publishes a set of encrypted shares $E_i(s_i)_{i\in\mathcal{P}}$ and some public `proof` which can be used for verification later.
- `true` or $\perp \leftarrow \texttt{Verify}(\texttt{proof}, E_i(s_i))$: The verification algorithm takes as input a proof and an encrypted share $E_i(s_i)$ and outputs `true` if $E_i(s_i)$ is a valid encryption on s_i according to `proof`. For ease of notation, we will let $\texttt{Verify}(\texttt{proof}, E_i(s_i)_{i\in A})$ denote the verification of all s_i's with $i \in A$. Now, `true` means all the verifications succeeded while $\perp$ means at least one verification failed.
- $s' \leftarrow \texttt{Reconstruct}(\texttt{proof}, E_i(s_i)_{i\in A})$: The reconstruction algorithm takes as input a proof, a set of encrypted shares $E_i(s_i)_{i\in A}$ of some subset $A \subset \mathcal{P}$ and outputs the reconstructed value s'. Note that $s' = s$ in case A is a qualified subset and the verifications of the encrypted shares succeeded according to the proof.

Here, it is assumed that we already have a registered public key of all the participants of some MPC protocol and additional system parameters have been generated beforehand. As can be seen, instead of generating and distributing the secrets directly, a dealer now publishes encrypted shares $E_j(s_j)$ with the known public keys of $P_j, j \in \mathcal{P}$. This means that if a party P_i wishes to share a secret s, he first generates the shares s_j, encrypts them with the corresponding public keys and publishes those along with a string $\texttt{proof}_i$ which shows that each E_j indeed encrypts the share s_j. This proof also commits the dealer to the value of the secret s and guarantees that no one can wrongly claim to have received a wrong share since anyone can verify this. Therefore, if the reconstruction succeeds, we are guaranteed that this is the original secret s.

For the reconstruction phase, the parties first decrypt their shares s_j from $E_j(s_j)$. After that, they compute a string $\texttt{proof}_j$ which shows that they performed the decryption correctly and publish s_j and $\texttt{proof}_j$. Using these proofs for reconstructing the secret, the other parties can now exclude the shares of dishonest participants or participants who failed the decryption step. If enough decryptions ($t + 1$) pass the verification, the parties can reconstruct the original secret successfully.

For a PVSS scheme to be considered secure, we require the scheme to satisfy the *correctness, soundness* and *privacy* properties.

Definition 2.2.3 (Correctness)**.** If a dealer honestly follows the `Distribute` algorithm to publish the encrypted shares $E_i(s_i)_{i\in\mathcal{P}}$ and a public proof `proof`, then the outcome $\texttt{Verify}(\texttt{proof}, E_i(s_i))$ is guaranteed to be `true`. Furthermore, if during reconstruction a party P_i honestly decrypts $E_i(s_i)$, publishes its share s_i and honestly generates the proof $\texttt{proof}_i$, then another party receiving the decrypted share s_i and $\texttt{proof}_i$ accepts this share. Finally, a qualified subset $\mathcal{A} \subseteq \mathcal{P}$ is guaranteed to reocnstruct the original secret s if the dealer and the parties in $\mathcal{A}$ honestly follow the `Distribute` and `Reconstruct` protocols.

Definition 2.2.4 (Soundness)**.** If $\texttt{Verify}(\texttt{proof}, E_i(s_i)) == \texttt{true}$, then for every qualified subsets $\mathcal{A}_1, \mathcal{A}_2 \subset \mathcal{P}$, the following holds:

$$\texttt{Reconstruct}(\texttt{proof}, E_i(s_i)_{i\in\mathcal{A}_1}) == \texttt{Reconstruct}(\texttt{proof}, E_i(s_i)_{i\in\mathcal{A}_2})$$

.
Furthermore, if a malicious party submits a fake share during reconstruction, verification of this share fails with an overwhelming probability.

Definition 2.2.5 (Privacy)**.** An adversary corrupting a set of participants C such that $|C| < t$ should not be able to learn anything about the secret s from the shares s_i with $i \in C$.

2.2.5 Efficiency

Naturally, performing calculations on data in a privacy-preserving way comes at a cost. There are three main properties that influence the efficiency of an MPC protocol:

1. The *round complexity* that specifies the number of communication rounds needed to evaluate the function.
2. The *communication complexity*, which is defined as the number of bits that are communicated between the parties.
3. The *computational complexity* which is the number of additions and multiplications that are performed by all the parties.

Unfortunately, there is no clear-cut solution that performs best in all of these aspects. Some approaches are naturally very efficient in terms of communication rounds but are very computation- and communication intensive such as garbled circuits while others require more rounds but demand less computational power from the parties. Which approach is best depends on the use-case. For example, if the parties consist of small IoT devices, it is wise to keep the computational load low while a low-bandwidth or high-latency environment requires a low number of communication (rounds).

Furthermore, there is a distinction between *concrete efficiency* and *asymptotic efficiency*. Asymptotically efficient protocols mainly focus on how certain parts such as round-, communication- or computational complexity scale in factors such as the circuit size, the number of parties or the deterrence rate. On the other hand, concretely efficient protocols primarily care for actual performance in terms of overall running time even if the protocol is not optimal in terms of complexity. Asymptotically efficient protocols often provide techniques that can be used to construct concretely efficient protocols.

2.2.5.1 Pre-processing Model

One important efficiency improvement for a large number of MPC protocols is the so-called *pre-processing model* which splits the problem into a *pre-processing* phase and an *online* phase. The pre-processing step is completely independent from the parties' inputs and can thus be performed beforehand. During this phase, the parties can produce correlated randomness that can later be consumed during the more complex online phase. By moving the majority of the heavy computations, a lot of state-of-the-art protocols obtain very efficient solutions in the online phase.

2.3 Defining Security

The goal of Multi-Party Computation (MPC) protocols is to allow a group of participants $P_1, \ldots, P_n$ to compute a shared function f over their private inputs $x_1, \ldots, x_n$ while keeping their inputs hidden from each other. This group of participants can be divided in two sets: *honest* participants and *corrupt* participants. The honest participants will strictly follow the protocol description while corrupt participants are assumed to be under the influence of a central adversary. What it means to evaluate a protocol securely in the presence of such an adversary can be defined in the real/ideal world paradigm [BPW04].

In general for an MPC protocol to be considered secure, it needs to satisfy two requirements: *privacy* and *correctness*.

Definition 2.3.1 (Privacy)**.** The only new information an adversary is able to learn from running the protocol are the output(s) of the protocol.

Therefore, an MPC protocol should ensure that a possible adversary is not able to derive any information about the inputs of the other parties in the protocol.

Definition 2.3.2 (Correctness)**.** The outcome(s) of the protocol received by the honest parties should be correct.

The correctness requirement can change depending on the guarantees that are desired. Sometimes it suffices to be guarantees that *if* a party receives an output, it is correct while for other protocols stronger guarantees are needed. The possible guarantees will be elaborated later.

The ideal way for realising a protocol satisfying these requirements is for each participant to send their input privately to a trusted third party, $\mathcal{F}$, who will compute the function $f(x_1, \ldots, x_n)$ and send this back to each party. $\mathcal{F}$ is incorruptible and always calculates the correct result. However, in the real-world setting, such a party might be costly and sometimes even impossible to find. Instead, MPC is an umbrella term for cryptographic techniques which simulate such a trusted third party in the real world by means of a protocol, Π, the parties can execute among themselves. An MPC protocol is able to satisfy the above requirements while the parties do not need to trust anyone. The trick often used to proof the security of an MPC protocol is to show that the protocol is indistinguishable from the ideal functionality, which is defined to be secure. The assumption being that if a protocol Π *looks* like another protocol which is known to be secure, Π should be secure as well. This idea is known as the transivity of security and was first formulated by Canetti in the UC framework [Can01]. Now, we define security in the real/ideal world as follows [Lin17]:

Definition 2.3.3 (Security in the real/ideal world paradigm)**.** Let $\mathcal{F}$ be a trusted third party that calculates the function f in the ideal world and let $\Pi : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party protocol calculating f in the real world. Π takes one input from every party $P_1, \ldots, P_n$ and yields every party one outcome. Furthermore, let $\mathcal{A}$ be an adversary in the real-world and $\mathcal{S}$ a simulator in the ideal world. Now, Π is said to compute f in a secure manner if for all real-world adversaries $\mathcal{A}$, we can find an ideal-world simulator $\mathcal{S}$ such that the output distribution of $\mathcal{S}$ and the honest parties is indistinguishable from executing Π in the presence of $\mathcal{A}$ in the real world.

The intuition of this real/ideal world security definition is that for any possible adversary in the real world, the information it can compute can be simulated in the ideal world and thus it can only perform attacks which would also be possible in the ideal world. This way, no more security can be compromised in the real world than possible in the ideal world.

2.3.1 Adversaries

To reason about the security against a hypothetical adversary, the capabilities of this adversary are formalised. First, an adversary is is capable of *corrupting* a set of participants in the protocol. This set of corrupted parties is either known upfront, in which case we speak of a *static* adversary or can change during the execution of the protocol, in which case the adversary is said to be *adaptive*.

An adversary is able to access the *views* of all the parties it corrupts. Such a view is defined as follows [Lin17]:

Definition 2.3.4 (View)**.** The `view` of party P_i during the execution of Π is denoted as $\texttt{view}_i^{\Pi}$ and consists of $(x_i, r^i; m_1^i, \ldots, m_t^i)$. Here, x_i is the input of P_i, r^i is the random tape of P_i and m_j^i, $1 \leq j \leq t$ the jth message received by P_i.

Furthermore, the adversary is able to combine the views it receives from all the parties it corrupts. All adversaries are characterised by the computational power they possess. Here, a distinction is made between adversaries that are bounded to polynomial-time computations and adversaries that have unbounded resources. Protocols allowing only bounded adversaries are said to be *cryptographic (or computational)* secure while protocols that protect against the latter are *information-theoretic* secure. Next, the *passive*, *active* and *covert* adversary models will be explained.

2.3.1.1 Passive Adversaries

A passive adversary, also called a semi-honest or honest-but-curious adversary, is an adversary which uses the views of the parties it corrupts to calculate as much information as possible while still following the protocol honestly. For example, such an adversary might try to combine the views of the corrupt parties to break the *privacy* of the inputs of one of the honest parties. In general, such an adversary is not able to break the *correctness* requirement of MPC protocols. In order to give a formal definition of security in the passive security model, the execution of a protocol in the real world and the ideal functionality need to be made concrete.

Suppose we want to calculate the function $f(x_1, \dots x_n) = (y_1, \dots, y_n)$. In the real world, we can execute an MPC protocol Π among parties $P_1, \dots, P_n$ with private inputs $x_1, \dots, x_n$. For simplicity, denote the vector of inputs as $\bar{x} = (x_1, \dots, x_n)$. Furthermore, let C be the set of parties that are corrupted by adversary $\mathcal{A}$. Now, we define $\texttt{REAL}_k[\mathcal{A}, C, \Pi, \bar{x}]$ as the output of $\mathcal{A}$ and the outputs of the honest parties in the execution of Π given a security parameter k.

In the ideal world, performing the same calculations is denoted as the functionality $\mathcal{F}_{\text{Passive}}$. A formal description of this ideal functionality can be found in ideal functionality $\mathcal{F}_{\text{Passive}}$.

Ideal Functionality $\mathcal{F}_{\text{Passive}}$

1. **Inputs:** In the ideal world, the environment $\mathcal{Z}$ sends all the inputs $x_1, \dots, x_n$ to $\mathcal{F}_{\text{Passive}}$.
2. **Reveal inputs:** The ideal world adversary (simulator) $\mathcal{S}$ is able to obtain the inputs of all the corrupt parties C by sending `get_inputs` to $\mathcal{F}_{\text{Passive}}$.
3. **Output:** $\mathcal{F}_{\text{Passive}}$ computes $(y_1, \dots, y_n)$ and returns back y_i to each P_i. All the honest parties simply output the result they receive while the adversary $\mathcal{S}$ outputs an arbitrary function of the initial inputs of the corrupted parties and the outputs received from $\mathcal{F}_{\text{Passive}}$.

Now, define $\texttt{IDEAL}_k[\mathcal{S}, C, \mathcal{F}_{\text{Passive}}, \bar{x}]$ as the joint distribution of the outputs of the honest parties and the adversary $\mathcal{S}$. Here, $\overset{c}{\equiv}$ denotes computational indistinguishably.

Definition 2.3.5 (Passive security). A protocol Π securely computes $\mathcal{F}$ with security against passive adversaries if for all real-world adversaries $\mathcal{A}$ we can find an ideal-world adversary $\mathcal{S}$ such that for all security parameters $k \in \mathbb{N}$:

$$\{\texttt{IDEAL}_k[\mathcal{S}, C, \mathcal{F}_{\text{Passive}}, \bar{x}]\}_{\bar{x} \in \{0,1\}^*} \overset{c}{\equiv} \{\texttt{REAL}_k[\mathcal{A}, C, \Pi, \bar{x}]\}_{\bar{x} \in \{0,1\}^*}$$

Which closely resembles the earlier definition 2.3.3 for general security in the real/ideal world model. Intuitively, this definition states that the output distribution in the real world should not be distinguishable from the output distribution in the ideal world for arbitrary inputs. This captures the idea that only the attacks possible in the ideal world are possible in the real world.

2.3.1.2 Active Adversaries

An active adversary or malicious adversary has all the power of a passive adversary in the sense that it gets access to all the views of the corrupted parties but on top of that *can* deviate from the standard protocol execution. For example, this allows him to send incorrect messages or stop sending messages altogether in order to violate the correctness of the output for the honest parties. MPC protocols that allow existence of such adversaries are said to have *active security*. Furthermore, it might be possible to identify exactly *who* did the cheating, for example to be able to remove him from a new attempt at executing the protocol. This stronger notion is called *cheater detection*.

This security notion can again be defined by formulating an ideal world for calculating a function f in the presence of an active adversary. In this case, The adversary $\mathcal{S}$ does not only get access to the views of the corrupted parties, but gets full control over these parties. The ideal functionality $\mathcal{F}_{\text{ACTIVE}}$ again computes all the outputs but instead of sending these to the corresponding parties directly, the ideal functionality sends the output to a hypothetical environment $\mathcal{Z}$ which decides whether the parties should obtain this output or $\perp$. This corresponds to the adversary prohibiting the honest parties from receiving the correct outcome.

2.3.1.3 Covert Adversaries

In 2007, Aumann and Lindell [AL07] proposed a new definition called a *covert* adversary. The idea is that this type of adversary is a bit less strict in the sense that he *is* capable of performing an active attack, but a certain chance of being caught cheating is enough to refrain him from doing so. Even if the chance of being caught is small, the cons of being caught might still be large enough to stop the adversary from cheating. The probability of being caught cheating is called the *deterrence rate* ϵ.

This idea seems more realistic and applicable to real-life scenarios. When looking at traditional security, even banks are not 100% secure against intruders but the idea of getting caught and sent to jail is most often enough to stop potential intruders from breaking in.

Aumann and Lindell originally defined three notions for covert security but in later years, the strongest model of *strong explicit cheat (SECF)* became the standard when judging covert secure protocols. We follow this model with one alteration, namely that if the ideal functionality receives ($\texttt{corrupted}, i$) as an input from a party, it will only send ($\texttt{corrupted}$) to the honest parties. The original notion is called *identifiable abort*, which is not achieved by our compiler since this has proven to be a hard guarantee to achieve in other research works on MPC. Our ideal functionality for covert security can be found in $\mathcal{F}_{\text{Covert}}$

The ideal functionality for calculating a function f in the presence of a covert adversary according to the SECF notion will be called $\mathcal{F}_{\text{Covert}}$, which allows the ideal-world adversary to perform a limited amount of cheating like an active adversary. He can attempt to cheat by informing the ideal functionality, but the $\mathcal{F}_{\text{Covert}}$ will randomly decide whether the attempted cheating was successful or not. With a probability of ϵ, the deterrence rate, $\mathcal{F}_{Covert}$ will inform all the parties of at least one corrupt party that tried to cheat. With a probability of $1-\epsilon$, the cheating was successful, meaning the simulator $\mathcal{S}$ learns all the parties' inputs and can decide what their output is. Now, the formal definition of this ideal execution can be found in ideal functionality $\mathcal{F}_{\text{Covert}}$ [DOS20].

Here, the input phase in Step 1 is slightly changed compared to the passive security ideal world in the sense that not all the correct inputs are given to the ideal functionality directly. Furthermore, steps 2 and 3 correspond to the possible ways for an active adversary to cheat during the computation phase in the protocol. Here, an adversary can abort the protocol, in which case the honest parties notice this. Furthermore, an

Ideal Functionality $\mathcal{F}_{\text{Covert}}$

1. **Inputs:** Every honest party P_i sends its input x_i to $\mathcal{F}_{\text{Covert}}$. The ideal world adversary $\mathcal{S}$ sends inputs on behalf of all the corrupted parties.
2. **Abort Options:** A corrupt party may send $(\texttt{abort}, i)$ or $(\texttt{corrupted}, i)$ as input to $\mathcal{F}_{\text{Covert}}$. If $(\texttt{abort}, i)$ was received, $\mathcal{F}_{\text{Covert}}$ will respond by sending $(\texttt{abort})$ to all the honest parties and halt. In case $(\texttt{corrupted}, i)$ was received, $\mathcal{F}_{\text{Covert}}$ sends $(\texttt{corrupted}, i)$ to all the honest parties and halts. If multiple parties send `corrupted` or `abort`, $\mathcal{F}_{\text{Covert}}$ informs the honest parties of only one of these events and halt. Furthermore, $(\texttt{corrupted}, i)$ is ignored in case it receives a combination of both events.
3. **Attempted cheat:** $\mathcal{S}$ can send $(\texttt{cheat}, i)$ as input of a corrupted party P_i to $\mathcal{F}_{\text{Covert}}$. Now, $\mathcal{F}_{\text{Covert}}$ will respond with `detected` with a probability of ϵ to all the parties and `undetected` with a probability of $1 - \epsilon$ to the adversary. In case the cheating attempt was undetected, $\mathcal{S}$ gets all the inputs x_i of the honest parties P_i and specifies an output y_i for each of them which $\mathcal{F}_{\text{Covert}}$ will output to P_i.

This is normally the end of the ideal execution. However, if no corrupted party sent $(\texttt{abort}, i)$, $(\texttt{corrupted}, i)$ or $(\texttt{cheat}, i)$, the ideal execution continues with:

4. **Answer adversary:** The ideal functionality computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and sends it to $\mathcal{S}$.
5. **Answer honest parties:** $\mathcal{S}$ can now decide to either `continue` or $(\texttt{abort}, i)$ for a corrupted P_i. In case the adversary continues, the ideal functionality returns y_i to each honest P_i. In the case of $(\texttt{abort}, i)$, the ideal functionality relays this to all honest parties.
6. **Output:** Honest parties always output the message they receive from $\mathcal{F}_{\text{Covert}}$ where the corrupted parties output nothing. The adversary outputs an arbitrary function of the initial inputs of the corrupted parties and the outputs received from $\mathcal{F}_{\text{Covert}}$.

adversary can be identified as corrupt, which will be seen by the honest parties as well. Finally, an adversary can attempt to make a corrupt party *cheat*, in which case there is a chance $1 - \epsilon$ of being successful. The last three steps correspond to the case in which the adversary follows the computation phase honestly but cheats in the reveal phase of the protocol. Similar to the passive security case, the joint distribution of the outputs of the honest parties and $\mathcal{S}$ is now denoted as $\texttt{IDEAL}_k^{\epsilon}[\mathcal{S}, C, \mathcal{F}_{\text{Covert}}, \bar{x}]$. The definition for covert security is now analogous to definition 2.3.5 for passive security:

Definition 2.3.6 (Covert security with deterrence rate ϵ)**.** A protocol Π securely computes $\mathcal{F}$ with security against covert adversaries with a deterrence rate of ϵ if for every real-world adversary $\mathcal{A}$, we can find an ideal-world adversary $\mathcal{S}$ such that for all security parameters $k \in \mathbb{N}$:

$$\left\{\texttt{IDEAL}_k^{\epsilon}[\mathcal{S}, C, \mathcal{F}_{\text{Covert}}, \bar{x}]\right\}_{\bar{x} \in \{0,1\}^*} \overset{\text{c}}{\equiv} \{\texttt{REAL}_k[k, \mathcal{A}, C, \Pi, \bar{x}]\}_{\bar{x} \in \{0,1\}^*}$$

2.3.1.4 Publicly Verifiable Covert (PVC) Security

While this notion of covert security described in the previous paragraph has been useful for MPC, it may still not be fully sufficient to discourage cheating by the adversary. Covert security only guarantees that if an adversary is cheating, this is detected by a party with a certain probability. However, it is impossible for this party to later proof to the other parties or even a third party that cheating actually occurred. This becomes especially problematic if there is some asymmetric power relationship between the parties. If, for example, a bank performs an MPC protocol with its clients and one client detects cheating, the bank can simply claim it did not cheat and the client has no leg to stand on. As a result, the bank escapes without any repercussions other than losing its reputation with this single client.

With this in mind, the notion of *publicly verifiable* covert security (PVC) was proposed by Asharov and Orlandi in 2012 [AO12]. This form of security provides the parties with a mechanism to generate a publicly verifiable *certificate* which undeniably convinces that a certain party has cheated. Now instead of losing reputation with only one party, reputation is lost with all the other parties and thus the consequences of cheating become much higher. Because of this construction, a larger level of dissuasion for cheating may be obtained.

In terms of the real/ideal world paradigm, we use the simplified notion by [Hon+19] where a `Judge` algorithm is added to a real-world protocol Π. If, in the execution of Π, cheating is detected, the protocol outputs a certificate `cert`. The `Judge` algorithm verifies this certificate and outputs the identity of the cheater, defined by the corresponding public key or nothing in case the certificate is invalid. The vector of public keys is defined as $\bar{pk} = (pk^1, \ldots, pk^n)$, corresponding to the P_is. Furthermore, we have extracted the verification procedure of the protocol to a separete `Blame` algorithm. `Blame` takes the view of a party P_i and returns a certificate `cert` and outputs $\texttt{corrupted}_j$ in case party P_j is found to be cheating Formally, we define covert seucurity with public verifiability as:

Definition 2.3.7 (Covert security with deterrence rate ϵ and public verifiability)**.** A protocol $(\Pi, \texttt{Blame}, \texttt{Judge})$ securely computes $\mathcal{F}$ with security against covert adversaries with a deterrence rate of ϵ and public verifiability if the following three conditions hold:

1. **Covert security:** Π is secure against a covert adversary according to definition 2.3.6 for covert security with deterrence rate ϵ. Additionaly, Π might now output `cert` in case cheating is detected.
2. **Public Verifiability:** If an honest party P_i detects cheating by another party P_j and outputs `cert` in an execution of Π, then $\texttt{Judge}(\bar{pk}, \mathcal{F}, \texttt{cert}) = pk^j$ except with negligible probability.
3. **Defamation-Freeness:** If party P_i is honest and executes Π in the presence of an adversary $\mathcal{A}$, then the probability that $\mathcal{A}$ creates $\texttt{cert}^*$ such that $\texttt{Judge}(\bar{pk}, \mathcal{F}, \texttt{cert}^*) = pk^i$ is negligible.

The intuition of this is that the protocol should have all the same functionality as a protocol with plain covert security. Furthermore, in case cheating is detected, the public verifiability mechanism should be able to convince other parties with an overwhelming probability. Finally, the defamation-freeness requirement states that it should be near impossible for an adversary to trick other parties into believing that an honest party has cheated during the execution of the protocol.

2.3.2 Security Guarantees

The level of security of an MPC protocol required or desired is of great influence on the design and efficiency of the protocol. Here, increasing the guarantees of an MPC protocol often comes at the cost of a less efficient protocol. The security of an MPC protocol is defined by the two requirements *privacy* of the inputs and *correctness* of the output.

As stated earlier, what it means for an output to be ‘correct’ depends on the security guarantees desired for the protocol. In its weakest form, the correctness requirement states that the honest participants do not receive incorrect outputs. Therefore, it should be impossible for an adversary to manipulate the protocol execution in such a way that the honest parties receive an incorrect result.

This weakest notion of correctness is called *security with abort*. In this setting, it is only guaranteed that the honest parties never receive a wrong output. This means that it is possible for an adversary to obtain the correct output and prevent the honest parties from receiving the output. While this is a rather weak notion of correctness, it can be sufficient in certain applications and does allow protocols to tolerate a large amount of corruptions.

A stronger notion of correctness to prevent an adversary from “stealing" the output is the notion of *fairness*. Fairness states that the honest parties should always obtain the output if the adversary does. Note that it is still possible that no one receives the correct output. Compared to security with abort, lower amounts of corruptions can be tolerated while ensuring fairness.

An even stronger notion of correctness to prevent the honest parties from receiving no output is the notion of *robustness*. Robustness ensures that honest parties always obtain the correct output, regardless of the actions of the adversary. As this is even harder to guarantee than fairness, an even smaller number of corruptions can be tolerated.

2.3.3 Corruption Thresholds

The amount of corruptions that can be tolerated by a protocol is called the *corruption threshold*, t. In general, this threshold can be any number smaller than the total number of players n (note that reasoning about a protocol execution where all the participants are malicious does not make sense intuitively). However, to reach certain security guarantees for various adversary models, the amount of corruptions that can be tolerated are limited. Common bounds are $t < n$ (any number of corruptions), $t < \frac{n}{2}$ (honest majority) and $t < \frac{n}{3}$.

Tight bounds on the amount of corruptions that can be tolerated for the computational models were already established in the late 80s. [GMW87; CDG87; BGW88; CCD88] have shown that in the computational model, any function can be calculated guaranteeing robustness against passive adversaries while tolerating any number of corruptions, i.e., $t < n$.

If security against active adversaries is desired, an *honest majority* is required to obtain the same guarantees.

In the information-theoretic model, they have proven that these bounds lower to $t < \frac{n}{2}$ against passive adversaries and $t < \frac{n}{3}$ against active adversaries to guarantee robustness. Relaxing to fairness increases the latter to $t < \frac{n}{2}$.

On the other hand, tolerating a *dishonest majority* comes at a significant cost. In this case, information-theoretic security is impossible to achieve and even with computational security, the adversary is able to abort the protocol after he learns the outcome and thus, fairness is impossible to achieve. In the case of 2-party computation, one corruption immediately constitutes a dishonest majority and thus 2-party protocols only work in the computational setting by tolerating only passive adversaries or settle for weaker security guarantees.

2.3.4 Interplay Between Properties

All the choices that can be made regarding the adversarial power, guarantees and circumstances in which the protocol operates have had a large influence on the design of MPC protocols in the literature. A summary of all these choices can be found in table 2.2. In general, the stronger the desired level of security, the slower the protocols that can be designed. Furthermore, some combinations of guarantees are impossible to combine. It is impossible to obtain an MPC protocol satisfying all the strongest security requirements. Oftentimes it is needed to compromise some security for more efficiency or stronger guarantees. In terms of adversarial models, covert security (with public verifiability) can be seen as an abstraction of passive and active security (with cheater detection) where it is often possible to tweak the deterrence rate to go from passive security with a low ϵ up to active security as ϵ approaches 1.

Security Choice	**Options**
Comp. power	• Computional security • Information-theoretic security
Number of corruptions	• $< n/3$ • $< n/2$ • $< n$
Adversary model	• Passive • Covert • Active
Adversary type	• Static • Adaptive
Security Guarantees	• Security with abort • Fairness • Robustness
Additional mechanisms	• Public verifiability (covert) • Cheater detection (active)

Table 2.2: MPC protocol security choices

Chapter 3

Concrete MPC Protocols

Based on the three fundamental techniques from Section 2.2, numerous concrete MPC protocols have been developed over time. In this chapter, the most important works for achieving concrete protocols in the passive-, active- and covert security models will be explained. Furthermore we will look at how the assumption of an honest majority of parties has lead to more efficient protocols.

3.1 Passive Security

Historically, there have been two general approaches for secure 2-party MPC in the passive or semi-honest adversary setting: Yao's approach [Yao86] based on garbled circuits and the GMW protocol [GMW87] based on secret-sharing.

Both constructions in their most simple form represent the functionalities as Boolean circuits. The idea of the GMW protocol is to first have the parties secret share their input bits, which means that for every input wire in the circuit, they split their share in two random bits α and β s.t. $\alpha \bigoplus \beta$ is the actual private input bit of the wire. Then they walk through the circuit and compute the outcome of every gate with their own shares to obtain random shares of the output of every gate. For linear operations such as the XOR gate this can be done locally, but for an AND gate, some communication is needed. In the end, each party will end up with shares of the output wires which they can communicate to each other to reconstruct the actual output.

The difficulty in terms of complexity of this circuit evaluation approach lies in the size of the circuit. For every AND gate of the circuit, the parties need to run an *oblivious transfer*. This communication overhead is fine for circuits that are not too large, but typically the GC approach by Yao in 1986 [Yao86] is more efficient for large circuits since oblivious transfer is only required to transfer the input bits after which only linear operations are needed to compute the gates.

3.1.1 BGW Protocol

To perform computations over arithmetic circuits, one of the most widely used protocols is the passively secure version of the BGW protocol [BGW88]. This protocol is based on Shamir's secret sharing scheme from Section 2.2.3.1. Recall that this is a *threshold* secret-sharing scheme, meaning it works with a certain threshold, t, where t + 1 participants can combine their shares in order to reconstruct the secret while less than t shares does not reveal any information on the underlying secret.

For n parties, this protocol provides information-theoretic security against $t < n/2$ passive corruptions and $t < n/3$ active corruptions. Furthermore, [BGW88] has proven that these bounds are optimal in the sense that nothing better than these bounds can be obtained in the information-theoretic setting. A concrete description of the protocol with passive security can be found in the protocol description of Π_{BGW}.

Protocol Π_{BGW}

This protocol works with an arbitrary number n of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ holding private inputs $x_1, x_2, \ldots, x_n$. Furthermore, the protocol assumes the parties agreed on a certain threshold $t < n/2$. Now, the parties jointly run the protocol in the following manner:

1. **Input sharing phase:** First, party $P_i \in \mathcal{P}$ uses Shamir's secret-sharing scheme with threshold $t + 1$ to obtain a set of shares $x_i^1, x_i^2, \ldots, x_i^n$ and sends share x_i^j to P_j.
2. **Circuit evaluation phase:** Now, the parties jointly emulate the computation of the circuit gate by gate. The parties compute their own share of the output of every gate in the following manner:
   (a) *Addition gate:* Suppose the inputs to the gates are a and b, where P_i holds a_i and b_i. Now, the parties can compute a secret sharing of the output of the gate by simply calculating $a_i + b_i$ locally.
   (b) *Multiplication by a constant gate*: Multiplication with a constant can also be done locally (i.e., without any interation). Suppose the inputs to the gate are a and some constant c, the output of the gate $c * a$ can be calculated by letting each P_i calculate $c * a_i$ locally.
   (c) *Multiplication gate:* Suppose the parties want to calculate their share of the output of a multiplication gate with inputs a and b. Simply letting each party P_i compute $a_i * b_i$ locally does not work since the resulting polynomial will share $a * b$ as required, but has a degree of $2t$ instead of t. Furthermore, the polynomial is not truly random anymore, which might cause information leakage during the reconstruction phase. Instead, the parties use an additional interactive protocol which will be called F_{mult} to compute the multiplication functionality.

      This protocol is defined as $F_{mult}((f_a(\alpha_1), f_b(\alpha_1)), \ldots, (f_a(\alpha_n), f_b(\alpha_n))) = (f_{ab}(\alpha_1), \ldots, f_{ab}(\alpha_n))$.
3. **Output phase:** After the evaluation phase, the parties hold shares of the output wires. The parties can reconstruct the actual output by sending their shares to the other parties and use $t + 1$ shares to reconstruct the output.

Intuitively, this protocol implements the share-compute-reveal paradigm in the sense that first the private inputs are secret-shared, then the parties evaluate an arithmetic circuit consisting of addition and multiplication gates to obtain shares of all the wires. Lastly, the parties reconstruct the output by publishing their respective shares of the output wire. For a more detailed description of why this works, we refer back the Section 2.2.3.1 on Shamir's secret sharing scheme. For an example for a multiplication functionality $\mathcal{F}_{\text{MUL}}$ to perform multiplications, see Section 3.1.1.1. Essentially, $\mathcal{F}_{\text{MUL}}$ is a functionality which takes as input two secrets a and b, shared via degree-t polynomials f_a and f_b respectively, and produces a degree-t polynomial f_{ab} which shares the secret $a * b$ as its constant term.

3.1.1.1 Multiplication by Gennaro et al.

In 1998, Gennaro et al. [GRR98] presented a simple protocol Π_{MUL} for implementing $\mathcal{F}_{\text{MUL}}$ compatible with the BGW aproach. Suppose again we have two secrets a and b shared via degree-t polynomials $f_a(x)$ and $f_b(x)$. Here, party P_i holds share $f_a(i)$ and $f_b(i)$ respectively. In order to compute the result of $a * b$, the parties first compute $f_{ab}(i) = f_a(i) * f_b(i)$, which together will produce the degree-2t polynomial:

$$f_{ab}(x) = f_a(x) f_b(x) = \gamma_{2t} x^{2t} + \ldots + \gamma_1 x + ab$$

To reduce this back to a degree-t polynomial, the parties engange in a process called *resharing*. In this process, party P_i will *reshare* its share on the degree-2t polynomial by choosing a random degree-t polynomial $h_i(x)$ such that $h_i(0) = f_{ab}(i)$. By now sending each party P_j the value $h_i(j)$, the parties can obtain consistent shares on $f_{ab}(x)$, which has $a * b$ as its constant term, by locally computing

$$h(j) = \sum_{i=1}^{2t+1} \lambda_i * h_i(k)$$

The constants can be calculated by the parties as

$$\lambda_i = \prod_{1 \leq k \leq 2t+1, k \neq i} \frac{k}{k-i}$$

A concrete description of the protocol by Gennaro et al. can be found in the protocol description of Π_{MUL}.

3.2 Multiplication Triples

The GMW and BGW protocols from the previous section can also be optimised by using so-called *multiplication triples (or Beaver triples)* [Bea91] to perform the multiplications. This construction is not necessarily better or worse than for example the Gennaro approach, but allows certain computations to be performed in a pre-processing phase because they are independent of the input values. By moving the most complex tasks to this pre-processing phase, the actual computations which are done in the *online* phase of the protocol becomes a lot faster.

A multiplication triple consists of secret shared values $[a]$, $[b]$ and $[c]$ where a and b are uniformly random and $c = a * b$. Given this, two secret-shared values $[x]$, $[y]$ can be multiplied in the following way: first calculate $d = x - a$ and $e = y - b$ and publicly reveal the result. This is dependent on their shares of the secrets, but is masked by the random values a and b and can thus be revealed safely. Using these, the parties can locally compute compute:

Protocol Π_{MUL}

This protocol works with an arbitrary number n of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. Furthermore, each party P_i holds shares $f_a(i)$ and $f_b(i)$ for respectively secrets a and b. They produce a degree-t polynomial sharing ab in the following manner:

1. **Local multiplication:** First, the parties locally compute the product $f_a(i)f_b(i)$, which constitutes a degree-2t polynomial sharing ab.
2. **Resharing:** Now, party P_i *reshares* this share by choosing a new polynomial $h_i(x)$ of maximum degree-t (as if it was a new input) such that

$$h_i(0) = f_a(i)f_b(i)$$

   And gives each party P_j a share $h_i(j)$.
3. **Degree Reduction:** Finally, each party P_j can compute its share of ab, which is simply the point $H(j)$ on the random polynomial $H(x)$ by locally computing the linear combination

$$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j).$$

$$\begin{aligned}
[z] &= [c] + d * [b] + e * [a] + d * e \\
&= [a * b + (x - a) * b + (y - b) * a + (x - a) * (y - b)] \\
&= [ab + bx - ab + ay - ab + xy - bx - ay + ab] \\
&= [x * y]
\end{aligned}$$

With this approach, any MPC protocol based on an LSSS can be cast into the pre-processing model. The pre-processing stage simply calculates many of such triples to be consumed during the online phase. By using this technique, each party must only broadcast two field elements (d and e) in the online phase compared to n field elements in the multiplication protocol by Gennaro et al.

The simplest way to calculate beaver-triples is to simply run the BGW multiplication subprotocol on random inputs but a lot of research has been performed on calculating such triples more efficiently. The best efficiency so far was achieved by Beerliová-Trubíniová and Hirt in 2008 [BH08] who amortised the cost of each triple to a constant number of field elements per party. Furthermore, Schneider and Zohner [SZ13] have proposed more optimisations that can make GMW outperform the current best implementations of Yao.

In general there is no clear answer as to which of these techniques is better. The computational load is fairly similar because most of the computation involves symmetric operations. Yao's approach is very efficient in terms of communication rounds, namely 1, but the communication required to send the GC to the other party is relatively high.

3.3 Active Security

In addition to showing how any function can be calculated in a secure manner, Goldreich, Michali and Wigderson also presented the *GMW Compiler* [GMW87]. Using this compiler, any function that securely computes a function in the passive security model can be translated to a protocol with active security. The intuition of their compiler is to take a passively secure protocol and “force" the potentially malicious adversaries to behave in a semi-honest manner.

Roughly speaking, this is achieved by letting each party *commit* to its inputs and, use *zero-knowledge proofs* to proof that the steps they perform in the protocol are correct without revealing anything.

Even though this approach does work, the large number of zero-knowledge proofs renders it too inefficient to be used in practice. Nevertheless, this work did show that any function can be evaluated securely against active adversaries and sparked the interest for more research on practically feasible actively secure protocols

Looking at work on realising active security in MPC protocols efficiently, there are again two general techniques nowadays. First, there are *cut-and-choose* mechanisms. The idea of cut-and-choose is to essentially run the passively secure protocol multiple times in parallel with the possibility to open an arbitrary amount of runs to verify the correctness. Note that here it is important that the executions can be opened without needing to reveal the private input of a party, since this would cleary violate the privacy requirement. Finally, the unopened run(s) can then be used to perform the secure evaluation. The second mechanism is using *(information-theoretic) MACS*. Here, each message is associated with a message authentication code (MAC) that can be checked for validity to check the calculations of the parties. The cut-and-choose and information-theoretic MAC approaches will be further elaborated in the next Sections 3.3.1 and 3.3.2 respectively.

3.3.1 Cut-and-choose

The basic idea is that one party constructs multiple versions of the same message. Other parties can then check and verify some of them to build trust in the honesty of the creator and use the remaining messages to safely execute the protocol. This technique was originally applied to Garbled Circuits but has also proven to work in secret-sharing based MPC protocols. The idea is that the generator P_1 creates k garbled circuits of the same Boolean circuit and sends them to the evaluator P_2. P_2 will then ask P_1 to reveal the randomnesses used to create a fraction (e.g., $\frac{k}{2}$) of the circuits and checks whether they are correct or not. Note that the revealed randomnesses do not reveal anything on the unopened circuits, who can then be used to perform the secure evaluation. By randomly checking a large enough fraction of the circuits, a malicious generator can only cheat with negligible probability. Furthermore, if the generator takes the majority result of the remaining circuits, correctness of the outcome is almost guaranteed. This is the approach used to obtain the first properly implemented, actively secure version of Yao's protocol in [LP07] and [LPS08].

Even though many improvements and optimisations for specific scenarios have been proposed, using cut-and-choose in this way is not yet efficient enough for practical purposes due to the high number of circuits that need to be generated to achieve an adequate level of security. In 2009, Nielsen and Orlandi introduced *LEGO cut-and-choose* [NO09] to improve the efficiency of cut-and-choose approaches. The idea is to apply cut-and-choose on individual gates instead of the entire circuit.

The approach by Nielsen and Orlandi models functions as circuits of only NAND gates. Furthermore, the output correctness is provided by having a fault-tolerant circuit instead of a majority vote. First, P_1 generates a number of garbled NAND gates on which the two parties perform an (optimised) cut-and-choose protocol. The unopened NAND gates are then randomly assigned into buckets. Gates within a single bucket are combined to act as a fault-tolerant garbled NAND gate that computes the function correctly as long as the majority of the gates in the bucket are correct. These fault-tolerant garbled gates are then connected to form the desired circuit in a process called *soldering*. Finally, P_2 evaluates this single, fault-tolerant circuit to obtain the correct output with great probability.

The main advantage of this idea is its flexibility. This makes it not only suitable for Garbled Circuits but has also proven successful in other paradigms as well as safely setting up correlated randomness with malicious adversaries. Furthermore, this approach is relatively easy and cheap to implement on top of existing protocols.

3.3.2 Information-theoretic MACs

The second technique for realising active security in MPC protocols is based on information-theoretic MACS. The intuition is that such a MAC prevents a corrupted party from lying about their share by "authenticating" their shares. Formally, a *Message Authentication Code (MAC)* is a value that can be used to confirm a message has been created by a certain party who knows the *MAC key* and to detect whether the message has been corrupted. Such a scheme consists of three algorithms:

- $k \leftarrow Gen$: Output a random MAC key k.
- $m \leftarrow MAC_k(x)$: Output the MAC m on x under key k.
- $0/1 \leftarrow Ver(k, m, x)$: Check the MAC on x.

The security requirement is that the verification algorithm should succeed if and only if m is a valid MAC on x, except with negligible probability. Furthermore, it should be nearly impossible to "guess" a correct value given a certain MAC. The information-theoretic aspect of such a MAC means that this security holds even against unbounded adversaries.

There are two main approaches for combining MPC with information-theoretic MACs. First, by authenticating every secret-shared value with unique MACs under the keys of every other party, called *pairwise MACs*. Secondly, there is the approach called *global MACs*. These use only one single, secret shared, MAC on each secret shared value instead of one MAC per pair of parties. Note that even though these MACs are information-theoretically secure, this does not necessarily hold for the protocols that use them for obtaining active security.

The first combination of information-theoretic MACs with MPC was the BDOZ protocol by Bendlin et al. in 2011 [Ben+11]. This protocol makes use of pairwise MACs and tolerates up to $n-1$ active corruptions with a very efficient online phase over arithmetic circuits in the pre-processing model. When a value is secret shared among the parties, now every party P_i receives, next to a share for each secret shared value $[x]$, a set of MACs on his share x_i under the keys of the other parties P_j as well as his own keys to verify the MACs of the other parties' shares. So in the sharing of $[x]$, each party P_i holds:

$$(x_i, (k_{i,j}, MAC_{k_{j,i}}(x_i))_{j \neq i})$$

Here, x_i is the share of $[x]$ for P_i. $k_{i,j}$ is the key P_i can use to verify the share x_j of P_j and $MAC_{k_{j,i}}(x_i)$ a MAC on the share x_i under the key of P_j. The intuition is now that when P_i reveals x_i he can proof to P_j that this share is correct by additionally revealing $MAC_{k_{j,i}}(x_i)$. P_j now knows the corresponding key and can verify the validity of the MAC. If the provided MAC on x_i is invalid, P_j knows that P_i tried to lie about his share.

However, recall that when arithmetic circuits are evaluated, the secret sharing scheme needs to be linear in order to be able to perform arbitrary operations. These same operations need to be performed to obtain the correct MACs on the results of these operations and thus the MAC scheme needs to be linear as well. To see how this is done in the BDOZ protocol, look at the used MAC scheme:

- A key K consists of a random pair $K = (\alpha, \beta) \in \mathbb{Z}_p^2$
- A MAC for a value $a \in \mathbb{Z}_p$ is of the form $MAC_K(a) = \alpha a + \beta \bmod p$

An important property of this MAC scheme is that even if multiple keys with the same α are used, it remains equally hard to guess a correct MAC since the addition of a random β is essentially an (information-theoretically secure) one-time pad. Because of this property, the key generation can be modified such that every key $k_{i,j}$ held by P_i is of the form: $(\alpha_i, \beta_{i,j})$ where α_i is fixed for P_i while a fresh $\beta_{i,j}(x_j)$ is used for every share x_j. Now, the MACs of shares x_j and y_j can be added in the following way:

$$(\alpha_i * x_j + \beta_{i,j}(x_j)) + (\alpha_i * y_j + \beta_{i,j}(y_j)) = \alpha_i * (x_j + y_j) + (\beta_{i,j}(x_j) + \beta_{i,j}(y_j))$$

Next, the entire protocol will be illustrated in the two-party setting but this easily generalises to an arbitrary number of parties. Suppose both parties want to compute a function over their private inputs. To secret share a private input x, the parties additively secret share x. This means we obtain x_1 and x_2 for P_1 and P_2 such that $x_1 + x_2 = x$. Now, each party P_i fixes a certain (public) MAC key α_i. Furthermore, in the sharing of $[x]$, P_1 and P_2 respectively hold x_1, β_1 and m_1; x_2, β_2 and m_2 such that:

- $m_1 = \alpha_2 x_1 + \beta_2 = MAC_{\alpha_2,\beta_2}(x_1)$ (P_1 has the MAC of its share x_1 under P_2's MAC key), and
- $m_2 = \alpha_1 x_2 + \beta_1 = MAC_{\alpha_1,\beta_1}(x_2)$ (P_2 has the MAC of its share x_2 under P_1's MAC key)

Because the MACs are now homomorphic by keeping α_i consistent, the same operations can be performed on the MACs that are performed on the shares. Given sharings of $[x]$ and $[x']$ such that P_1 holds $x_1, x'_1, \beta_1, \beta'_1, MAC_{\alpha_2,\beta_2}(x_1), MAC_{\alpha_2,\beta'_2}(x'_1)$ and analogous for P_2. $[x + x']$ can be calculated in the following way:

$$[x + x'] = [x_1 + x'_1 + x_2 + x'_2]$$

where, due to the homomorphism of the MACs, the correct MAC to authenticate $x_1 + x'_1$ and $x_2 + x'_2$ can be calculated by simply letting each party add its MACs on its shares:

$$MAC_{\alpha_2,\beta_2}(x_1) + MAC_{\alpha_2,\beta_2'}(x_1') = MAC_{\alpha_2,\beta_2+\beta_2'}(x_1 + x_1'), \text{ and}$$
$$MAC_{\alpha_1,\beta_1}(x_2) + MAC_{\alpha_1,\beta_1'}(x_2') = MAC_{\alpha_1,\beta_1+\beta_1'}(x_2 + x_2')$$

Because P_1 knows β_1 and β_1', he can calculate $\beta_1 + \beta_1'$ to obtain the correct key for checking the validity of P_2's MAC and vice versa. Other operations work in a similar way. All the parties can verify the respective MACs on shares which were formed with their own respective keys when opening a share, for example when the entire circuit has been evaluated.

The BDOZ approach generalises to n parties in a straightforward way, for a single sharing $[x]$, the shares of the parties are authenticated under the MAC key of every other party. This generalisation does work but is rather inefficient. Each player needs to have his own key and each of the n shares need to be authenticated with n MACs, so this scales quadratic in n.

To resolve this efficiency issue, a new approach was proposed by Damgård, Pastro, Smart and Zakarias in 2012 [Dam+12]. In their work, often referred to as the SPDZ (or "speeds") protocol, they instead authenticate the secret value itself using a single global key. To prevent forgery of MACs when this key is known to malicious parties, this key is then secret shared as well. This approach called *Global MACs* brings the storage down to constant-sized shares for each party.

Concretely, there is a global MAC key α, that is secret shared to n parties: $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that $\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_n$. In SPDZ, a sharing of x, $[x]$ yields every party P_i a share of the form $(x_i, \gamma(x)_i)$ with $\sum_{i=1}^{n} x_i = x$ and $\sum_{i=1}^{n} \gamma(x)_i = \alpha * x$. In other words, the secret x and the MAC on x, $MAC_\alpha(x) = \gamma(x) = \alpha * x$ are additively secret shared over the parties. Note that the MAC on x is again a linear representation just as in BDOZ and thus supports calculations including multiplications based on multiplication triples. However if parties simply announce their shares, the key α would be revealed and forging fraudulent MACs would become possible.

Instead, the parties only announce their shares of x, $x_0, \ldots, x_{n-1}$ to reveal an unauthenticated "candidate" value for x. Now to check the MAC on x without compromising the secrecy of α, each party P_i commits to $z_i = \gamma(x)_i - x * \alpha_i$. Finally, all parties open their commitments and simply check whether the sum of the committed values is zero, which works because of the following observation:

$$\begin{aligned}\sum_{i=1}^{n} z_i &= \sum_{i=1}^{n} \gamma(x)_i - x * \alpha_i \\ &= \sum_{i=1}^{n} \gamma(x)_i - (\sum_{i=1}^{n} \alpha_i) * x \\ &= \gamma(x) - \alpha * x \\ &= 0\end{aligned}$$

While global MACs provide a big efficiency improvement over pairwise MACs, the pairwise approach still has some relevance because it yields slightly stronger security guarantees. Because each party validates the MACs of every other party, the parties know exactly which parties tried to cheat in the protocol. This might be (very) desirable, for example with the (relatively weak) security with abort setting. With cheater detection in place, the possibility of being exposed as a cheater creates a stronger deterrence to cheating.

3.4 Triple Generation

An important aspect of the above protocols is that they are designed with an offline (or pre-processing) phase and an online phase. The goal of the pre-processing phase is to set up the correlated randomness in the form of multiplication triples to speed up the online phase where the actual calculations are performed. Since these triples are independent of the inputs of the parties, they can be generated beforehand. The most efficient protocols for MPC nowadays exploit this model by moving the majority of the (expensive) computations to this pre-processing phase to have a very efficient online phase. As a consequence, triple generation is an area where major efficiency improvements have been made.

3.4.1 Public-key Cryptography

Both BDOZ [Ben+11] and SPDZ [Dam+12] use *homomorphic encryption* to generate the multiplication triples in the offline phase. The original BDOZ protocol uses an arbitrary additively homomorphic encryption scheme such as Pailler [Pai99] to calculate shares of multiplication triples in an interactive way. Furthermore, zero-knowledge proofs are needed to prevent the parties from lying about their share in the multiplication triples and to proof the correctness of their computation on the ciphertexts. SPDZ uses a different encryption scheme called *somewhat homomorphic encryption (SHE)*. This essentially has the same properties as fully homomorphic encryption except for the fact that a limited number of computations can be performed, e.g., only one multiplication is supported. This allows the parties to multiply their shares in the ciphertexts directly and thus requires only proofs of plaintext knowledge.

To generate triples in this way, SPDZ uses SHE with *distributed key generation* and *distributed decryption* properties. This means there is one single public key for all the parties and an additively secret-shared private key for all the parties. Furthermore, distributed decryption allows the parties to decrypt their individual shares of a certain ciphertext. Now, to obtain multiplication triples of the form $c = ab$, every party first computes an encryption of randomly generated shares a_i and b_i. They broadcast these shares and use homomorphic addition to obtain encryptions of $a = \sum_{i=1}^{n} a_i$ and $b = \sum_{i=1}^{n} b_i$. Next, they calculate the encryption of c by multiplying the two ciphertexts and use distributed decryption to obtain their individual shares of c, c_i. Current state-of-the-art pre-processing protocols based on homomorphic encryption have been presented in Overdrive by Keller et al. in 2018 [KPR18].

3.4.2 Oblivious Transfer

Already in 2012, Nielsen et al. [Nie+12] proposed a different approach for generating the triples described above. Their approach is based on Oblivous Transfer and works in the two-party setting. This approach is simpler and less computationally heavy, but does require more communication between the parties. Their protocol nicknamed TinyOT has a roughly similar online phase to [Ben+11] but works over Boolean circuits instead of arithmetic circuits. In general, expressing the functionality as a Boolean circuit has led to faster protocols because symmetric cryptography can be used whereas arithmetic circuits often require more expensive public-key cryptography. A couple of the state-of-the-art protocols based on Boolean Circuits with active security can be found in [DZ13; LOS14; Dam+17].

On the other hand, arithmetic circuits also have their upsides. Their key advantage is that they allow a calculation to be represented more naturally and thus support operations over integers in an easier and more efficient way. Here, secure linear operations can be performed locally and thus "come for free". In 2016, Keller and Orsini [KOS16] presented MASCOT, where they managed to combine the Oblivious Transfer approach with arithmetic circuits, thus obtaining the benefits of performing calculations over integers directly as well as the benefits of using the much faster OT technique in the pre-processing phase. Using extensions of the OT technique for additional efficiency benefits, they outperform the SPDZ protocol by over 200 times if the network with which the parties communicate with each other is assumed to be relatively fast. Together with the previously mentioned Overdrive protocol [KPR18], this is currently one of the most efficient solutions for actively secure protocols based on secret sharing in the arithmetic circuit model against a *dishonest majority*.

3.5 Alternative Threat Models

So far, most the described protocols have assumed the adversaries to be either passive or active and that every party can be corrupted (for the actively secure protocols). However, these strict assumptions lead to relatively inefficient protocol designs while in reality such tight assumptions might not always be necessary.

3.5.1 Honest Majority

Most of the previously mentioned secure protocols all assume that up to $n - 1$ of the parties may be corrupted. This makes sense for the two-party case, but for the multi-party case this can often be relaxed. If instead the *majority* of the participants are assumed to be honest, a considerable improvement in protocol performance can be made. Furthermore, it is now possible to achieve fairness and information-theoretic security, which was proven to be impossible against a dishonest majority.

The simplest setting for reasoning about *honest majorities* is the case of three-party computation with at most one corruption. In this model and with passive security, some of the most efficient work is that of Araki et al in 2016 [Ara+16] based on a variant of *replicated secret sharing*. Where the original BGW protocol required sending 12 bits per AND gate, this work only required 1 bit of communication per AND gate. An important distinction of this protocol is that each party only has to communicate with one other party and the correlated randomness can be generated without interaction.

One year later, this work was extended by Furukawa et al. [Fur+17] to be secure against active adversaries as well. They followed the approach of pre-processing multiplication triples and used the cut-and-choose mechanisms to guarantee correctness of the triples to be consumed. This approach required 10 bits per AND gate which was later further improved again by Araki et al. [Ara+17] who brought this down to 7 bits and furthermore showed how to improve its practical efficiency by optimising some computationally expensive parts. Current state-of-the-art performance in this area is achieved by ASTRA [Cha+19]. These works have as an additional benefit that they are suitable to be run with inputs represented over arbitrary rings (such as 2^{32} or 2^{64}). These rings are easier to implement and yield good performances in general due to their more natural representation on modern computers.

In the honest majority setting for an arbitrary number of parties, state-of-the-art performance for active security is achieved by Chida et al. [Chi+18]. Here, an optimised 3-party protocol is presented which requires 2 field elements to be communicated per multiplication gate, which increases to 12 field elements for an arbitrary number of parties. Roughly speaking, their protocol is based on secret sharing and works by running an extra copy of the circuit with random values. After walking through the entire circuit, the parties then perform a simple, single check to ensure no cheating attempt took place during the computation phase. Note that this check can be postponed to after the computation phase only because this phase in their secret sharing protocol does not leak information on the private inputs of the parties. Similar results have been achieved by Abspoel et al. [Abs+21] for rings. The main difference between these works is the way in which the check is executed since this has proven to be more difficult for rings compared to fields.

3.5.2 Covert Security

As explained in Subsection 2.3 when the notion of Covert security was introduced, the hope was that by settling for less security, more efficiency could be gained. The key characteristic of covertly secure protocols is to compromise a reasonable amount of security in exchange for less complexity and thus, faster protocols for the real world by introducing a *chance* of catching an active cheater. This chance is called the *deterrence rate* (e.g., $\frac{1}{2}$). Naturally, a deterrence rate of 0 would yield the same security as a passively secure protocol whereas a deterrence rate of 1 results in the security of an actively secure protocol. Furthermore, in most of the covertly secure protocols that have been presented it is possible to exactly identify *who* the cheater is and thus the security guarantees become even stronger than simply active security. In this case, covert security can be seen as a generalisation of passive security and active security *with* cheater detection. By tweaking the deterrence rate, covertly secure protocols are essentially able to cover the entire spectrum between these two notions.

When this idea was proposed by Aumann and Lindell in 2007 [AL07], they also presented the first idea for covert security based on garbled circuits, which essentially boils down to a cut-and-choose approach, where the observation is made that way less circuits need to be checked if it is good enough to catch the cheater with only 50% chance.

This blueprint of replicating computations has proven to be successful for obtaining some of the fastest covertly secure protocols based on garbled circuits in [GMS08; Lin13; AO12; KM15; Hon+19]. The main differences between these protocols do not lie in the cut-and-choose approach but in the construction of the oblivious transfer needed to ensure a party can not abort based on the GC's that are checked later. Furthermore, cut-and-choose has also been proven effective in improving the pre-processing phase of secret sharing based protocols such as SPDZ [Dam+12]. Where normally costly zero-knowledge proofs are necessary to prevent active cheating, these can be replaced with cheap cut-and-choose methods to obtain covert security. A direct implementation comparison of this approach was made by Damgård et al. in 2013 [Dam+13] who reported up to a 40x improvement in the offline phase with a deterrence rate of $\frac{4}{5}$. However, as with most covertly secure protocols, it is possible to dynamically tweak this chance to a certain extent and choose between either more security or more performance.

3.5.3 Public Verifiability

Finally there is the notion of public verifiability in covert security (PVC) introduced by Asharov and Orlandi in 2012 [AO12]. By giving the honest parties a way to prove a certain party has cheated during the protocol execution, the adversary is even more discouraged from cheating. When caught cheating, instead of only losing reputation for one honest party, you now lose your reputation for all the honest parties. Consequently, a lower deterrence rate is sufficient to discourage cheating. Since the value of this deterrence rate largely determines the efficiency of the protocols, having public verifiability leads to even more efficiency gains.

After the introduction by Asharov and Orlandi [AO12], Kolesnikov and Malozemoff [KM15] improved on their work and devised a protocol in which this public verifiability almost comes for free. Both approaches realise public verifiability by simply signing all the messages sent during the protocol execution. However, this means that also the oblivious transfers required at the beginning of the protocol need to be signed. Asharov and Orlandi realised this using *signed-OT* which is based on some expensive public-key operations. Kolesnikov and Malozemoff eliminated a large part of these public-key operations and reached 9-2000x more efficiency (highly dependent on the characteristics of the circuit) compared to state-of-the-art maliciously secure protocols at the time by Lindell [Lin13]. However, the size of the “certificates" for proving cheating are impractically large.

The state-of-the-art PVC protocol by Hong et al. [Hon+19] improves on the previous work by avoiding signed oblivious transfer. With a deterrence rate of $\frac{1}{2}$ they manage to keep the overhead in the 2-party setting down to only 20-40% compapred to state-of-the-art passively secure protocols. Furthermore, their approach is the first to deliver constant-size certificates of cheating. All in all, this would be the best choice for many practical applications of secure two-party computation as long as the high communication costs associated with replicated garbled circuits are manageable for the application.

To obtain covert security in the protocol mentioned above, parties P_1 and P_2 run k instances of a garbled-circuit protocol secure in the passive setting. Of these instances, $k-1$ will be checked by the parties while the last one is used to obtain the actual output. Now, to verify honest behaviour, the executions are made deterministic by letting P_1 seed all the instances with a certain randomness. Using OT, P_2 obtains the randomnesses for $k-1$ of these instances without P_1 knowing which instances are being checked. Because P_1's behaviour is entirely dependent on P_2's messages and the seeds, it is possible for P_2 to verify the correctness of all the messages sent by P_1. To generate a publicly verifiable certificate, we let P_1 sign the transcript of each instance as well as the OT protocol. If P_1 now cheats in any of the instances of the protocol, P_2 can prove this using the certificate, his own view and the obtained randomness for the execution.

3.6 State-of-the-art

An overview of the protocols that perform best for various scenarios can be found in table 3.1. As we have seen in this chapter, there are many different protocols who have their own area in which they shine. In general when looking to reduce the amount of communication rounds, garbled circuit based approaches such as [BMR90] (passive security) and approaches based on TinyOT[Nie+12] (active security) all achieve a constant amount of communication rounds. In the multi-party setting the work of Hazay et al. [HSS17] obtain actively secure MPC with a constant number of rounds based on a combination of TinyOT and

Reference	Technique	Parties	Adversary (#corr)
[BMR90]	Garbled Circuits	n	Passive ($n-1$)
[Bea91]	Secret Sharing	n	Passive ($n-1$)
[KPR18]	Secret Sharing	n	Active ($n-1$)
[HSS17]	Garbled Circuits	n	Active ($n-1$)
[Hon+19]	Garbled Circuits	2	Covert (1)
[Dam+13]	Secret Sharing	n	Covert ($n-1$)
[Cha+19]	(Replicated) SS (Rings)	3	Active (1)
[Chi+18]	(Replicated) SS	n	Active ($< n/2$)
[Abs+21]	(Replicated) SS (Rings)	n	Active ($< n/2$)

Table 3.1: Overview of the state-of-the-art MPC protocols in terms of number of parties and adversary characteristics.

authenticated garbling. Simply speaking, authenticated garbling is a technique which combines garbled circuits with information-theoretic MACs to obtain actively secure garbled circuits. In contrast to garbled circuit approaches, secret-sharing based protocols typically require less bandwidth. For passive security, good performance for secret sharing was in the landmark paper of [GMW87] in the two-party setting. In the multi-party setting, Beaver's circuit evaluation approach [Bea91] is seen as the state-of-the-art for passive security. Between garbled circuits and secret sharing, there is not much difference in terms of computation but in secret-sharing based protocols, we have seen the advantage of a pre-processing phase as demonstrated by Beaver. By moving the majority of the complexity to an input-independent pre-processing phase, we can very quickly perform the actual computations once the inputs are available. State-of-the art performance in this category has been achieved in MASCOT [KOS16] and the subsequent protocols in Overdrive [KPR18] (LowGear and HighGear). Here, most of the improvements have come from improvements in securely generating multiplication triples. The two main techniques for this have been some form of homomorphic encryption and oblivious transfer.

Furthermore, we have seen the alternative notion of covert security. Here, state-of-the-art performance in this setting was achieved in [Dam+13] for protocols based on secret sharing. In this case, we can optimise the pre-processing phase using cheaper cut-and-choose approaches while running the same online phase. For garbled circuits, a state-of-the-art protocol with covert security was presented in [Hon+19] for the two-party setting, who also achieve public verifiability.

Finally, the advantages of the existence of an honest majority have been shown. In this case, state-of-the-art performance for the specific three-party case with one corruption has been achieved by using replicated secret sharing for both passive and active security with ASTRA [Cha+19], which is based on the two works of Araki et al. earlier. These works have the additional advantage that they support the representation of inputs as ring elements (such as the ring over 2^{32} or 2^{64}, leading to very fast protocols in practice. In the generic honest-majority setting over fields, state-of-the-art performance is achieved in [Chi+18]. For the same setting over rings, their performance is matched by Abspoel et al. [Abs+21].

Chapter 4

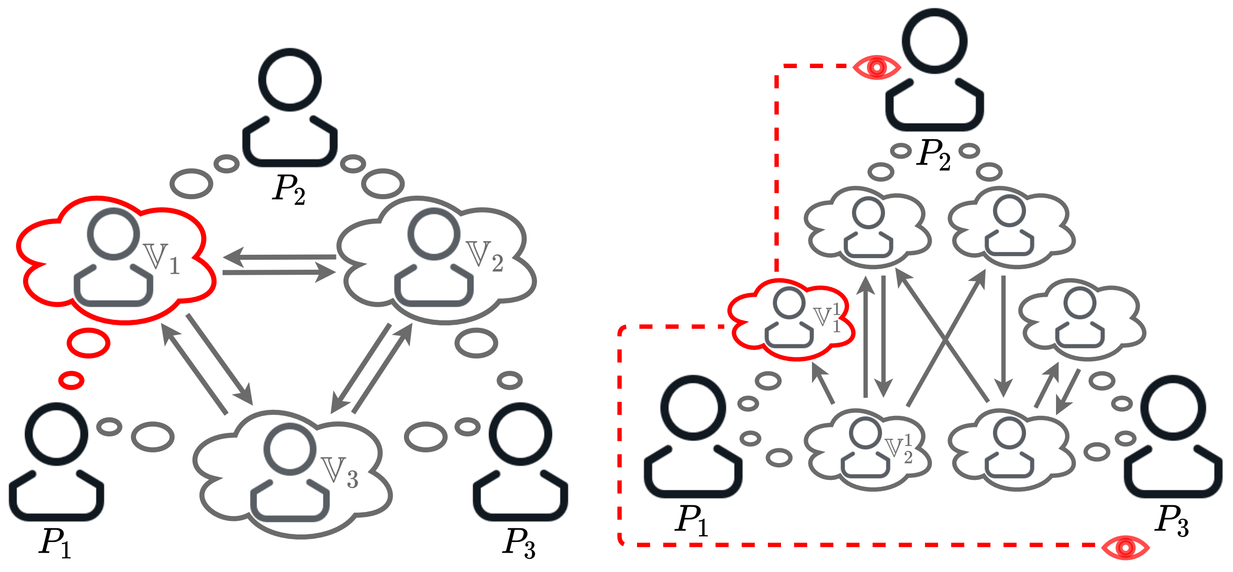
MPC Compilers

When looking at the state-of-the-art concrete protocols in Chapter 3, they are based on exploiting the characteristics of a certain technique. For example it was shown how state-of-the-art performance for active security and *n* parties could be realised using secret sharing and pre-processing with OT. Furthermore, for obtaining covert security (with public verifiability), replicating garbled circuits has proven to be very effective. However, the fact that these protocols get their performance from exploiting the underlying technique means that if in the future a very efficient way to (e.g.) evaluate neural networks were to be found in the passive security setting, there is no generic approach to make this active or covert secure.

This is where the idea of general transformations or *compilers* comes in. Actually, the first real MPC compiler has already been explained in the form of the GMW compiler. By letting parties commit to their inputs and “proving" the correctness of the messages communicated, they transform arbitrary passively secure protocols into actively secure ones. While this was a fundamental feasibility result, the resulting protocols are far from practical. Next to efficiency issues, their approach does not make *black-box* use of the underlying protocol, meaning the semi-honest protocol needs to adhere to a set of properties for the GMW compiler to work. In contrast, most works on such MPC compilers nowadays make use of replicating calculations, like the cut-and-choose approach we have seen in the previous Chapter 3. Another related idea is known as *MPC in the head* or *player virtualisation*. This idea was first introduced by Ishai et al. in 2007 [Ish+07; IPS08] in the context of zero-knowledge proofs and later optimised by Lindell et al. [LOP11]. By treating the passively secure protocols in a *black-box* manner, they were able to obtain the first generic transformation in the form of a compiler.

4.1 Player Virtualisation

Even though the player virtualisation paradigm is fairly straightforward, it has led to many promising insights for the design of generic compilers for increasing the level of security of MPC protocols. The basic intuition of this paradigm is to let the parties “imagine" or “simulate" a number of virtual parties. These parties then execute the weakly secure protocol on behalf of the real parties to compute the desired functionality. As mentioned previously, the first work to introduce this paradigm for constructing a generic compiler for obtaining active was by Ishai et al. in 2007 [Ish+07; IPS08] and is known as the *IPS Compiler*. In this compiler, the real players simulate a set of virtual parties. Now, to achieve active security from a passive secure protocol, these virtual parties are then checked for honest behaviour by other real



(a) Many-to-one virtualisation (b) One-to-many virtualisation

Figure 4.1: Two player virtualisation strategies

parties via a construction known as *watchlists*. The idea is that each party chooses the randomnesses it will use to execute the semi-honest sub-protocols for each virtual party. Using oblivious transfer, each other client then obtains a random subset of these random seeds with which they can check the behaviour of this party for a subset of the virtual parties. Their protocol is black-box with respect to the passively secure inner protocol but not yet with respect to an actively secure outer protocol.

Roughly speaking, two fundamentally different approaches for applying player virtualisation to increase the security level of an MPC protocol can be distinguished. A schematic illustration of these two approaches can be found in figure 4.1.

The first approach in figure 4.1a is to let multiple parties simulate the *same* virtual party, which we call *many-to-one* virtualisation. Increased security in this approach is obtained by the original parties knowing what the virtual party should be sending and verifying that the other real party, simulating the same virtual party, follows the protocol honestly (for this virtual party). In figure 4.1a, we can for example see P_1 checking the behaviour of P_2 in the virtual party $\mathbb{V}_1$. Since P_1 simulates the same virtual party, he knows exactly what P_2 should be sending and can detect faulty behaviour. This approach is generally taken when aiming for active security since *every* message communicated by the real party can be verified. However, letting all the parties check all the messages of all the other parties results in the need for one virtual party per pair of real players, which incurs a quadratic complexity in terms of virtual parties and leads to a large computation and communication overhead.

The second approach is depicted in figure 4.1b, where the idea is to let one real party simulate multiple virtual parties or *one-to-many* virtualisation. In this approach, increased security is usually obtained by letting the real parties check a subset of the virtual parties of the other players. This is a more efficient approach to the previous one since the number of virtual parties that need to be simulated scales linearly in the number of parties. However, this approach does not guarantee to catch malicious behaviour since not all virtual parties can be checked to maintain privacy of the inputs. This approach is thus mainly applied when aiming to obtain covert security. This approach is however rather flexible as the number of virtual parties can be scaled up or down in order to facilitate more checks and an increase in the deterrence rate. In figure 4.1b, we can see P_2 and P_3 checking the behaviour of virtual party $\mathbb{V}_1^1$ of P_1. If P_1 wants to cheat, he must cheat in at least one of its virtual parties and as a result, P_2 and P_3 have a 50% probability to detect cheating behaviour in this example.

4.1.1 Covert Security

The idea of replicating calculations to obtain a compiler for realising *covert security* was first introduced by Damgård, Geisler and Nielsen in 2010 [DGN10], in the form of a cut-and-choose approach. Their approach works for any secret sharing based protocol by performing the calculations twice, once on the actual shares and once on a set of *dummy shares* which can be used to verify correctness later. With an overhead of two times the passive protocol, they reached a deterrence rate of $\frac{1}{4}$. However, their approach only works for an honest-majority.

The previously introduced compiler by Lindell, Oxman and Pinkas [LOP11] (based on the player virtualisation paradigm) *does* work in the dishonest majority setting for obtaining covertly secure protocols. However, the communication- and round complexities of the compiler depend on the complexity of both the inner and the outer protocol. This means the resulting protocols either have a lot of rounds or a large communication overhead and do not scale well for a larger number of parties. Furthermore, the resulting protocols are not publicly verifiable.

4.1.2 Active Security

In 2018, Damgård, Orlandi and Simkin [DOS18] presented one of the first compilers for *active* security by using player virtualisation. In their work, they use the many-to-one player virtualisation from figure 4.1a. If every virtual party is at least emulated by one honest party, this party can detect faulty behaviour by other parties emulating the same virtual party and abort if necessary to ensure correctness of the output. Furthermore, if the corrupt parties do not emulate more virtual parties than the corruption threshold of the original protocol Π_f, privacy is also preserved. A rough overview of the steps in the compiler for active security, $\texttt{COMP}_{\text{ACTIVE}}$ can be found in compiler $\texttt{COMP}_{\text{ACTIVE}}$. Note that a lot of (non-trivial) details have been omitted here in order to focus on the core ideas behind the compiler. For example, the way in which the parties agree on the inputs in a secure manner and the augmented protocol $\Pi_{f'}$ that is actually executed by the virtual parties. For a full, formal description of the compiler we refer to the original work [DOS18].

Note that if more than one active corruption would be present this transformation would not work anymore since the adversary may corrupt two parties executing the same virtual party and get away with cheating in that virtual party. To support more corruptions, the number of parties executing the same virtual parties would need to be increased at the cost of efficiency. Concretely, given n parties with t corruptions, each virtual party would need to be simulated by at least $t + 1$ parties, meaning each party will participate in the simulation of $t + 1$ virtual parties.

Compiler $\texttt{COMP}_{\text{ACTIVE}}$

This compiler transforms a passively secure protocol Π_f for n parties into an actively secure protocol for n parties with one active corruption.

1. Start with a passively secure n-party protocol ($n \geq 3$).
2. Instead of letting the real parties execute the passively secure protocol directly, let every pair of real parties simulate virtual parties that will compute the desired functionality using the passively secure protocol.
3. For $1 \leq I \leq n$, real party P_i and P_{i+1} will simulate $\mathbb{V}_i$ (where wrap-around is assumed, meaning $P_{n+1} = P_1$). To obtain the inputs for the virtual parties, each party P_i uses additive secret sharing to share its input among all virtual parties. Each virtual party $\mathbb{V}_i$ now has as input to the protocol one share of every original input: $(x_1^i, \ldots, x_n^i)$. For the random tapes for the virtual parties, the real parties invoke an ideal functionality that sends to random tape to be used to the corresponding real parties.
4. The virtual parties will now execute a related passively secure protocol $\Pi_{f'}$ in the following way:
    - Whenever $\mathbb{V}_i$ sends a message to $\mathbb{V}_j$, this is achieved by letting P_i and P_{i+1} both send the same message to P_j and P_{j+1}. Here, observe that if either P_i or P_{i+1} would be trying to cheat at this point, this would result in a mismatch between the two messages.
    - To achieve active security, the receiving party $\mathbb{V}_j$ which consists of P_j and P_{j+1} locally check whether the received messages are identical and if not, notify all the parties and abort.
5. Finally, if none of the parties aborted during the protocol execution, each real party outputs whatever $\Pi_{f'}$ outputs.

This transformation works well for a small number of parties, but for larger numbers of parties, the linear computational overhead as a result of every party needing to simulate $t + 1$ virtual parties becomes large. Next to that, this approach incurs a multiplicative overhead in terms of communication since each party simulating a virtual party needs to send a message to every real party simulating another virtual party whenever a message needs to be exchanged between these virtual parties. If in the original passively secure protocol l messages are sent during the execution, the protocol resulting from this compiler sends roughly $O(l * t^2)$ messages. Finally, their approach is based on sacrificing the number of corruptions the protocol can tolerate. For a larger number of parties, the amount of corruptions the resulting protocol can tolerate drops to around $\sqrt{n}$. All in all, this approach is not practical for a large number of parties.

4.1.2.1 Weak Privacy

As a solution to the large amount of redundancy introduced in [DOS18], a followup work was presented in 2020 by Eerikson et al. [Eer+20] in the specific 3-party setting with at most 1 corruption. By making slightly stronger assumptions on the passively secure protocols, they are able to produce protocols roughly twice as efficient in terms of communication overhead compared to [DOS18].

Their improvement over the previous compiler comes from the observation that for many concrete protocols, for example those following the share-compute-reveal paradigm such as [BGW88] and [Bea91], no information on the private inputs can be learned by an active adversary up until the very last step in the protocol, the reveal phase. This slightly stronger assumption is known as *active security with "weak privacy"* [Gen+14] or "active privacy" [PL15]. This means that any misbehaviour during the computational phase of the protocol execution may break the *correctness* of the outcome but cannot compromise the *privacy* of the inputs.

Because of this observation, not every message needs to be checked during the computation phase and thus it suffices to only perform one check after the computation phase. Because of this, not every message needs to be checked immediately anymore, and thus not every message needs to be directly communicated redundantly anymore. The idea of their compiler is now to elect one real party for each virtual party to be the "brain" of that virtual party which sends all the messages on behalf of that virtual party to all the real parties. The other real parties for that virtual party, the "pinkies" still receive these messages from the brains and can locally follow the protocol execution. Compared to the compiler of [DOS18], this leads to a change from $O(l * t^2)$ to $O(l * t + t^2)$ in terms of communication complexity.

Now, instead of validating every message separately as soon as they are sent, all the parties perform only a single check to validate the correctness of all the previous messages before the reveal phase. This can be done very efficiently by merely checking the consistency of the *hashes* of the protocol transcripts. Now, if any of the brains were be to misbehave during the computation phase of the protocol, at least one of the messages it has sent during the computation phase must be different from the view of at least one of the pinkies and thus the hashes must mismatch.

4.2 Covert Security with Public Verifiability

Compilers for covert security *with* public verifiability were only presented recently. In 2020, Damgård, Orlandi and Simkin presented the first compiler that transforms arbitrary, passively secure two-party protocols into two-party covertly secure protocols *with* public verifiability [DOS20]. Furthermore, their compiler provides a *black-box* transformation, meaning that it does not need any assumptions on the inner workings of the passively secure protocols it transforms. As a result, it will work for any protocol, regardless of the technique it is based on. This means it provides a generic transformation for any protocol based on, e.g., secret-sharing, garbled circuits or homomorphic encryption as well as any future insights that might arise.

While Damgård, Orlandi and Simkin sketched how their two-party compiler works in the multi-party setting, the first fully described multi-party compilers for covert security with public verifiability were presented in 2021 by Faust et al. [Fau+21] and concurrently by Scholl, Simkin and Siniscalchi [SSS21], which is unpublished at the time of writing.

All these three works have made a distinction between compilers for input-independent protocols and a separate compiler for input-dependent protocols. The input-independent compilers are all based on the cut-and-choose strategy, while the input-dependent protocols follow the one-to-many player virtualisation strategy. This distinction is made since a lot of state-of-the-art MPC protocols consist of a pre-processing phase without private inputs followed by an online phase. The input-independent compilers are conceptually simpler and more efficient since there are no private inputs to be kept secret.

In [DOS20], it is proven that a protocol which constitutes of a covertly secure offline phase with public verifiability and an actively secure online phase constitutes an overall protocol with covert security and public verifiability. Since the online phase of a lot of protocols in the pre-processing model such as SPDZ-like protocols is actually not that complex, the approach to optimize such protocols is *not* to apply the compilers to the (input-dependent) online phase of a passively secure version of SPDZ. Instead, the go-to approach is to combine the actively secure online phase of SPDZ with our compiler applied to a passively secure offline protocol. Since the offline phase contributes the vast majority of the complexity of such protocols, a lot of improvement on the overall protocol can be made by applying the input-independent compiler to these offline protocols.

4.2.1 Input-independent Compilers

As explained earlier, the input-independent compilers all follow the cut-and-choose strategy to transform passively secure protocols into covertly secure protocols with public verifiability. Roughly speaking, all of these compilers follow the same blueprint for producing covertly secure two-party and multi-party protocols in respectively [DOS20] and [Fau+21; SSS21]:

1. First, the parties are bound to the randomness seeds they should use for each of the k executions of the passively secure protocol.
2. Next, the parties engage in k parallel executions of the passively secure protocol where each party uses the agreed randomness from the first step. For each execution, party P_i ends up with the transcript $\texttt{trans}^i_j$ of execution j.
3. After the k executions, the parties randomly choose a subset of t of these executions, $t < k$ to verify the behaviour of the other party/ parties.
4. The parties announce the randomness they have used in the protocol runs in $[t]$ to the other parties.
5. Knowing these randomnesses, the behaviour of the parties becomes deterministic and thus the parties can simulate the behaviour of all the other parties in the passively secure protocol executions. Now, they compare the resulting transcript $\texttt{trans'}^i_j$ to the actual transcript $\texttt{trans}^i_j$ of the jth execution for party P_i.
6. If there is a mismatch between $\texttt{trans}^i_j$ and $\texttt{trans'}^i_j$, the party finds the first party which has sent an incorrect message and generates a certificate to blame this party. The resulting deterrence rate is $\frac{t}{k}$.

The idea of "derandomizing" the protocol runs is similar to the concrete protocol by Hong et al. [Hon+19]. However, the resulting protocols are not yet publicly verifiable. To achieve this, the parties *sign* the (messages in the) transcript (see Section 2.1.2 for more details). This way, the parties can be held accountable for the messages they have sent during the executions. If a party is now found guilty of cheating, the other parties can publish the signed transcript of the protocol execution of the cheating party, which anyone can verify.

It is essential for every protocol with security and public verifiability against covert adversaries that the adversary is not able to abort and prevent the generation of a certificate after it has learned what executions are going to be opened. The intuitive approach described above will therefore not work since the adversary can simply stop responding after it has learned which executions will be opened in step 3. The main differences between the works by [DOS20] and [Fau+21; SSS21] lie in the way how the t executions are chosen and how this so-called *selective abort* is prevented. Here, [DOS18] use *watchlists* while [Fau+21; SSS21] use *time-lock puzzles*.

4.2.1.1 Watchlists

In [DOS20], the selective abort is prevented by letting the parties select individually which executions they want to open and asking the other party in an *oblivious* way to open these executions by revealing the used randomness already before the protocol executions. Because the opening information is transferred in an oblivious way, the adversary does not know which of the executions are being checked and thus the adversary would need to make the choice to abort the protocol without information of the checked executions.

This *watchlist* approach, works well in the two-party case but falls short for multi-party protocols. For the resulting covert protocol to work, we need to be guaranteed that at least one execution remains unopened, which can then be used as the output of the protocol. With only two parties, we can guarantee this by simply letting the parties choose a watchlist of size $< \frac{k}{2}$. However, with multiple parties independently choosing a subset of the protocol runs to open, the size of the watchlist is upper bounded by $\frac{t-1}{n}$, which results in a low deterrence rate. The other option is to increase the deterrence rate by choosing a larger watchlist size and running the protocol several times until we get lucky and there is one execution which is unchecked by all the parties. However, this makes it very easy for an adversary to perform a *denial-of-service* attack; by simply choosing a very large watchlist, the adversary can require this protocol to be run many times.

4.2.1.2 Time-lock Puzzles

Instead of the probablistic approach needed to obtain large deterrence rates with the watchlist approach, it would be better if we could somehow guarantee one unopened execution by performing a coin-toss to select *one* execution which we leave unopened to maximise the deterrence rate by opening the other $k-1$ executions. To do this successfully in a compiler, we need to guarantee that if an adversary sees the result of this coin-toss, it is already too late to abort the protocol. In other words, the honest parties need to have all the information to produce a certificate *before* the result of the coin-toss is revealed to all the parties. This way, they can create a certificate even if the adversary stops responding.

The works of [Fau+21] and [SSS21] both solve this problem using so-called *time-lock puzzles*. Time-lock puzzles are a technique to encrypt something *into* the future such that we are guaranteed that the message is kept secret at least for some predefined time. The shared coin as well as the randomness seeds according to this shared coin are then locked in such a time-lock puzzle. In [Fau+21] This puzzle is then signed by all the parties as evidence. If the adversary has to sign the puzzle before it could possibly have opened the puzzle, the selective abort is prevented. Next, the shared coin is revealed and all the parties announce their seeds according to this coin. If the adversary stops responding at this point, the honest parties can solve the puzzle to reveal the randomness used by the adversary. By leaving the execution corresponding to the shared coin toss unopened, all but one of the total of k executions can be checked. This way, a deterrence rate of $\frac{k-1}{k}$ can be consistently achieved which greatly decreases the complexity for achieving high deterrence rates compared to the watchlist approach. A deeper analysis of this behaviour can be found in Chapter 7.

For creating the time-lock puzzle, the compiler uses a general-purpose MPC protocol with active security where each party inputs their own coin toss and the seeds to open *all* the protocol executions. The ideal functionality which is implemented by the protocol then calculates the shared coin toss, generates a puzzle containing the opening information of all the parties for the checked executions according to the shared coin toss and sends this to all the parties.

The adversary can now decide to sign the puzzle or abort without knowing which execution will be checked. After everyone signed the puzle, the shared coin toss is revealed to all the parties. Now, two cases are distinguished; 1. everyone honestly shares the opening information for the checked executions to the other parties or 2. the adversary does not share his opening information, in which case the honest parties can solve the time-lock puzzle and still obtain enough information to produce a certificate proving malicious behaviour. The first case is referred to as the *optimistic case* while the latter is the *pessimistic case*.

The time-lock approach thus needs a general-purpose MPC protocol with active security for the puzzle generation, which are known to be complex. Furthermore, the time-lock puzzle approach also requires a trusted setup for producing the public parameters and a base puzzle. This trusted setup will most likely also require an actively secure MPC protocol in practice.

4.2.2 Input-dependent Compilers

The three works from the previous section all follow the one-to-many player virtualisation strategy in a similar manner for input-dependent protocols as well. Instead of letting the parties directly execute the passively secure protocol k times in parallel, they all simulate k virtual parties who execute a related kn-party protocol just once. Intuitively, the input-dependent compilers work in the following steps:

1. Suppose there are parties $P_1, P_2, \dots, P_n$ who wish to compute some function $f(x^1, x^2, \dots, x^n) = (y_1, y_2, \dots, y_n)$. Here, P_i holds private input x^i and receives output y_i. Furthermore, they have a kn-party protocol which calculates a related functionality.

2. Now, party P_i imagines k virtual parties $\mathbb{V}_1^i, \mathbb{V}_2^i, \dots, \mathbb{V}_k^i$. This results in a total of $n * k$ virtual parties who will execute the nk-party protocol.

3. P_i splits his private input x^i into shares $x_1^i, x_2^i, \dots, x_k^i$ using the k-out-of-k additive secret sharing scheme. These shares will be used as the inputs of the virtual parties belonging to P_i. Furthermore, the parties agree on the k random seeds that will be used by the k virtual parties.

4. The kn virtual parties now execute a related, passively secure protocol. Each virtual party inputs its share of the original input and receives an additively secret-shared output. Here, the real parties communicate on behalf of their virtual parties. Each real party P_i ends up with a transcript $\texttt{trans}_j^i$ of each virtual party $\mathbb{V}_j^i$ and k output shares.

5. The parties randomly choose a subset of $t < k$ of the virtual parties which they would like to check. This subset of virtual parties will be checked for every real party.
6. The parties announce the randomness and inputs used by each virtual party in $[t]$ to the other parties.
7. To verify the behaviour of a virtual party, the real parties can simulate the nk-party protocol using the randomness, input and messages sent/ received by that virtual party to obtain the simulated transcript $\texttt{trans'}^i_j$.
8. If there is a mismatch between $\texttt{trans}^i_j$ and $\texttt{trans'}^i_j$, the party produces a certificate to blame the corresponding real party.
9. If no cheating behaviour is detected, the virtual parties end up with an k-out-of-k additive secret sharing of the outputs of the real parties.

The structure of this blueprint is similar to the input-independent compilers but the additional need to keep the inputs private yields some extra challenges. To hide the input of the real party, we use additive secret sharing to hide the input and use these shares as the inputs of the virtual parties in a related protocol which takes the t-out-of-t secret sharing of the inputs of all the real parties and outputs a t-out-of-t secret sharing of each party's output. Furthermore, to derandomise and verify the behaviour of the virtual parties, we now need to reveal the input used by the virtual party in addition to the randomness seed and the incoming messages of this virtual party. To be able to correctly simulate the protocol execution of a virtual party, we therefore require all the messages being sent to all the parties, including messages sent from one virtual party to another virtual party belonging to the same real party.

4.2.2.1 Watchlists

The similarities and differences of the watchlist approach and the timelock approaches in the input-dependent setting are very similar compared to the input-independent compilers. Note that in [DOS20] only a rough sketch for extending their two-party compiler to the multi-party case is presented. If we want to tolerate a fraction of c corruptions, this results in a total of cn corrupted parties. In the watchlist approach, for every pair of real parties (P_i, P_j) with $i \neq j$, P_i checks exactly one virtual party of P_j, chosen uniformly at random. To guarantee privacy of the inputs of the real parties, every real party simulates $cn + 1$ virtual parties, which ensures the corrupted parties together see at most cn out of the $cn + 1$ views of the virtual parties of a real party, which is not enough to reconstruct the actual input of the real party.

In total, an adversary has control over $cn * (cn + 1)$ virtual parties and gets the views of $(1 - c)n * cn$ other virtual parties. To ensure the privacy of the inputs of the real parties is preserved, we thus require the protocol executed by the virtual parties to tolerate a total number of corruptions:

$$cn * (cn + 1) + (1 - c)n * cn = cn^2 + cn$$

To calculate the deterrence rate of this approach, observe that a cheating party must cheat in *at least* one of its $cn + 1$ virtual parties. The probability that *at least one* of the honest parties checks this virtual party can be calculated as the inverse of the probability that *none* of the honest parties check this virtual party. As stated in the original work, this results in a deterrence rate of

$$\left(1 - \frac{1}{cn+1}\right)^{(1-c)n}$$

Using binomial approximation, we get that the deterrence rate of the watchlist approach is $\epsilon \approx \frac{1-c}{c}$.

4.2.2.2 Time-lock Puzzles

Time-lock puzzles are applied in a very similar fashion to the input-independent compiler. Now, instead of inputting only the seeds and a coinflip to the puzzle generation functionality, the parties also input the inputs used by their virtual parties. Furthermore, we can reach the same deterrence rates as the input-independent compiler by simply selecting $k-1$ out of the k virtual parties to open. Since the inputs of the real party are shared using k-out-of-k additive secret sharing, we can safely open all but one of the virtual parties. This way, we can dynamically tweak the compiler to be more secure or more efficient by simply changing the number of virtual parties. With the watchlist approach, the deterrence rate is more of a consequence of the number of corruptions that need to be tolerated. Finally, since all the parties are checking the *same* virtual parties of a real party, the kn-party protocol now needs to tolerate only $kn-1$ passive corruptions instead of the $cn^2 + cn$ corruptions in the watchlist approach.

Chapter 5

New Design for PVC Compilers

While the time-lock solutions from 2021 in [Fau+21; SSS21] have significantly improved the earlier watch-list approach from 2020 [DOS20], there are still a few shortcomings with respect to the practicality of these protocols. First of all, these compilers still make use of general-purpose actively secure MPC for constructing the time-lock puzzles. The construction and evaluation of known instantiations of such time-lock puzzles requires a number of public-key operations which are known to be expensive in actively secure MPC. These time-lock puzzles *always* need to be solved by the parties in case any cheating happened in order to be able to produce a certificate.

Furthermore, the time-lock based approaches heavily relies on the assumption of synchronous communication. In the theoretical setting, it is easy to define a timeout for a communication round, but this does not work on the networks that we can use in practice such as the internet. In the internet, packets sent from one machine to another machine need to go via a number of hops (such as routers). At each of these hops, the packets may be delayed or even lost completely, after which the packet has to be retransmitted. Because of this, it is very hard to predict *when* a packet should be arriving at the destination and thus it is very hard to define a timeout after which we can be certain we can proceed to a new communication round. Therefore, the security of MPC protocols in practice is preferably not reliant on the (unrealistic) assumption of a synchronous communication network.

However, looking at the asynchronous communication model, it is impossible to distinguish whether a party or the network is just being slow, or a party is acting maliciously. Because of this, the adversary must not be able to compute additional information and gain an advantage before sending its next message. However, time-lock constructions can not guarantee this. In the time-lock approaches of [Fau+21; SSS21], the time needed to solve the puzzles is set rather low because the honest parties must always solve the time-lock puzzle in case of any malicious behaviour to be able to produce a certificate. As a result, it seems very much possible for a malicious party to solve this time-lock puzzle in a reasonable amount of time and see which executions are going to be checked before signing the data. With this power, the adversary can prevent the creation of a certificate in case he does not like which executions are being checked. This completely breaks the definition of publicly verifiable covert security. The other parties can not differentiate between a malicious party solving a time-lock puzzle or an honest party who is simply slow.

In this chapter we present a new solution to prevent the selective abort attack in the shared coin-toss approach based on *publicly verifiable secret sharing*. The compiler treats the passively secure protocol in a black-box manner and is thus suitable to be applied to a large number of known or future MPC protocols. Furthermore, the subprotocols introduced by the compiler are suitable to be run in an asynchronous manner. Therefore, if the passively secure protocol is asynchronous, the resulting covertly secure protocol will also be asynchronous which makes it very attractive for practical use-cases. By performing more or fewer k parallel executions, we can dynamically choose between more security or more efficiency. Here, k repetitions results in a deterrence rate ϵ of $\frac{k-1}{k}$.

We will first present a compiler for passively secure protocols *without* private inputs such as the important class of pre-processing protocols. We start with a technical overview of our construction, after which we will look at the building blocks used by our compiler. After that, a compiler for input-independent protocols $\texttt{COMP}_{\text{PVC}} = (\Pi_{\text{comp}}, \Pi_{\text{open}}, \Pi_{\text{reconstruct}})$ will be presented together with the accompanying `Blame` and `Judge` algorithms. Finally, we present a security analysis of this compiler. In the last section of this chapter we explain how we can apply player virtualisation to design a compiler that works for input-dependent protocols as well.

5.1 Technical Overview

As a general blueprint for the previous works on compilers for covert security, some form of cut-and-choose is used. In general, a passively secure base protocol is executed k times in parallel after which $t < k$ executions are opened and checked by all the parties. For input-independent protocols such as triple generation, the unopened execution(s) can then be taken as the output of the protocol.

In this work, opening executions is done by committing the parties to k seeds that will be used by the parties to derive their internal randomness in the k parallel executions of the protocol. After the k executions are finished, the parties derandomise $k-1$ of these executions by simply publishing the seed they have used in these executions. To pick the $k-1$ executions to check, the parties engage in a joint coin-toss protocol to select the one execution that should remain hidden. Using the seeds of all the parties for the selected executions, the parties can *simulate* the executions and compare the transcripts to the transcripts obtained from the actual execution. In case of a mismatch between these transcripts, they can produce a *certificate* which proves the cheating to *anyone*.

To be able to produce a certificate, the parties are made accountable for their messages by letting each party *sign* all the messages it is sending in the protocol executions. If it is later detected that a party has cheated during any of the executions, anyone can verify that the adversary did sign the malicious messages. As explained before, it is important that the adversary can not prevent the creation of a certificate in case they do not like the executions that are being checked. Therefore, the naive joint coin-toss approach illustrated above will not work since there the adversary can simply abort after seeing the outcome of the coin-toss. Instead, we first let all the parties use a publicly verifiable secret sharing scheme (PVSS) as explained in Section 2.2.4 to share their opening information to all the parties. After this, the parties run the joint coin-toss to select which executions are going to be opened. The parties then simply publish their seeds according to the coin-toss. If an adversary stops responding at this point, the parties already have the shares of the adversary and can reconstruct the seed anyways.

The PVSS scheme guarantees that the shares distributed by the adversary will reconstruct a well-defined seed, to which the adversary is also bound by the published proof. In case the verification of the shares fails for some seed of some party, the parties abort and anyone can verify that the adversary tried to cheat in the distribution phase. This verification can be performed without interaction with the distributor or any of the other parties. Therefore, we can include this in a publicly verifiable certificate. Furthermore, the proof of correct decryption in the PVSS scheme ensures that an adversary can not 'incriminate' an honest party by publishing a different share of a seed of the honest party in case it needs to be reconstructed. Finally, since *anyone* can verify the PVSS scheme, it is not possible for an adversary to somehow claim that an honest party is malicious.

In the work on publicly verifiable secret sharing, some form of public communication is assumed, meaning that all the parties see the same messages that are distributed and decrypted. Hypothetically, one can view this as some public bulletin board where all the parties post their messages to. For this work, we emulate this public bulletin board by means of a secure broadcast. To broadcast a message m securely, the sender sends m to all the other parties first. After that, each party hashes the received message m and sends this to all the other parties. By pairwise comparing the received hashes, the parties can be guaranteed that they received the same message m.

It is important to keep the seeds locked until after the coin-toss. This means the adversary should not get enough information (shares) to reconstruct the seeds on its own but should always require at least one share of an honest party. Therefore, our approach is based on the assumption that the majority of the participants are honest.

As explained earlier, the time-lock approach breaks in case of asynchronous communication. Our approach is expected to be suitable to be ran in the asynchronous model since the seeds are not locked by timing assumptions but by means of secret sharing. Another important property of asynchronous MPC protocols is to be able to continue even if some of the parties stop responding. Our approach has this property since we do not require every party to succeed in decrypting their share but can continue as soon as we have enough shares according to the chosen threshold. The adversary still has the possibility to abort the protocol before distributing the shares of its seed openings, but the choice to abort has to be made *without* knowing the outcome of the coin toss. We stress that this is also inherent to any of the previous works on such compilers. Looking at the time-lock approach for example, the adversary is able to abort at any point in the actively secure puzzle generation. This abort can not be prevented, but it is important to ensure that the decision to abort has to be made without knowing the outcome of the coin-toss. Since the outcome of the coin-toss is only revealed after all the parties have already received consistent shares by the adversary, this is clearly achieved by our construction. Finally, compared to the time-lock approach, we avoid the use of complex general-purpose actively secure MPC and the required trusted set-up for the time-lock puzzles while obtaining the same (optimal) deterrence rate.

5.2 Building Blocks

We require several building blocks to obtain a working compiler. These are commitment schemes, signature schemes, joint coin-tossing, publicly verifiable secret-sharing and finally a passively secure base protocol.

Our construction makes use of a commitment scheme $(\texttt{Com}, \texttt{Open}, \texttt{Verify})$ (definition 2.1.1). For ease of notation we assume the commitment scheme to be non-interactive, meaning no additional interaction is required between the parties when opening a commitment. With some simple modifications, our compiler will work with interactive commitment schemes as well.

Next to that, we use a signature scheme $(\texttt{Gen}, \texttt{Sign}, \texttt{Verify})$ (definition 2.1.4) with *existential unforgeability against chosen plaintext attacks*. Before the start of the protocol, we assume each party P_i has executed $\texttt{Gen}$ to register a public-private key pair (pk_i, sk_i). Finally, we use a publicly verifiable secret sharing scheme $(\texttt{Distribute}, \texttt{Verify}, \texttt{Reconstruct})$ (definition 2.2.2).

5.2.1 Joint Coin Tossing

The adversary should not be able to influence the outcome of the coin-tossing protocol. Therefore, we require a coin-tossing protocol with security against an active adversary $\mathcal{A}$. An ideal functionality $\mathcal{F}_{\text{coin}}$ receives $\texttt{ok}_i$ from each party $P_i, i \in [n]$ and outputs a random k-bit string $\texttt{seed}$ to all the parties:

Ideal Functionality $\mathcal{F}_{\text{coin}}$

- Consider a number of parties $P_1, P_2, \ldots, P_n$
- If $\mathcal{F}_{\text{coin}}$ receives a message $(\texttt{flip})$ from P_i, it stores $(\texttt{flip}, P_i)$ in memory if it is not stored in memory yet.
- Once $\mathcal{F}_{\text{coin}}$ has stored all the messages $(\texttt{flip}, P_i)$ for $i \in [n]$, $\mathcal{F}_{\text{coin}}$ picks a random value r and sends $(\texttt{flip}, r)$ to all the parties.

5.2.2 Passively Secure Protocol

Our input-independent compiler takes a passively secure protocol (without private inputs) Π_{pass} for calculating an arbitrary function $(y_1, y_2, \ldots, y_n) \leftarrow f()$ and transform it into a protocol Π_{PVC} with covert security and public verifiability. Furthermore, we assume the passively secure protocol is secure against a constant fraction of corruptions $0 \leq c \leq n-1$. Furthermore, we require each party engaging in Π_{pass} to receive a transcript which is consistent with the transcripts of the other parties if everyone behaved honestly. This is required in order to be able to compare simulated transcripts and real transcripts later on.

To obtain such transcripts, we require (1) a fixed ordering of the messages and (2) that every party is able to see all the messages sent in the execution. Every passively secure protocol can be changed such that it has (1) by, for example, adding a program counter to all the messages. Similar to [Fau+21], we can simply broadcast every message sent during the protocol safely by assuming the passively secure protocol tolerates $n-1$ corruptions since this implies the adversary is allowed to view every message anyways. If we would want to tolerate passively secure protocols with security against lower numbers of corruptions, we could use symmetric keys between every pair of parties to encrypt the pairwise communication channels. Similar to [DOS20], we can then later reveal the keys for the communication channels for the opened executions alongside the randomness seeds. For ease of notation, we will not include this in our compiler description but believe adding this would be straightforward.

5.3 Input-independent PVC Compiler

In this section we will present the construction for our compiler for covert security with public verifiability $\texttt{COMP}_{\text{PVC}} = (\Pi_{\text{comp}}, \Pi_{\text{open}}, \Pi_{\text{reconstruct}})$ for protocols without private inputs. We start by explaining the seed generation procedure. After that, the mechanism for opening executions is explained followed by the accompanying seed reconstruction procedure. Using these, the complete compiler is presented followed by the additional `Blame`- and `Judge` algorithms. Finally, a security analysis of the input-independent compiler is presented. For readability, we present our protocols in the synchronous model. To make these run in an asynchronous fashion, techniques for "emulating" a synchronous protocol on an asynchronous network as presented in [Dam+09] can be used.

5.3.1 Seed Generation

To obtain covert security for passively secure protocols, we require the ability to verify the behaviour of the parties in every step of the protocol, including the randomness derivation. To ensure the security of *any* passively secure protocol, we therefore need to be able to verify the randomness generation of the parties to be uniform as well. To this end, the parties first engage in a seed generation protocol Π_{seed} for each of the k executions. The seed generation protocol can be found in protocol Π_{seed}. Every party P_i first generates a *private* seed $\texttt{seed}^i_{\text{priv}}$ and a public seed *share* $\texttt{seed}^i_{\text{pub}}$. Now, each party can reconstruct the (same) public seed $\texttt{seed}_{\text{pub}}$ as:

$$\texttt{seed}_{\text{pub}} = \bigoplus_{j=0}^{n} \texttt{seed}^j_{\text{pub}}$$

Finally, the parties receive their own private seed, information for opening their own seed and commitments to the private seeds of all the parties and the public seed. The (private) seed from which they will actually derive their randomness during the execution of the passively secure protocol is calculated as $\texttt{seed}^i = \texttt{seed}^i_{\text{priv}} \oplus \texttt{seed}_{\text{pub}}$. Because of the XOR, we are guaranteed that the seeds used by the parties are actually distributed uniformly at random.

Protocol Π_{seed}

This protocol works with an arbitrary number n of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. To generate uniformly random seeds for every party, the parties execute the following steps:

1. Party P_i samples uniformly random a private seed $\texttt{seed}^i_{\text{priv}}$, generates $(c^i, d^i) \leftarrow \texttt{Com}(\texttt{seed}^i_{\text{priv}})$ and sends c^i to the other parties.
2. P_i samples uniformly random a public seed share $\texttt{seed}^i_{\text{pub}}$ and sends $\texttt{seed}^i_{\text{pub}}$ to all the other parties.
3. Each P_i calculates the public seed $\texttt{seed}_{\text{pub}}$ as $\bigoplus_{j=0}^{n} \texttt{seed}^j_{\text{pub}}$.
4. If the parties have not received all the expected messages before some predefined timeout, the parties send `abort` to all the other parties and output `abort`. Otherwise, P_i outputs $\left(\texttt{seed}^i_{\text{priv}}, d^i, \texttt{seed}_{\text{pub}}, \{c^j\}_{j \in [n]}\right)$.

5.3.2 Execution Opening

Protocol Π_{open}

At the start of the protocol, all the parties know the encrypted seed shares $\{E_h(d_h)^{(i,j)}\}$ of every party $P_i, i \in [n]$ in every execution $j \in [k]$ for every party $P_h, h \in [n]$. Furthermore, the corresponding proofs $\texttt{dproof}_j^i$ as well as the signatures σ_j^i are known. Finally, each party P_i holds a set of private seed openings $\{d_1^i, d_2^i, \ldots, d_k^i\}$, a set of outputs $\{y_1^i, y_2^i, \ldots, y_k^i\}$ and a set of transcripts $\{\texttt{trans}_1^i, \texttt{trans}_2^i, \ldots, \texttt{trans}_k^i\}$. To open $k-1$ protocol executions, do the following:

Share verification:

1. First, the parties use the `Verify` algorithm of PVSS to check the validity of all the shares to generate the set:

$$M = \left\{(l, m) \in ([n], [k]) : \text{PVSS}.\texttt{Verify}\left(\texttt{dproof}_m^l, E_h(d_h)_{h \in [n]}^{(l,m)}\right) = \perp\right\}.$$

   If any of the parties obtain $M \neq \emptyset$, choose the tuple $(l, m) \in M$ with minimal l and m, calculate the certificate $\texttt{cert}_{invs} = \left(pk_l, \texttt{data}_m, E_j(d_j)_{j \in [n]}^{(l,m)}, \sigma_m^l\right)$ and output $\texttt{corrupted}_l$.

Joint coin tossing phase:

2. If all the verifications succeed, each party P_h sends $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$, receives $(\texttt{flip}, r)$ and calculates the joint coin toss as $\texttt{coin} = r \mod k$.
3. Now, the parties exchange the set of seeds they have used in the $k-1$ executions according to the coin toss such that each party P_i obtains:

$$\mathcal{D}_i = \left\{d_j^h : h \in [n], j \in [k] \setminus \texttt{coin}\right\}$$

   Optimistic case: Each party P_i generates $\phi_j^i \leftarrow \texttt{Sign}(d_j^i)$ for all of its seed openings $\{d_j^i\}_{j \in [k] \setminus \texttt{coin}}$ and sends (ϕ_j^i, d_j^i) to all the other parties. Each party P_i verifies the signatures and constructs $\mathcal{D}_i$.

   Pessimistic case: If a number of parties P_j fails to publish their seed shares and/or valid signatures within a given amount of time, the parties engage in an execution of $\Pi_{\text{reconstruct}}$ to obtain $\mathcal{D}_i$.

Output:

4. Finally, each party outputs $(\mathcal{D}_i, \texttt{coin})$.

After the k executions of the passively secure protocols are finished, the parties will engage in the protocol Π_{open} for opening $k-1$ of the executions. "Opening" an execution means that the parties get access to the randomness seeds used by the other parties in the execution to make their behaviour deterministic. This way, we can then compare their expected behaviour with their actual behaviour. Instead of choosing $k-1$

executions to be opened, our compiler lets the parties throw a coin jointly to select which execution is going to remain closed and used as output of the protocol. The ideal behaviour of this coin toss can be found in ideal functionality $\mathcal{F}_{\text{coin}}$. To ensure the adversary can not prevent the honest parties from receiving a (random) outcome of the coin toss, we require a protocol with active security for this. A description of the entire opening protocol can be found in the description of Π_{open}.

At the start of the protocol, all the parties have access to the encrypted shares of the seed openings of all the parties. Using the publicly verifiable secret sharing scheme, the parties can verify the sharings of all the seed openings and in case a cheating attempt is detected, output a certificate containing the published share encryptions together with the released proof, a signature of the cheater and some additional information needed by a judge. Since we make use of a publicly verifiable secret sharing scheme, this information can be checked by *anyone* to see that indeed a cheating attempt took place. In case multiple cheating attempts take place, the one with the lowest party and execution id is selected to ensure all the parties agree on which party cheated, as required by the original definition of covert security. If all the verifications succeed, the parties are guaranteed that the shares will reconstruct to some well-defined secret and thus they are guaranteed that the adversary can not prevent the honest parties from creating a publicly verifiable certificate if cheating is detected later on.

In the coin-tossing phase, the parties interact with the coin-tossing functionality $\mathcal{F}_{\text{coin}}$ to obtain the shared coin and distribute the correct seed openings according to this coin. Now, two things can happen: (i) The parties simply announce the seed openings for those executions or (ii) some parties refuse to announce (some of) their seed openings or their message is slow to arrive. In (i), the parties obtain all the required seed openings and are done with the protocol. We call this the optimistic case. In (ii), which we call the pessimistic case, the parties are still missing some of the seed openings. In this case, the parties engage in the reconstruction protocol $\Pi_{\text{reconstruct}}$ to reconstruct the missing openings from the shares which were distributed earlier and still obtain the required openings. Note that we not simply report the parties that refuse to open seeds as cheaters since we do not have any evidence and thus can not obtain a publicly verifiable certificate. Furthermore, in the asynchronous model a party that does not open a seed might still be honest but its message may be lost in the network. The reconstruction protocol will be explained in the next section.

5.3.3 Seed Reconstruction

A formal description of the reconstruction protocol can be found in $\Pi_{\text{reconstruct}}$. Note that this protocol is only executed in case some of the parties are missing some of the seed openings. If all the parties behaved honestly, $\Pi_{\text{reconstruct}}$ can be ignored. Before tossing the shared coin in the opening protocol, the parties have already shared all of their seed openings to the other parties using an arbitrary (n, t)-PVSS. To *guarantee* that the honest parties can reconstruct missing seed openings in case an adversary refuses to distribute them, we require that $n > 2t$.

If reconstructions are required, the parties start $\Pi_{\text{reconstruct}}$ by announcing to all the parties which seed openings they are still missing. For every missing message they receive, they decrypt their own share of the published share encryptions of the corresponding seed opening. This share together with a publicly verifiable proof of correct decryption is then sent to the parties which are still missing the seed opening. Using these proofs, these parties can then pool together $t+1$ shares of which the proofs are valid to reconstruct the correct seed opening. Since we assume an honest majority, we are guaranteed that we will indeed receive at

Protocol $\Pi_{\text{reconstruct}}$

At the start of the protocol, the encrypted seed openings shares $\{E_h(d_h)^{(i,j)}\}$ of every party $P_i, i \in [n]$ in every execution $j \in [k]$ meant for every party $P_h, h \in [n]$ as well as the corresponding proof strings $\texttt{dproof}^i_j$ are publicly known. The parties recover the seed openings they are missing in the following way:

Missing seeds announcement:

1. Each party P_i starts with a non-complete set of seed openings $\mathcal{D}_i$. Assume P_i did not receive the seed openings d^l_m of some party P_l in some execution P_m. Call the set of tuples (l, m) of missing seed openings $\mathcal{E}_i$.
2. For every tuple $(l, m) \in \mathcal{E}_i$, P_i sends a message $\texttt{missing}^i_{(l,m)}$ to all the other parties.

Missing seed reconstruction:

3. For every $\texttt{missing}^j_{(l,m)}$ message received by P_i, P_i performs the following steps:
   - If $m == \texttt{coin}$, skip this message.
   - Otherwise, P_i decrypts its corresponding share d_i from $E_i(d_i)^{(l,m)}$, computes the string $\texttt{rproof}^i_{(l,m)}$ and send $(d_i, \texttt{rproof}^i_{(l,m)})$ to P_j.
4. For every tuple $(l, m) \in \mathcal{E}_i$, P_i does the following:
   - For every received messages of the form $(d_j, \texttt{rproof}^j_{(l,m)})$, P_i verifies the $\texttt{rproof}^j_{(l,m)}$.
   - Once $t + 1$ of the received proofs are successfully verified, P_i reconstructs the seed opening d^l_m from the $t + 1$ shares and adds this to $\mathcal{D}_i$.

Output:

5. Finally, P_i outputs the set of seed openings $\mathcal{D}_i$.

---

least $t+1$ honest shares. The use of a PVSS allows everyone to verify that the distribution and reconstruction were done honestly, which prevents an adversary from accusing an honest party of cheating.

5.3.4 Complete Compiler

By piecing all of the aforementioned blocks together, we are ready to present the entire compiler in Π_{comp} for transforming an arbitrary n-party MPC protocol Π_{pass} with passive security and no private inputs into an n-party MPC protocol with covert security and public verifiability. This compiler takes a commitment scheme, a signature scheme, a PVSS and an actively secure coin tossing protocol to produce a PVC protocol in the honest majority setting. Finally, note that before the execution of the protocol, we require that every party participating has registered a public key.

Roughly speaking, our compiler proceeds in four separate phases: *seed generation*, *protocol execution*, *blame information creation* and *execution opening & verification*. During the seed generation phase, the parties set up the seeds from which they will derive their randomness during the protocol executions. Each party is

Protocol Π_{comp}

Before the protocol execution, we assume the parties have agreed on the amount of executions k, protocol description Π_{pass} and the public keys of all the parties $\{pk_i\}_{i\in[n]}$. Furthermore, each party P_i knows its own secret key sk_i. Finally, the compiler assumes a publicly verifiable secret sharing scheme PVSS is available. Now, the passively secure protocol Π_{pass} is compiled into a protocol Π_{PVC} with covert security and public verifiability in the following way:

Seed generation:

1. For each $j \in [k]$, party P_i and all the other parties engage in an execution of Π_{seed} to obtain:

$$\left(\mathtt{seed}^{(i,j)}_{\text{priv}}, d^i_j, \text{seed}^j_{\text{pub}}, \{c^i_j\}_{h\in[n]}\right)$$

   And P_i computes its seeds for all the executions $j \in [k]$ as: $\text{seed}^i_j = \text{seed}^{(i,j)}_{\text{priv}} \oplus \text{seed}^j_{\text{pub}}$.

Protocol execution:

2. Next, all the parties engage in k executions of Π_{pass} where P_i uses the random seed $seed^i_j$ and obtains an output y^i_j and transcript $\mathtt{trans}^i_j$ in each execution $j \in [k]$.

Blame information creation:

3. For each d^i_j, $j \in [k]$, P_i generates and publishes:

$$\left(\left\{E_h(d_h)^{(i,j)}\right\}_{h\in[n]}, \mathtt{dproof}^i_j\right) \leftarrow \text{PVSS}.\mathtt{Distribute}(d^i_j).$$

4. For each $j \in [k]$, party P_i creates a signature $\sigma^i_j \leftarrow \mathtt{Sign}_{\text{sk}_i}(\mathtt{data}_j)$ where $\mathtt{data}_j$ is defined as:

$$\mathtt{data}_j = \left(i, j, \text{seed}^j_{\text{pub}}, \{c^l_j, \mathtt{dproof}^l_j\}_{l\in[n]}, \mathtt{trans}_j\right)$$

   P_i broadcasts all the σ^i_j's and verifies the received signatures.

Execution opening & verification:

5. Next, all the parties engage in an execution of the execution opening protocol Π_{open} such that each P_i obtains: $(\mathtt{resp}, \mathtt{coin}) \leftarrow \Pi_{\text{open}}$.
6. If $\mathtt{resp} == \mathtt{corrupted}_j$, P_i outputs $\mathtt{corrupted}_j$.
7. Otherwise, P_i calculates $(m, \mathtt{cert}) = \mathtt{Blame}(\mathtt{view}^i)$.
8. If $\mathtt{cert} \neq \perp$, P_i broadcasts $\mathtt{cert}$ and outputs $\mathtt{corrupted}_m$. Otherwise, output $y^i_{\mathtt{coin}}$.

bound to these seeds since the commitments are known to everyone. In the next phase, the parties run the passively secure protocol k times in parallel using the random seeds from the previous phase and obtain an output and transcript for each of the executions. In the blame information creation phase, the parties are required to distribute the opening information for all of their randomness seeds to ensure the other parties will be able to reconstruct these seeds in case the party fails to publish these seeds later on if they have cheated in the protocol. Furthermore, every party is required to sign all the data needed to hold them accountable for a judge later on. Note that all of this is done *before* the coin-toss, which means the adversary has to guess whether to cheat or not, independent of the coin toss. Next, $k - 1$ executions are opened using the actively secure coin-tossing protocol by distributing the randomness seeds used by all the parties. At this point, it is too late for an adversary to abort since it's opening information has already been distributed in the previous phase. Finally the parties verify the behaviour of all the other parties during the opened executions, this procedure has been extracted to a separate `Blame` algorithm. If no cheating is detected, the parties output their output of the unopened execution. Otherwise, they output the obtained certificate.

An adversary can try to cheat in a number of ways in the resulting protocol Π_{PVC}. First, it can do so by causing the seed openings of its own seeds to fail. This could be achieved by either (i) distributing inconsistent shares in step 3 of Π_{comp} or (ii) sending an incorrect opening in step 3 of Π_{open}. (i) is easily detected by the verification algorithm of the PVSS scheme. Since everyone can verify this and a signature on the proof string has been published, the adversary can not later on deny have done this. Furthermore, the proofs of correct decryption ensure that the adversary can not announce a wrong share and the honest parties will always obtain the correct seed openings. Case (ii) is noticed when any of the seed openings fail. In this case, the adversary has already published a signature on the commitment *and* on the opening which means anyone can simply see that the opening fails and the adversary can not deny sending these values due to the signatures.

Furthermore, an adversary can attempt to cheat by deviating from the protocol description in any of the parallel executions. Since the protocol is ran without private inputs, deviating simply means using different randomness than what is expected based on the seed. If all of the seed openings succeeded, the parties can detect this when simulating the protocol executions later on. Since everyone knows the commitment and the opening, everyone knows the randomness that should have been used. Furthermore, the commitments to the seeds have been signed and thus an adversary can not deny that he has used the wrong randomness. In the next section, this verification of the behaviour of the parties using a `Blame` algorithm is explained.

5.3.5 Blame Algorithm

In the `Blame` algorithm, the behaviour of the parties is checked and a certificate is generated in case cheating was detected. This algorithm is an integral part of the compiler but for ease of notation, has been presented here as a separate algorithm. In reality, parts of the algorithm could already be run while performing other tasks in the compiler as soon as the required information is available to the parties. A formal description of the blame algorithm can be found in `Blame`(), which takes the view of a party as input.

First, the `Blame` algorithm verifies the seed openings of all the parties. If any of the seed openings fail, depending on the way in which the seed was obtained, a certificate of *invalid opening* is generated. If any of the seed openings fail (return $\perp$), the one with the lowest party- and execution id is picked to ensure the parties agree on which party cheated. Based on the way in which the seed opening was obtained, an *invalid opening (1 or 2)* is generated. If the seed opening was obtained via $\Pi_{\text{reconstruct}}$ (case 1), the certificate consists of the signature on the `data` of the corresponding execution, the `data` itself, the encrypted seed shares and all the seeds and reconstruction proofs from which the seed was obtained. If the seed was obtained

Algorithm `Blame(view)`

The `Blame` algorithm takes as input the view `view` of a party, which consists of:

- Public seeds $\mathtt{seed}^j_{\text{pub}}$ and public coin `coin`
- All the seed commitments and openings $\{c^i_j, d^i_j\}_{i\in[n], j\in[k]\setminus\mathtt{coin}}$
- Encrypted seed shares $\{E_h(d_h)^{(i,j)}\}_{h,i\in[n],j\in[k]}$
- PVSS proofs for distribution $\{\mathtt{dproof}^i_j\}_{i\in[n],j\in[k]}$ and reconstruction $\{\mathtt{rproof}^j_{(l,m)}\}_{j\in[n],(l,m)\in\mathcal{E}}$
- Public keys $\{pk_j\}_{j\in[n]}$, signatures $\{\sigma^i_j\}_{i\in[n],j\in[k]}$ and $\{\phi^i_j\}_{i\in[n],j\in[k]}$
- The set of tuples of missing seed openings before the pessimistic case $\mathcal{E}$
- Additional information $\{\mathtt{data}_j\}_{j\in[k]}$

To verify the behaviour of the parties, do:

1. Open the private seed of all the parties $P_i, i \in [n]$ in each execution $j \in [k] \setminus \mathtt{coin}$ as $\mathtt{seed}^{(i,j)}_{\text{priv}} \leftarrow \mathtt{Open}(c^i_j, d^i_j)$.
2. Construct the set $S = \{(l,m) \in ([n],[k]\setminus\mathtt{coin}) : \mathtt{seed}^{(i,j)}_{\text{priv}} == \perp\}$. If S is not empty, pick the tuple (l,m) with the lowest l, m and produce an invalid opening certificate:
   - **If** $(l,m) \in \mathcal{E}$, set $\mathtt{cert}_{\text{invo1}} = \left(pk_l, \mathtt{data}_m, \{d_j, \mathtt{rproof}^j_{(l,m)}\}_{j\in[n]}, \{E_j(d_j)^{(l,m)}\}_{j\in[n]}, \sigma^l_m\right)$
   - **Otherwise**, set $\mathtt{cert}_{\text{invo2}} = \left(pk_l, \mathtt{data}_m, d^l_m, \phi^l_m, \sigma^l_m\right)$

   And output $(m, \mathtt{cert}_{\text{invo(1/2)}})$.
3. If all the verifications succeeded, set the randomness seeds of each party P_i as $\mathtt{seed}^i_j = \mathtt{seed}^{(i,j)}_{\text{priv}} \oplus \mathtt{seed}^j_{\text{pub}}$ for each execution $j \in [k] \setminus \mathtt{coin}$.
4. Re-run each execution j of Π_{pass} for $j \in [k] \setminus \mathtt{coin}$ by simulating party P_i using random seed $\mathtt{seed}^i_j$ to obtain each transcript $\mathtt{trans}'_j$.
5. Using $\mathtt{data}_j$, construct the set $S = \{m : \mathtt{trans}_m \neq \mathtt{trans}'_m\}$. If S is not empty, pick the lowest m and find the party P_l that sends the first message in $\mathtt{trans}_m$ which is inconsistent with the expected message from $\mathtt{trans}'_m$ and construct a protocol deviation certificate:

$$\mathtt{cert}_{\text{dev}} = \left(\mathrm{pk}_l, \mathtt{data}_m, \{d^i_j\}_{i\in[n],j\in[k]\setminus\mathtt{coin}}, \sigma^l_m\right)$$

   And output $(m, \mathtt{cert}_{\text{dev}})$. Otherwise, output $(\cdot, \perp)$.

from the cheating party directly (case 2), the certificate only has to contain the signatures on the data and the seed opening, and the data and seed opening itself.

If all the seeds can be opened correctly, the `Blame` algorithm commences with verifying the behaviour of the parties during the protocol executions. To this end, the executions are simulated using the randomness seeds obtained in the previous step, resulting in *expected* transcripts $\mathtt{trans}'_j$ for $j \in [k] \setminus \mathtt{coin}$. If for any execution the actual transcript does not match with the expected transcript, the first party deviating from the protocol is identified and a *deviation certificate* is generated. This certificate consists of the data signature, the data itself and all the seeds for the execution in which the cheating took place. The certificates generated in the `Blame` algorithm as well as the one generated in Π_{comp} can then be sent to *anyone* who can check them to confirm that a party has cheated during the protocol.

5.3.6 Judge Algorithm

The `Judge` algorithm takes a certificate with which a party is accused of cheating. The `Judge` algorithm then checks this certificate to confirm that this party actually cheated. If the party indeed cheated according to the certificate, its public key is outputted and $\perp$ otherwise. Since the certificate contains all the information required to convince someone that cheating occurred, *anyone* running the `Judge` algorithm can verify the certificate. This includes third parties who do not have anything to do with the protocol execution itself, making our compiler is *publicly verifiable*. In practice, the `Judge` algorithm could for example be implemented in a smart contract to automatically punish a cheater.

The judge starts with a certificate $\texttt{cert}_{\text{type}}$. Regardless of which certificate type it receives, it first verifies the signature of the accused party on the `data`. If this signature is invalid, we can never be sure that the information was actually created and sent by the accused party and thus $\perp$ is returned. If the signature is valid, the judge commences with a certain set of steps, depending on the type of the certificate: *invalid sharing*, *invalid opening (1)*, *invalid opening (2)* or *deviation*. If the certificate is malformed and does not match any of the four templates, $\perp$ is returned.

In case of an invalid sharing, the judge only has to use the PVSS to see that indeed one or more of the published shares are invalid according to the proof string, which has also been signed indirectly. If the verification fails, we thus know that the accused party did indeed cheat and its public key is outputted.

If an invalid opening (1) certificate was given, this means that a party accuses another party from distributing an incorrect seed opening in the *pessimistic case*. Now first the published shares are verified. Note that if the verification fails, not the public key of the accused party is outputted but simply $\perp$ since if the verification already failed, the party will have generated an invalid sharing certificate. If the verification succeeds, this means that the accused party did give valid shares and we verify $t + 1$ of the decrypted shares of the other parties. Since we assume the existence of an honest majority, we can guarantee this by choosing the threshold t such that $n > 2t$. If enough valid shares are available, these can be used to reconstruct the seed opening, which is now guaranteed to be the one originally distributed by the accused party. If opening the seed now fails, this means that the accused party must have distributed the wrong opening information and thus its public key is returned. If we do succeed in opening something, the party was wrongly accused and $\perp$ is returned. The case for an invalid opening which was retrieved in the *optimistic* case is verified in a similar but more straightforward way. Here a judge only needs to verify whether the signature ϕ is a valid signature on the seed opening information. If this is the case but the seed opening fails, the accused party sent the wrong opening information and thus its public key is outputted.

Finally for the deviation certificate, the judge does not have to check the validity of the seed openings but can immediately use those to calculate the randomness each party should be using for a certain execution. The judge then simulates this execution of the passively secure protocol and compares the obtained transcript to the original transcript. If these indeed mismatch, the judge verifies whether the accused party was indeed the first party deviating from the protocol description. If this is also the case, its public key is outputted.

Algorithm `Judge(cert)`

We assume the judge knows the function Π_{pass} to be computed. To check a certificate, do:

- If $\texttt{Verify}(\texttt{data}_m, \sigma^l_m) = \bot$, output $\bot$.
- Else, interpret $\texttt{data}_m$ as $\left(l, m, \texttt{seed}^m_{\text{pub}}, \{c^j_m, \texttt{dproof}^j_m\}_{j\in[n]}, \texttt{trans}_m\right)$.

Depending on the type of certificate, do:

invs:

- $\texttt{cert}_{invs} = \left(pk_l, \texttt{data}_m, E_j(d_j)^{(l,m)}_{j\in[n]}, \sigma^l_m\right)$.
- If $\text{PVSS}.\texttt{Verify}(\texttt{dproof}^l_m, E_j(d_j)^{(l,m)}_{j\in[n]}) = \bot$, output pk_l. Otherwise, output $\bot$.

invo1:

- $\texttt{cert}_{\text{invo1}} = \left(pk_l, \texttt{data}_m, \{d_j, \texttt{rproof}^j_{(l,m)}\}_{j\in[n]}, \{E_j(d_j)^{(l,m)}\}_{j\in[n]}, \sigma^l_m\right)$.
- If $\text{PVSS}.\texttt{Verify}(\texttt{dproof}^l_m, E_j(d_j)^{(l,m)}_{j\in[n]}) = \bot$, output $\bot$.
- Verify $t+1$ of the $\texttt{rproof}^j_{(l,m)}$'s and use the corresponding d_j's to reconstruct d^l_m. If no $t+1$ valid shares are available, output $\bot$.
- If $\texttt{Open}(c^l_m, d^l_m) \neq \bot$, output $\bot$. Otherwise, output pk_l.

invo2:

- $\texttt{cert}_{\text{invo2}} = \left(pk_l, \texttt{data}_m, d^l_m, \phi^l_m, \sigma^l_m\right)$.
- If $\texttt{Verify}_{\text{pk}_l}(d^l_m, \phi^l_m) = \bot$, output $\bot$.
- If $\texttt{Open}(c^l_m, d^l_m) \neq \bot$, output $\bot$. Otherwise, output pk_l.

dev:

- $\texttt{cert}_{\text{dev}} = \left(\text{pk}_l, \texttt{data}_m, \{d^i_j\}_{i\in[n], j\in[k]\setminus\texttt{coin}}, \sigma^l_m\right)$.
- For every party P_i and execution $j \in [k] \setminus \texttt{coin}$, open $\texttt{seed}^{(i,j)}_{\text{priv}} \leftarrow \texttt{Open}(c^i_j, d^i_j)$ and calculate $\texttt{seed}^i_j$ as $\texttt{seed}^{(i,j)}_{\text{priv}} \oplus \texttt{seed}^j_{\text{pub}}$.
- Re-run execution j of Π_{pass} by simulating each party P_i using random seed $\texttt{seed}^i_j$ to obtain transcript $\texttt{trans}'_j$.
- If $\texttt{trans}'_j == \texttt{trans}_j$, output $\bot$.
- If the first party that sends an incorrect message in $\texttt{trans}'_j$ is indeed P_l, output pk_l. Otherwise, output $\bot$.

Otherwise:

- If the certificate does not match any of the four formats, output $\bot$.

5.4 Security

First of all we have defined an ideal functionality for the coin flipping functionality. As can be seen in the description of $\mathcal{F}_{\text{coin}}$, all the parties receive the output from the ideal functionality simultaneously. This means that we require the real-world protocol with which this functionality is built to (at least) guarantee *fairness*. As explained in Section 2.3.2, it has been proven in [BGW88] that obtaining fairness is always possible in the case of an honest majority. Therefore we will assume the existence of such a protocol Π_{coin} which securely implements $\mathcal{F}_{\text{coin}}$ with fairness in the presence of an honest majority.

To proof that the compiler presented above satisfies the definition for covert security with public verifiability (Definition 2.3.7), we first state the guarantees in Theorem 1 and then proof that our compiler satisfies the requirements of *covert security (with deterrence rate ϵ)*, *public verifiability* and *defamation-freeness* separately.

Theorem 1. *Suppose the PVSS ($\texttt{Distribute}$, $\texttt{Verify}$, $\texttt{Reconstruct}$) satisfies the privacy, correctness and soundness properties with a threshold $t < n/2$. Furthermore, assume the commitment scheme ($\texttt{Com}$, $\texttt{Open}$, $\texttt{Verify}$) is binding and hiding. Let the signature scheme ($\texttt{Gen}$, $\texttt{Sign}$, $\texttt{Verify}$) be existentially unforgeable under chosen plaintext attacks. Finally, assume Π_{coin} implements $\mathcal{F}_{coin}$ with active security against a dishonest minority. If Π_{pass} provides passive security against $n-1$ corruptions, the compiler $\texttt{COMP}_{\text{PVC}} = (\Pi_{comp}, \Pi_{open}, \Pi_{reconstruct})$ with the additional algorithms $\texttt{Blame}$ and $\texttt{Judge}$ is covertly secure with public verifiability against $t < \frac{n}{2}$ corruptions with deterrence rate $\epsilon = 1 - \frac{1}{k}$.*

Proof.

Covert Security To show that our compiler meets the definition for covert security with deterrence rate $\epsilon = 1 - \frac{1}{k}$, we will construct a simulator $\mathcal{S}$ in the ideal world, talking to the trusted party $\mathcal{F}_{\text{Covert}}$ and the real-world adversary $\mathcal{A}$. After that, we will argue that the joint distribution of $\mathcal{S}$ and the output of $\mathcal{F}_{\text{Covert}}$ is indistinguishable from the views of all the parties in the real-world execution of $\texttt{COMP}_{\text{PVC}}$. Let the adversary $\mathcal{A}$ corrupt all the parties in some set $\mathbb{A}$ with $|\mathbb{A}| < \frac{n}{2}$. Furthermore, let $\mathbb{P} = [n] \setminus \mathbb{A}$ be the set of honest parties. The Simulator $\mathcal{S}$ now looks as follows:

0. For $P_i \in \mathbb{P}$, $\mathcal{S}$ generates a random pair (sk^i, pk^i) and sends all pk^i to $\mathcal{A}$ for $i \in \mathbb{P}$. $\mathcal{S}$ receives all pk^i for $i \in \mathbb{A}$ from $\mathcal{A}$.
1. $\mathcal{S}$ honestly engages in k executions of Π_{seed} with $\mathcal{A}$. For each $i \in \mathbb{P}$ and execution $j \in [k]$, $\mathcal{S}$ receives $(\texttt{seed}^{(i,j)}_{\text{priv}}, d^i_j, \texttt{seed}^j_{\text{pub}}, \{c^h_j\}_{h\in[n]})$.
2. $\mathcal{S}$ engages in k executions of Π_{pass} with $\mathcal{A}$ where for $i \in \mathbb{P}$, $\mathcal{S}$ uses randomness derived from $\texttt{seed}^i_j = \texttt{seed}^{(i,j)}_{\text{priv}} \oplus \texttt{seed}^j_{\text{pub}}$. Let $\texttt{trans}_j$ be the transcript obtained by $\mathcal{S}$ for execution $j \in [k]$. Let y^i_j be the output obtained by P_i in execution j.
3. Each party P_i distributes its seed openings d^i_j for execution $j \in [k]$ as $(\{E_h(d_h)^{(i,j)}\}_{h\in[n]}, \texttt{dproof}^i_j) \leftarrow \text{PVSS}.\texttt{Distribute}(d^i_j)$. $\mathcal{S}$ does this honestly for $P_i \in \mathbb{P}$ while $\mathcal{A}$ does this for $P_i \in \mathbb{A}$.
4. $\mathcal{S}$ computes signatures σ^i_j for each $P_i \in \mathbb{P}$ and execution $j \in [k]$ as an honest party and sends these to $\mathcal{A}$. For each $i \in \mathbb{A}$, $\mathcal{S}$ receives σ^i_j from $\mathcal{A}$.
5. If any of the received signatures are invalid or $\mathcal{S}$ has not received the (expected) messages from $\mathcal{A}$ in any of the communication rounds, S sends $(\texttt{abort})$ to $\mathcal{F}_{\text{Covert}}$ and $\mathcal{A}$ and halts.
6. For each set $(\{E_h(d_h)^{(i,j)}\}_{h\in[n]}, \texttt{dproof}^i_j)$ with $i \in \mathbb{A}$, $\mathcal{S}$ uses $\text{PVSS}.\texttt{Verify}$ to check whether the distributed shares are valid and if not:

  - Send $(\texttt{corrupted}, i^*)$ to $\mathcal{F}_{\text{Covert}}$ where i^* is the first party to distribute invalid shares.
  - Compute an invalid sharing certificate like an honest party would do and send this to $\mathcal{A}$.
  - Output whatever $\mathcal{A}$ outputs and stop the simulation.

7. $\mathcal{S}$ checks whether $\mathcal{A}$ has cheated in any of the execution in step 2. Let P_l be the first party to cheat in some execution m. Add all tuples (l, m) to a set M.
8. $\mathcal{S}$ decrypts all the shares $\{E_i(d_i)_{i \in \mathbb{P}}^{(l,j)}\}$ to reconstruct d_j^i for each party $P_i \in \mathbb{A}$ for each execution $j \in [k]$. For each of the pairs (c_j^i, d_j^i), if $\texttt{Open}(c_j^i, d_j^i) = \perp$ then add (i, j) to the set M as well if it was not in there yet.
9. With M as the set of all executions in which $\mathcal{A}$ cheated in some way, we distinguish three distinct cases:

   $|M| > 1$: In this case, cheating is guaranteed to be detected and $\mathcal{S}$ sets $\texttt{flag} == \texttt{detected}$.

   $|M| = 1$: In this case, $\mathcal{S}$ sends $(\texttt{cheat}, l)$ to $\mathcal{F}_{\text{Covert}}$ and receives either $\texttt{detected}$ or $\texttt{undetected}$.
   - In case $\texttt{detected}$ was received, set $\texttt{flag} = \texttt{detected}$.
   - In case $\texttt{undetected}$ was received, set $\texttt{flag} = \texttt{undetected}$.

   $|M| = 0$: In this case, set $\texttt{flag} = \texttt{all\_honest}$.
10. Depending on $\texttt{flag}$, do the following:
    (a) If $\texttt{flag} == \texttt{detected}$, repeat the following steps:
      1*. All parties send $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$ and receive $(\texttt{flip}, r)$.
      2*. All parties calculate $\texttt{coin}^* = r \mod k$.
      3*. If $|M \setminus \texttt{coin}^*| > 0$, set $\texttt{coin} = \texttt{coin}^*$ and continue. Otherwise, rewind $\mathcal{A}$ to before step 1* and try again.

    (b) If $\texttt{flag} == \texttt{undetected}$, repeat the following steps:
      1**. All parties send $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$ and receive $(\texttt{flip}, r)$.
      2**. All parties calculate $\texttt{coin}^* = r \mod k$.
      3**. If $\texttt{coin}^* \in M$, set $\texttt{coin} = \texttt{coin}^*$ and continue. Otherwise, rewind $\mathcal{A}$ to before step 1** and try again.

    (c) If $\texttt{flag} == \texttt{all\_honest}$, all parties send $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$, receive $(\texttt{flip}, r)$ and calculate $\texttt{coin} = r \mod k$.
11. For each execution $j \in [k] \setminus \texttt{coin}$, $\mathcal{S}$ computes $\phi_j^i \leftarrow \texttt{Sign}(d_j^i)$ for each $P_i \in \mathbb{P}$ and sends (ϕ_j^i, d_j^i) to $\mathcal{A}$. $\mathcal{S}$ receives (ϕ_j^i, d_j^i) for $P_i \in \mathbb{A}$ from $\mathcal{A}$.
    - For every *valid* pair (ϕ_m^l, d_m^l) received, if $\texttt{Open}(c_m^l, d_m^l) \neq \perp$ and (l, m) is in M only because it was detected in step 8, remove (l, m) from M. If now $M = \varnothing$, set $\texttt{flag} = \texttt{all\_honest}$.
    - For every *invalid* pair (ϕ_m^l, d_m^l) received for some honest party $P_i \in \mathbb{P}$, it sends $\texttt{missing}_{(l,m)}^i$ to $\mathcal{A}$.

    If $\mathcal{S}$ receives a message $\texttt{missing}_{(l,m)}^i$ from $\mathcal{A}$ and $m \neq \texttt{coin}$, $\mathcal{S}$ sends decryptions of all the shares $\{E_i(d_i)_{i \in \mathbb{P}}^{(l,m)}\}$ and the corresponding proofs to $\mathcal{A}$.
12. Finally depending on $\texttt{flag}$, $\mathcal{S}$ does the following:
    (a) If $\texttt{flag} == \texttt{detected}$:
      - Send $(\texttt{corrupted}, l)$ to $\mathcal{F}_{\text{Covert}}$.
      - Compute a certificate like an honest party would do and send this to $\mathcal{A}$.
      - Output whatever $\mathcal{A}$ outputs and stop the simulation.

    (b) If $\texttt{flag} == \texttt{undetected}$: Send y_{coin}^i as the output of each P_i to $\mathcal{F}_{\text{Covert}}$, outputs whatever $\mathcal{A}$ outputs and stop the simulation.

(c) If `flag == all_honest`: Send `continue` to $\mathcal{F}_{\text{Covert}}$, output whatever $\mathcal{A}$ outputs and stop the simulation.

Public Verifiability First, we proof that $\texttt{COMP}_{\text{PVC}}$ prevents a selective abort and after that we proof that the generated certificates will be accepted by the `Judge` with an overwhelming probability.

Note that an adversary *is* able to abort the protocol and hence the generation of a certificate *before* the coin-toss, but should be unable to prevent the generation of a certificate *after* it has seen the outcome of the coin-toss. To this end, observe that every party is asked to distribute the opening information for all of its seeds in Step 3 of Π_{comp} and sign the `data` that is required to create a certificate for every execution in $[k]$ in Step 4. Both of these happen before the coin-toss is even performed. At this point the adversary can thus not base it's decision to cheat on the outcome of the coin toss. Cheating at this point means the adversary either distributes incorrect shares for (some of) its seed openings or distributes an incorrect signature for (some of) the `datas`. If any of these two happens, the other parties can detect it by the verifiability of the PVSS (Step 1 of Π_{open} and the signature scheme (Step 4 of Π_{comp}) before the coin toss. If a party distributes incorrect shares, this can be seen by *anyone* and thus public verifiability of this cheating attempt is easily obtained. If any of the signature verifications fail, the parties simply abort, which is accepted as explained earlier.

On the other hand, if verifications for all the shares for all of the seed openings succeeded, and valid signatures for all of the `datas` have been received by all the parties, the honest parties are guaranteed to be able to generate a certificate if they detect cheating. To see this, observe that Π_{coin} gives all the parties the outcome of the coin toss at the same time. After that, the parties either obtain the correct seed openings from all the other parties (i) directly (optimistic case) or (ii) can reconstruct the secret from the earlier distributed shares (pessimistic case). To see why (ii) holds, we look at the properties of the PVSS. Since the PVSS satisfies the *correctness* property, we are guaranteed that since the verification of the distributed shares succeeded, any reconstructed seed will always be the one that was originally distributed, except with negligible probability. Furthermore, this means that if a party correctly decrypts its share and publishes the corresponding proof, any honest party will accept this share. Due to the *soundness* property, we are also guaranteed that a subset of $t + 1$ of valid shares is guaranteed be able to reconstruct the original secret. Since we assume an honest majority, we are guaranteed that at least $t + 1$ parties successfully decrypt and publish a proof in Step (3) of $\Pi_{\text{reconstruct}}$. After that, we are guaranteed that a party missing a seed opening obtains at least $t + 1$ valid shares and is thus able to reconstruct the original seed opening successfully. On the other hand, the adversary is unable to reconstruct the seed openings belonging to `coin` since this requires at least 1 share of an honest participant. A seed opening d^l_m and signed data σ^l_m are enough to create a certificate if party $P_l, l \in [n]$ is detected to have cheated in execution $m \in [k]$.

Now, it remains to show that if an honest party P_i outputs `cert` when it detects cheating by another party P_j, then the `Judge` outputs pk^j except with negligible probability. To proof this, we show it for the four types of certificate separately:

invs If an honest party outputs an invalid sharing certificate, this is because the PVSS.`Verify` method failed for some share for some seed opening of a party P_l for some execution m. Since the corresponding proof $\texttt{dproof}^l_m$ has been signed directly in σ^l_m and the PVSS is publicly verifiable, the `Judge` can check the validity of the signature and whether the verification indeed fails.

invo1 If an honest party outputs an invalid opening (1) certificate, this is because for some party P_l and some execution m, the opening information d^l_m received indirectly via the shares $\{E_j(d_j)^{(\mathrm{l,m})}\}_{j\in[n]}$ is inconsistent with the commitment c^l_m. This c^l_m has been signed directly in σ^l_m while d^l_m has been obtained via the PVSS. Using the publicly verifiable $\mathtt{dproof}^l_m$, the `Judge` can verify that the distribution was done correctly and due to the signature, P_l can not claim to have distributed a different seed. Using the $\mathtt{rproof}^j_{(\mathrm{l,m})}$s and the corresponding d_j for $j \in [n]$, the `Judge` can find at least $t + 1$ valid shares, which are guaranteed to reconstruct the originally distributed d^l_m. Now, the `Judge` can simply verify that the opening information d^l_m is indeed inconsistent with c^l_m.

invo2 If an honest party outputs an invalid opening (2) certificate, this is because for some party P_l and some execution m, the opening information d^l_m received directly from P_l is inconsistent with the commitment c^l_m. This c^l_m has been signed directly in σ^l_m while d^l_m has been signed in ϕ^l_m. The `Judge` can simply check the validity of the signatures and whether c^l_m and d^l_m are indeed inconsistent.

dev Finally, if an honest party outputs a deviation certificate, this means that for some execution $m \in [k] \setminus \mathtt{coin}$ of Π_{pass}, the honest party has detected cheating. Let P_l be the first party to send an inconsistent message in $\mathtt{trans}_j$. The `Judge` can open all the private seed shares for this execution as $\mathtt{seed}^{(j,m)}_{\mathrm{priv}} \leftarrow \mathtt{Open}(c^j_m, d^j_m)$ for $j \in [n]$ and compute the seeds that should have been used by all the parties. After that, the `Judge` can simulate the execution as well to obtain $\mathtt{trans}'_m$. Now, $\mathtt{trans}_j$ has been signed directly in σ^l_m and d^l_m either directly via ϕ^l_m or indirectly via σ^l_m (optimistic or pessimistic respectively). Therefore, if $\mathtt{trans}_m == \mathtt{trans}'_m$ *and* P_m is also the first party to send an inconsistent message in $\mathtt{trans}'_m$, the `Judge` knows P_m must have cheated. Note that cheating in the input-independent setting simply means a party used randomness that was inconsistent with its randomness seed. Since in σ^l_m, party P_m signed the public seed $\mathtt{seed}^m_{\mathrm{pub}}$ as well as the commitment to its own seed opening c^l_m, there is no way for P_m to somehow claim to have used a different randomness seed.

Defamation-freeness Recall that in order for a protocol to have *defamation-freeness*, it should be impossible for an adversary to craft a certificate that incriminates an honest party successfully (i.e. such that the `Judge` accepts it), except with negligible probability. To proof that our compiler has this, we will show that *if* an adversary would be able to craft such a certificate incriminating an honest party P_i, this contradicts the security of either the commitment scheme, the PVSS or the signature scheme. We do this for the four types of certificates separately.

invs If the `Judge` accepts an invalid sharing certificate this means that for some seed opening d^i_m of an honest party P_i in execution $m \in [k]$, PVSS.$\mathtt{Verify}(\mathtt{dproof}^i_m, E_h(d_h)^{(l,m)}_{h\in[n]}) = \perp$. Since P_i would only honestly publish the proof as well as the encrypted shares, this means that the *correctness* of the PVSS should be broken which contradicts the security assumptions from theorem 1.

invo1 If the `Judge` accepts an invalid opening (1) certificate, this means that for some seed commitment c'^i_m sent by an honest P_i and opening d'^i_m reconstructed via the PVSS, $\mathtt{Open}(c'^i_m, d'^i_m) = \perp$. Note that c'^i_m has been signed in σ'^i_m while d'^i_m has been obtained via the PVSS. An honest party P_i would only distribute the correct seed opening with the PVSS and a successful reconstruction should therefore always lead to the correct seed opening unless the *soundness* of the PVSS is broken. Furthermore, an honest party P_i only signs the commitment and openings that he received from the seed generation protocol. Therefore, either c'^i_m and d'^i_m must be correct or an adversary must be able to break the *existential unforgeability* of the signature scheme. Both of these contradict the security assumptions from theorem 1.

invo2 If the Judge accepts an invalid opening (2) certificate, this means that for some seed commitment c'^i_m and opening d'^i_m sent by an honest P_i, $\texttt{Open}(c'^i_m, d'^i_m) = \perp$. Note that c'^i_m has been signed in σ'^i_m while d'^i_m has been signed in ϕ'^i_m. Now, an honest party P_i only signs the commitment and openings that he received from the seed generation protocol. Therefore, in order for an adversary to make the opening fail, it must be able to break the *existential unforgeability* of the signature scheme, contradicting the security assumptions from theorem 1. Otherwise, c'^i_m and d'^i_m must be correct.

dev Finally if the Judge accepts a deviation certificate, this means that the certificate must contain a transcript $\texttt{trans}'_m$ for an execution m signed by P_i in σ^i_m where P_i is the first to send a message that is inconsistent with its randomness seed $\texttt{seed}^i_m$. However, since P_i is honest, he will follow the protocol honestly, this means that he will behave honestly in execution m and only sign the transcript honestly. If the signature σ^i_m on $\texttt{trans}'_m$ is valid but the transcript does blame P_i, this means that the adversary must be able to break the *existential unforgeability* of the signature scheme, contradicting the security assumptions. On the other hand, it could be that the transcript and the signature are valid, but that somehow the adversary can convince the Judge that another randomness seed $\texttt{seed}'^i_m \neq \texttt{seed}^i_m$ should have been used by P_i. Recall that the randomness seed for P_i is calculated as $\texttt{seed}'^{(i,m)}_{\text{priv}} \oplus \texttt{seed}'^m_{\text{pub}}$. Here, the private seed is obtained via the pair (c^i_m, d^i_m). Since this public seed $\texttt{seed}^m_{\text{pub}}$ as well as c^i_m have also been signed in σ^i_m, this again means that the adversary must be able to forge a signature. Finally, the adversary could find another d'^i_m such that $\texttt{Open}(c^i_m, d'^i_m)$ is valid, but in that case the adversary has found two messages m and m' such that $\texttt{Com}(m) == \texttt{Com}(m')$, which means it must be able to break the *binding* property of the commitment scheme. All of this contradicts with the security assumptions from theorem 1.

□

5.5 Input-dependent Protocols

$\texttt{COMP}_{\text{PVC}}$ only works for protocols that do not take any private inputs. This allowed us to de-randomise $k-1$ of the protocol executions since the behaviour of the parties is only dependent on the protocol specification and the used randomness. However, in the *input-dependent* setting, the input of a party also influences the behaviour of that party in the protocol. Since this input is private, we can not simply open this alongside the randomness seed since this would require us to send it to the other parties. Instead, we follow the one-to-many player virtualisation strategy as explained in Section 4.2.2. We call the resulting compiler for transforming passively secure protocols with private inputs into covertly secure protocols with public verifiability $\texttt{COMP}^*_{PVC} = (\Pi^*_{\text{PVC}}, \Pi^*_{\text{open}}, \Pi^*_{\text{reconstruct}})$.

$\texttt{COMP}^*_{PVC}$ and $\texttt{COMP}_{PVC}$ work in a very similar fashion. The main differences lie in the way the passively secure protocol is executed. Instead of running Π_{pass} k times in parallel, the n real parties now each simulate k virtual parties who, together, execute a related nk-party protocol Π^*_{pass} only once. To hide their private inputs for opening later, the real parties use k-out-of-k secret sharing to share their private input to their virtual parties, who use these as their private inputs in Π^*_{pass}. Since any $k-1$ of these shares do not reveal anything about the original inputs of the real party, we can safely open $k-1$ of the virtual parties by revealing the inputs and randomnesses used by these virtual parties and obtain the same deterrence rates of $\frac{k-1}{k}$ as in the input-independent protocol.

The function implemented by Π^*_{pass} can now be characterised as follows. Suppose we have the function $f : \mathbb{F}^n \rightarrow \mathbb{F}^N$ that takes an input x^i from each party $P_i, i \in [n]$ and yields every P_i an output y_i. Now, Π^*_{passs} should implement a related functionality $f' : \mathbb{F}^{nk} \rightarrow \mathbb{F}^{nk}$ such that:

$$f\left(\oplus_j x^1_j, \oplus_j x^2_j, \ldots, \oplus_j x^n_j\right) = \bigoplus_{i=0}^{k} f'_i\left(x^1_1, x^1_2, \ldots, x^n_k\right)$$

for all x^i_j with $1 \leq i \leq n$ and $1 \leq j \leq k$. For now, we assume we can find such a related functionality. In Chapter 6, a concrete example of a generic way for constructing such a related protocol in a real framework will be presented.

5.5.1 Opening the Virtual Parties

Checking the behaviour of the virtual parties works in a similar fashion as checking the behaviour of a real party in the input-independent compiler. Now instead of only revealing the randomness seeds from which the opened virtual parties will derive their randomness, we also reveal the inputs of these virtual parties. Important here is the observation that if a real party attempts to cheat, he has to attempt to cheat in *at least* one of its virtual parties. During the protocol execution, the real parties communicate on behalf of their virtual parties. To keep track of the views of the virtual parties, we require every message sent and received by the virtual parties to be broadcast like in the input-independent compiler. This means that also a message sent from one virtual party to another virtual party belonging to the same real party needs to be broadcast. As stated earlier, the main difference between this compiler and the input-independent compiler is the fact that instead of executing an n-party protocol k times, we now execute an nk-party protocol once. For every message sent from a sender to a receiver in the original protocol, the sender now has to send one message from all of its virtual parties to all of the virtual parties of the receiver. This thus scales with a factor of k^2, making computation as well as the communication complexity of this compiler more costly compared to the input-independent compiler.

5.5.2 The protocol

Since $\texttt{COMP}^*_{\text{PVC}}$, the additional $\texttt{Blame}^*$ and $\texttt{Judge}^*$ algorithms as well as the additional protocols Π^*_{open} and $\Pi^*_{\text{reconstruct}}$ are very similar to the algorithms and protocols presented for the input-independent compiler, we do not present their formal specifications. Compared to the input-independent protocols, the main difference is that the repetitions are now not repetitions of the passively secure protocol but virtual parties. Next to a seed for every repetition, the parties now also need to generate an input for each virtual party using additive secret sharing. This input is treated in a similar fashion as the seeds in the original protocol, meaning a commitment is made on each virtual party input as well. Furthermore, these inputs are also distributed using the PVSS in Step 3 and added to the signed data in Step 4. The final notable difference is in Step 3, where now only *one* execution of the passively secure nk-party protocol is executed with security against $nk - 1$ corruptions. To verify the behaviour of a virtual party, the parties can simulate the execution of the passively secure protocol by simulating the opened virtual parties based on their input, randomness and their incoming messages. Note that to verify the behaviour of a virtual party according to a protocol specification, we do not need to know whether the incoming messages are correct since they do not influence the correctness of the response of that virtual party. The unopened virtual parties can therefore be simulated by the messages they sent in the original transcript and assuming that those virtual parties behaved honestly. This is similar to how we assume the parties behaved honestly in the unopened execution in the input-independent compiler.

Chapter 6

Instantiation with MPyC

The input-dependent and input-independent compilers presented in Chapter 5 are fully generic, meaning they can be applied to any protocol with or without inputs respectively. For concreteness, we have applied and implemented the input-dependent compiler in an actual framework for multi-party computation called *MPyC* [Sch18]. In this chapter, we will describe the concept of asynchronous computation, the structure of the MPyC framework with the underlying protocols for performing MPC, the approach for transforming these protocols from passive to covert security using *player virtualisation* and finally how this was implemented.

6.1 Asynchronous Computation using `asyncio`

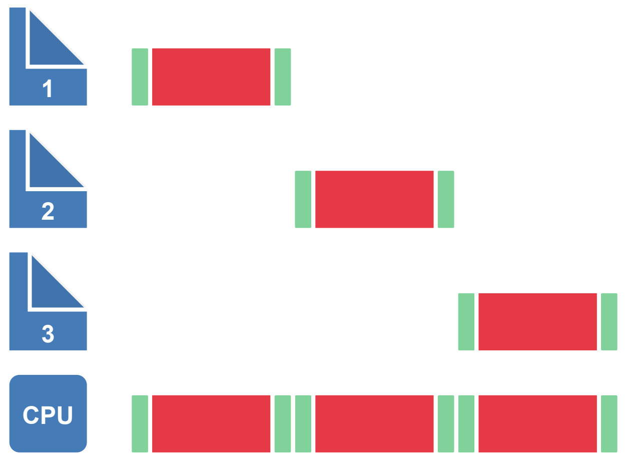
In this work, we will be implementing a proof-of-concept in the MPyC framework [Sch18], which performs all of its computations in an asynchronous manner. In order to understand how this influences the design of MPC protocols, this section will explain what asynchronous computation is and how this can be implemented in practice.

To realise asynchronous computation, MPyC makes heavy use of python's `asyncio` library.[1] The goal of asynchronous computations is to utilise the CPU as efficient as possible by constantly performing useful tasks instead of waiting for another task to finish before proceeding execution. To highlight the importance of asynchronous programming, lets look at an the example of downloading files from a server via the internet. Suppose we want to use python to download multiple files from a certain server using a getFile method:

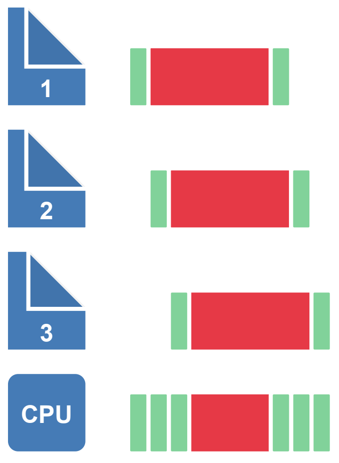
```
content = getFile("https://server.com/", file=1)
```

If we were to fetch multiple files sequentially, the machine will query the first file, block while it waits for the response to arrive, fetch the second file etc. On the other hand, if we would be able to do this in an asynchronous fashion, we can already start querying the second file while we are waiting for the first one to arrive. A schematic overview of the CPU usage with synchronous and asynchronous file querying can be found in figure 6.1. Here, the green parts represent the CPU doing useful work while the red parts are downtime while we are waiting for the files to arrive. As can be seen, if we can already start doing useful

[1]https://docs.python.org/3/library/asyncio.html



(a) Synchronous querying



(b) Asynchronous querying

Figure 6.1: Synchronous vs. asynchronous file querying. The green parts indicate time where the CPU is doing useful work while the red parts indicate downtime. As can be seen, the CPU is used much more efficiently with asynchronous querying.

work while we are waiting for the files to arrive, we can utilise the CPU much more efficiently.

By using `asyncio`, we can realise this exact behaviour using the two keywords *async* and *await*. A method can be turned into an asynchronous method by adding the *async* keyword, such a method is called a *coroutine*. Inside such a coroutine we can use the *await* keyword to indicate we want to wait for another coroutine to complete. While waiting for the other coroutine to complete, asyncio knows it can start doing something else. The central mechanism to schedule all of this is the *event loop*. The event loop is essentially a list of scheduled tasks which need to be executed asynchronously. This loop runs a certain task until it reaches an await statement. While waiting for the awaited coroutine to complete, the event loop will start running the next task. If we turn the getFile method into an asynchronous method, we can now query the three files in an asynchronous manner like this:

```
import asyncio

loop = asyncio.get_event_loop()

async def query_files():
  tasks = []
  for i in range(1,4):
    task = loop.create_task(getFile("https://server.com/", file=i))
    tasks.append(task)
  for task in tasks:
    content = await task
    print(content)

loop.run_until_complete(query_files())
loop.close()
```

Here we see the asynchronous query_files() method, which creates a new task for every file to be queried. Next, we await all the tasks, meaning the event loop may pause the query_files() function until the file task is done and can thus start downloading the file. As downloading the file may take some time, the event loop can then continue with the next task.

As explained previously, when we normally make a call to a method such as getFile(), we block the program execution until this method is done and returns something, such as a file. In order to prevent this blocking behaviour with asyncio, an asynchronous method immediately returns with a *future* result. Such a future is essentially a placeholder which will later be filled with the content the method should actually return, such as the file that was queried. By awaiting the future instead, we indicate we do not want to continue the execution until the future is filled with the actual content.

6.2 MPyC

MPyC is a python[2] framework for MPC developed in 2018 by Berry Schoenmakers, inspired by some of the fundamental building blocks of the Virtual Ideal Functionaly Framework (VIFF) [Gei10]. Using MPyC, arbitrary functionalities can be developed which are then executed in an MPC fashion. These functionalities can be as simple as calculating the average over a number of private inputs but can also be used to perform more involved tasks such as performing secure AES encryption and decryption and even training and evaluating complex machine learning tasks such as convolutional neural networks. The (open-sourced) source code of MPyC can be found on their GitHub page[3], where also more examples of MPyC implementations can be found.

The goal of MPyC is to provide its users with a simple python front-end to program arbitrary computations without dealing with the complex underlying MPC protocols, which are entirely handled by the back-end of MPyC. A simple program for calculating the average of the inputs of an arbitrary number of parties in MPyC can be found in listing 6.1. As can be seen, the actual computation which takes places in line 10 does not really differ from pure python.

```python
from mpyc.runtime import mpc

secint = mpc.SecInt()

private_input = secint(int(input("Private input: ")))

mpc.run(mpc.start()) #required only when run with multiple parties

inputs = mpc.input(private_input)
avg_input = sum(inputs) / len(inputs)
print('Average input: {}'.format(mpc.run(mpc.output(avg_input))))

mpc.run(mpc.shutdown())
```

Listing 6.1: Calculation of the average input in MPyC

[2]https://www.python.org/
[3]https://github.com/lschoe/mpyc/

MPyC implements the standard BGW protocol [BGW88] with the simplified multiplication protocol by Gennaro et al. [GRR98] from Section 3.1.1.1 for performing the secure computations with n parties. This means that MPyC works in the *honest majority* setting ($t < n/2$) and provides information-theoretic security against passive adversaries. In terms of communication, MPyC assumes the existence of pair-wise secure and authenticated channels. This means that an adversary is not able to eavesdrop on the communication between two honest parties. Furthermore, an adversary is not able to remove packets or 'add' packets by impersonating an honest party. To realise this, MPyC uses standard cryptographic tools such as SSL to encrypt the channels between the involved parties. Finally, all the protocols have been adapted to work in an *asynchronous* setting, meaning there are no predefined communication rounds.

6.2.1 Asynchronous Computation in MPyC

MPyC uses the `asyncio` library to perform asynchronous computations. For more information on `asyncio` and asynchronous computing in general, we refer back to Section 6.1. In most MPC protocols, some form of communication between the parties is required. However, having to wait for all the messages in a communication round before commencing the next round would result in a lot of idling. In fact, this means that we need to wait for the *slowest* party in every round. MPyC aims to avoid waiting as much as possible by executing all the computations asynchronously via special *MPyC coroutines*. As normal python coroutines, an MPyC coroutine responds immediately with a future. However, these futures are now *typed* to indicate what type is going to be received from the coroutine. These can even be more complex structures such as a nested list of types. These types, called `sectypes`, are essentially a "wrapper" around a future to indicate which type of element is expected in the future. By overloading the operators of these sectypes, a programmer can write regular arithmetic operations in the front-end of MPyC. Currently, MPyC supports 3 sectypes: finite fields, integers and fixed point numbers.

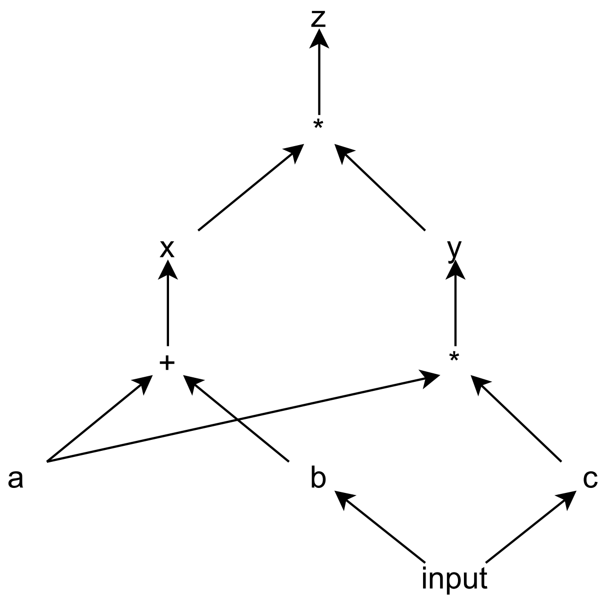
When programming arithmetic operations in the front-end, the order in which these operations are performed is only dependent on the underlying dependencies in the program. As as a result, the CPU can for example continue performing local operations while waiting for the reshares of a multiplication. To understand what is happening, consider a simple MPyC program and the corresponding expression tree in figure 6.2. Here, first the `secint` type is defined, which will be the type of the variables in the program. Next, all the parties are asked for a (private) input and another secint is instantiated that simply contains the value two. Next, *only* parties 1 and 2 get to input their private inputs into the computation. This means that they will use Shamir's secret sharing to distribute their input to the other parties. Here, b and c are `futures` of type `secint`, which will later contain the local Shamir shares of the parties running the program. Now, three additional `secints` will be instantiated by MPyC for x, y and z. Only in the last line the actual computation of z is forced by the output command.

Underwater, an expression tree as depicted in figure 6.2b is implicitly constructed. Here, an arrow denotes a dependency between two items. In this case, when z is output, the required dependencies are recursively gathered until the leafs are reached. What is important here is that x and y are mutually independent, meaning they can be computed in parallel. So if the party is, for example, waiting for its share of b but has already received c and is in possession of a, he can already start computing y. When both the futures for x and y are filled with their respective values, the program can compute z.

Since there is no guaranteed order in which the parties perform the operations, there is no guaranteed order in which shares are received from the other parties. However, it is important to keep track of which mes-

```
#Standard setup omitted

secint = mpc.SecInt()

input = secint(input("input: "))

a = secint(2)

b,c = mpc.input(input, [1,2])

x = a + b
y = a * c
z = x * y

print(mpc.run(mpc.output(z)))
```

(a) MPyC program



(b) Expression tree

Figure 6.2: An MPyC program and the corresponding expression tree

sages received from the other parties belong to which futures in the program. To this end, MPyC makes use of a so-called *program counter (PC)*. The idea of this is that every future that is encountered will be associated with a unique PC and a reference to this stored in a buffer. Now, every party accompanies the messages it is communicating with the correct program counters. When a message with a certain PC is received, it is added to the corresponding future or stored in the buffer in case the party did not reach that part of the program yet. To accomplish this, however, it is assumed that all the parties execute the exact same program on their own machines since otherwise the order in which futures are encountered are mixed between the parties.

6.2.2 Framework & Functionalities

The MPyC framework consists of a number of *core* classes which provide the necessities for performing the MPC, such as secret sharing, sectypes, logic and communication with the other parties. Furthermore, MPyC has a number of additional modules to provide extra functionality using the primitive operations from the core. For example, a module mimicking python's `random` module in a secure manner. For this work however, we will only describe the parts of the framework which are necessary to understand how we have extended the core of MPyC with player virtualisation. A schematic overview of the relevant part of the MPyC framework can be found in figure 6.3.

Here, an arbitrary MPyC program can be seen, which interacts with one instance of a `runtime`. The runtime is essentially the centre of MPyC and communicates with all the other modules in the framework. The runtime is also where all the operations of the sectypes are defined. Furthermore, the methods for inputting, outputting, resharing and gathering shares can be found. Every party runs one runtime on their machine. The `asyncoro` is responsible for handling all the asynchronous computation and communication

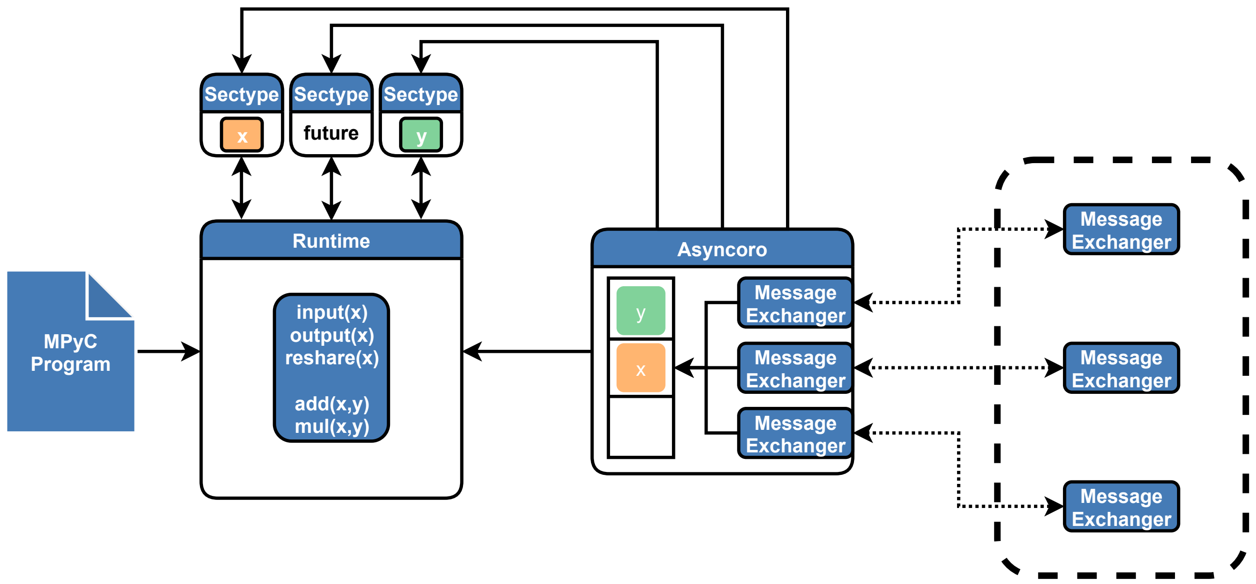


Figure 6.3: MPyC Framework

with the other parties. To this end, the asyncoro spawns n sub-classes called `message exchangers`, which all handle the communication with one message exchanger of another party. Furthermore, the asyncoro has a message buffer where incoming messages from the peers are stored, which are then added to the appropriate futures based on their program counter. The sectypes holding these futures are used by the runtime to perform the computations. In this case, we can see the sectypes of the shares of x and y which have already been filled and another sectype, say z, which has not yet been received and is thus still a future.

In terms of functionalities, the runtime implements a number of fundamental methods which are used by other methods to provide additional functionality. Let `mpc` be an instance of such a runtime and x, y and z instances of a `sectype`. The methods will be explained with three parties for comprehensibility. Note that in reality, MPyC can be run with any number of parties. Furthermore, these parties hold private inputs v_1, v_2 and v_3 respectively. The runtime has the following fundamental methods:

Input: the `rt.input` method is used to provide private input from the parties to the computation. In the front-end, encountering the line:

```
x,y,z = mpc.input(v_i)
```

corresponds to all the parties providing input to the computation, where x, y and z will hold one of the shares of v_1, v_2 and v_3 respectively. In the runtime, this leads to a Shamir sharing of the values appropriate with respect to the used `sectype`. If we let $f_a(x)$ be the polynomial sharing a, party P_i receives the point $(i + 1, f(i + 1))$ In this example, as by default, all the parties provide input but the method can also take a list of identifiers to specify a selection of parties to provide input. Note that each party needs to call the input method even if they will not provide input themselves since everyone needs a share of the inputs of the other parties (x, y and z in this case).

Output: In the front-end, we can write the following line:

```
result = mpc.output(x)
```

This line corresponds to opening x to all the parties. As with the input method, a list can be specified if only a subset of the parties should receive the output. To actually reconstruct the output, the parties send their share of x to all the other parties. Now, an incoming share x_i from party P_i is parsed as the point $(i + 1, x_i)$ on the polynomial. To get the correct value of x, the parties simply reconstruct this polynomial.

run: For example in the case of the output above, `result` will be a *future* object at first. To ensure the result contains the actual result of x before the program terminates, one should use the `mpc.run` method, which runs a given future or coroutine until the actual value is obtained like this:

```
result = mpc.run(mpc.output(x))
```

Linear combinations: Linear combinations of private inputs and public constants a, b and c can be programmed by writing in the front-end:

```
result = a * x + b * y + c * z
```

In the back-end, the overloaded operators for `sectypes` are invoked, which in turn use the runtime to decide what should be done. For example, the `mpc.add` method corresponds to a + operator in the front-end.

Multiplication: Finally, one can perform multiplications of shares in the front-end by writing:

```
result = x * y
```

In the back-end, this will trigger the `mpc.mul` method, which takes two arguments, say x and y. When the multiplication method is triggered, the parties perform a number of steps. First, they use `mpc.gather` to ensure x and y contain the actual content they should have. Next, they perform the local multiplication and reshare as in the multiplication protocol by Gennaro et al from Section 3.1.1.1.

6.3 Player Virtualisation in MPyC

As a stepping stone towards realising a covertly secure version of MPyC with public verifiability by applying the compilers from the previous section to the framework, we have designed and implemented the player virtualisation strategy in MPyC. One of the goals of MPC frameworks is to ensure that a programmer *using* the framework does not have to *deal* with the additional complexity of performing the computations in a secure manner. Performing the computations with a protocol with stronger security guarantees should thus not be harder to implement. Therefore, it is important that the front-end with which the programmer interacts remains the same. As an additional benefit to this, programs written in MPyC with passive security in mind can be run with covert security with only minor adjustments.

Furthermore, the covertly secure version of MPyC should provide the same functionalities as the passively secure version. If we view the core functionalities offered by MPyC such as input, additions etc. as an ideal functionality $\mathcal{F}_{MPyC}$, the runtime of MPyC can bee seen as a real-world protocol implementing $\mathcal{F}_{MPyC}$. New protocols such as our covertly secure protocol or even an actively secure version of MPyC in the future can thus be implemented by constructing a new `runtime`. Now, a programmer who has programmed the example program from listing 6.1 can change the level of security by simply changing the line:

```
from mpyc.runtime import mpc
```

into

```
from mpyc.covert_runtime import mpc
```

to execute the program with our runtime for covert security instead of the standard passively secure runtime.

In this work, we implemented the *one-to-many* player virtualisation strategy from Section 4.1 as a stepping stone towards achieving a covertly secure version of MPyC with public verifiability. With this in place, we can later apply cut-and-choose techniques on the virtual parties. We can verify the honest behaviour of a subset of virtual parties by giving the information required to derandomise those virtual parties. To do this securely, we need to ensure that this information will not give us any information on the inputs of the real parties.

6.3.1 What Function to Compute?

Suppose we have an MPyC program ran with n parties, which implements a functionality:

$$f(x^1, x^2, \dots, x^n) = (y_1, y_2, \dots, y_n)$$

Where party P_i has input x^i and receives output y_i. Furthermore, assume we would like to tolerate cn corruptions for some $0 \leq c < 1/2$. Call this threshold t. Furthermore, name the MPyC protocol implementing this functionality with passive security against cn corruptions Π_{MPyC}. In order for our compiler to work, we need to find a generic way to construct a related functionality f' suitable to be run with mn virtual parties, calculating the same function as f. Here, m depends on the desired deterrence rate ϵ. Assume a protocol, Π_{COV} implements the related functionality f'. Recall that in general this functionality takes an m-out-of-m secret-sharing of the input of each party and produces an mn-out-of-mn secret sharing of the output.

The function Π_{COV} these parties are computing has the following form:

$$\begin{aligned}\Pi_{COV}(x^1, x^2, \dots, x^n) &= f'(x_1^1, x_2^1, \dots, x_m^n) \\ &= f(\oplus_{j\in[m]} x_j^1, \oplus_{j\in[m]} x_j^2, \dots, \oplus_{j\in[m]} x_j^n)\end{aligned}$$

where

$$x^i = \bigoplus_{j=0}^{m} x^i_j$$

Additionally, Π_{COV} should protect against $t * m$ passive corruptions since one real corruption corresponds to m corrupt virtual parties.

In this case, we do not necessarily need an mn-out-of-mn secret-sharing of the output and can take a shortcut, revealing the outputs to the real parties directly instead of secret sharing them first. This can be done since MPyC releases the entire output at once and not, e.g., bit-by-bit. Therefore a potential adversary can not base its decision to abort the protocol on a partial output, breaking covert security as explained in [DOS20]. Because of this, we can take a shortcut and "undo" the additive secret sharing after the parties have exchanged the Shamir shares of their inputs. This way, the virtual parties hold only one share on the inputs of the real parties instead of m shares.

6.3.2 Augmented Functionalities

Next, we will describe how the functionalities of MPyC have been changed to construct a generic augmented protocol that implements the related functionality $\mathcal{F}_g$. Here we often refer to a "virtual party" performing a certain operation. Note however, that this essentially means that the real party to which the virtual party belongs performs the computation using the *context* that is associated with a certain "virtual party". This context simply consists of the input of that virtual party, the shares that are meant for the virtual party and the randomness which is being used by this virtual party.

Input: The first step to accomplish this is an augmented input protocol. In this augmented input protocol, first each real party P_i uses additive secret sharing to split it's input x^i into shares $x^i_1, x^i_2, \ldots, x^i_m$. These will be used as the input of virtual parties $\mathbb{V}^i_1, \mathbb{V}^i_2, \ldots, \mathbb{V}^i_m$. Similar to Π_{MPyC}, the virtual parties now use Shamir's secret sharing to generate shares for all the other virtual parties, including the ones belonging to the same real party. This means virtual party $\mathbb{V}^i_j$ will share it's input x^i_j as:

$$x^i_j = \left(x^i_{j,(1,1)}, \ldots, x^i_{j,(1,m)}, \ldots, x^i_{j,(n,1)}, \ldots, x^i_{j,(n,m)}\right)$$

where $\mathbb{V}^k_l$ receives share $x^i_{j,(k,l)}$. Furthermore, the threshold used to generate these shares increases from t to $t{*}m$. If $f(x)$ is a polynomial sharing an arbitrary secret, $\mathbb{V}^k_l$ receives the point $(k * m + l + 1, f(k * m + l + 1))$

After the sharing phase, each virtual party has received a share of the inputs of all the mn virtual parties. Concretely, virtual party $\mathbb{V}^i_j$ holds the shares $x^1_{1,(i,j)}, \ldots, x^1_{m,(i,j)}, \ldots, x^n_{1,(i,j)}, \ldots, x^n_{m,(i,j)}$. Now, the virtual parties will locally "reconstruct" their shares of the original inputs of the real parties by undoing the additive secret sharing to reduce the number of shares per virtual party to n shares, corresponding to the inputs of the n real parties. As an example, virtual party $\mathbb{V}^i_j$ can do this by computing:

$$\left(x^1_{(i,j)}, x^2_{(i,j)}, \ldots, x^n_{(i,j)}\right) = \left(\bigoplus_{k=1}^{m} x^1_{k,(i.j)}, \bigoplus_{k=1}^{m} x^2_{k,(i.j)}, \ldots, \bigoplus_{k=1}^{m} x^n_{k,(i.j)}\right)$$

In the original protocol, a real party P_i holds one share x_i^j on some private input x^j of P_j. In our augmented protocol, the real party now holds m shares, which can be represented as the vector $\vec{x} = [x_{(i,1)}^j, \ldots, x_{(i,m)}^j]$.

Linear operations: Recall that if in the original MPyC protocol the following line is executed in the front-end:

```
c = a * x
```

Where a is a public constant and x a share of the private input of some party in the protocol, the parties could simply multiply their share on x with a locally to obtain a correct share on c. In the augmented protocol, it turns out these operations can still be performed locally. Now, since x is in fact the vector $\vec{x}$, this operation corresponds to a scalar multiplication of a and $\vec{x}$ to obtain the share in $c = [ax_1, ax_2, \ldots, ax_m]$. Other linear operators such as addition work in a similar way as well.

Multiplication: In order to perform a multiplication of two shares x and y in the front-end like:

```
z = x * y
```

This means that in our augmented protocol, we essentially need to construct the Schur product of $\vec{x}$ and $\vec{y}$. However, simply calculating the Schur product results in shares of degree $2 * tm$ and thus we need to do a resharing for each virtual party like the original protocol. For this, each virtual party reshares it's result of computing $z = x * y$ to all the other virtual parties and received shares from all the other virtual parties. Now, they can locally compute a correct share on z by reconstructing the polynomial from the first $2*tm+1$ points they receive from the other virtual parties. Note that since the virtual parties are in fact controlled by one real party, the real party essentially knows m points up front.

Output: In order to output a certain value to the parties, we can follow a similar procedure as in the original protocol. Suppose we want to output a certain value y to all the parties. Each real party holds m shares on y corresponding to their own virtual parties. Now, if we want to output y to all the parties, they can simply reconstruct the value of y by sharing all of their m shares with each other and reconstruct the correct value from these points. Here, the share y_j^i of virtual party $\mathbb{V}_j^i$ is parsed as the point $(i*m+j+1, y_j^i)$ on the polynomial.

Note that here we take a shortcut and reconstruct the outputs of the real parties directly instead of first computing an mn-out-of-mn additive secret sharing and reconstructing from those output shares. As explained earlier, this is secure since MPyC reveals the output at once.

6.3.3 Implementation

As explained before, we implemented a new runtime class for our covertly secure protocol. All of our changes have been made in the back-end of MPyC which means the front-end of MPyC did not change at all, which is the only part the vast majority of MPyC users see. An updated overview of our extension of the MPyC framework with player virtualisation can be found in figure 6.4

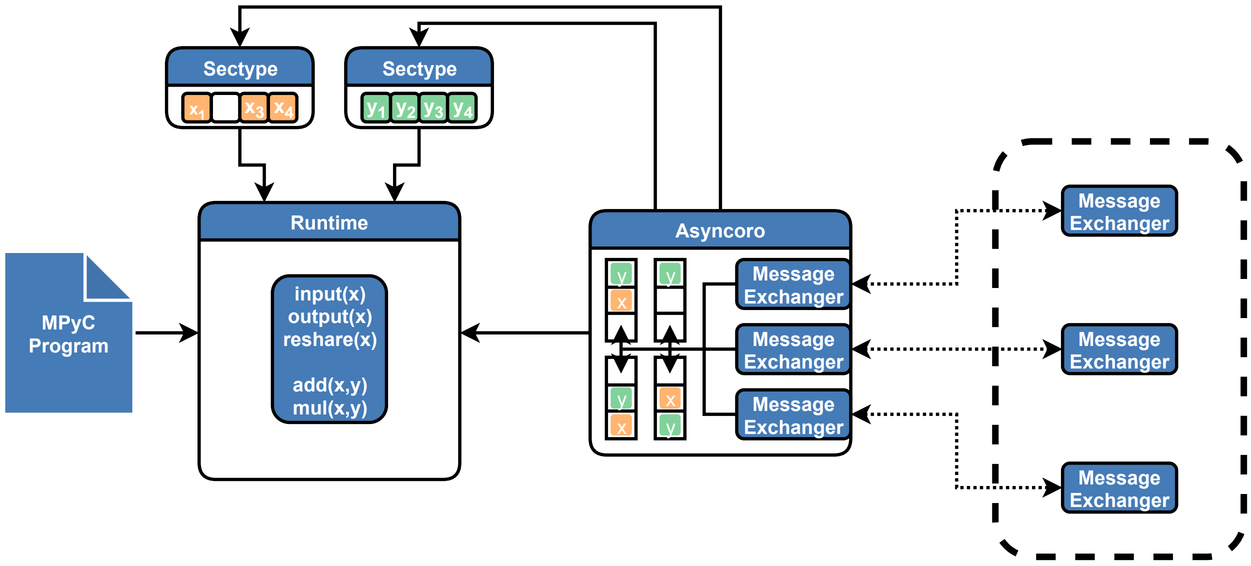


Figure 6.4: Covertly Secure MPyC Framework. Compared to the standard MPyC framework in Figure 6.3, the asyncoro now contains a buffer for each virtual party and the sectypes contain slots for each of the virtual parties.

6.3.3.1 Communication Complexity

For every time a share is sent from one party P_i to another party P_j in the original protocol, the new protocol needs to send one share from every $\mathbb{V}_k^i$ (virtual parties of P_i) to every virtual party of P_j: $\mathbb{V}_l^j$, $1 \leq k, l \leq m$. In practice, this means the real P_i needs to send $m * m$ shares to P_j. Luckily, MPyC already supported the option to communicate multiple shares in a single packet, which means that the *amount* of messages communicated could remain the same, but the size of the packets grows with a factor m^2. However, we still choose to send m messages containing the m shares with the same virtual party as destination. That way, we can make better use of the asynchronous nature of MPyC. Instead of having to wait for the larger packet to be transmitted, we can already start performing computations with the shares for one virtual party while waiting for the shares for the other virtual parties. Depending on the CPU power of the parties, the bandwidth and the latency of the network this can simply be altered where the other option is to send all the m^2 messages in one packet and let the other party perform more operations to assign the shares to their corresponding virtual parties.

The next challenges lie in the way in which we can keep track of the virtual parties to which the incoming messages belong. First, because we opt for an asynchronous approach, we can not simply assume that the first message is meant for the first virtual party. This has been solved by altering the *message exchanger* such that it not only communicates the program counter to which the shares belong, but also the virtual party for which the shares are meant. Secondly, all the shares sent for each share in the original protocol still correspond to the same message in the original protocol. Because of that, we choose to let them be communicated with the *same* program counter to avoid confusion. However, the mapping from futures to program counters in the message buffer of the asyncoro is unique, meaning that we can not simply store multiple shares with the same program counter in the buffer. Instead, we implemented the asyncoro to keep track of a message buffer *per* virtual party.

6.3.3.2 Types & Computations

As explained previously, MPyC knows a number of secure types or *sectype*s which form a wrapper around a future to support arithmetic operations on values that have not yet been received. In the original protocol, the line

```
x = mpc.input(v, [1])
```

would result in the input method returning a future immediately, which will later be filled with the share to be received from party 1. In the new protocol however, we do not receive one share but m shares of x for all our virtual parties. To accomodate this, we have implemented a new `sectype` called a `VirtualSecFld`. Instead of having m separate `SecFld` elements, a `VirtualSecFld` holds m slots to be filled with the shares of the m virtual parties. The interaction between the buffers and these `VirtualSecFld`s can also be seen in figure 6.4, where the shares for y have all been filled while we are still waiting for the share of x for the second virtual party to become available. For our proof-of-concept, we only considered finite field elements but we are certain this approach of representing the virtual parties as vectors will also work for the other sectypes MPyC knows and similar protocols in other frameworks.

In addition, the actual `runtime` has been altered in order to work with the new `asyncoro` and `VirtualSecFld`. In general, the runtime performs three important tasks: First, the runtime is responsible for taking the right values from the sectypes and defining the actions that should be taken when a certain operation on the sectype is programmed in the front-end. Second, the runtime is responsible for invoking the functions from, for example, the secret-sharing module, the module for finite field logic, etc. Lastly, the runtime is responsible for distributing the shares to the correct functions in the asyncoro and interpreting the messages received from the other parties.

For the first point, the notable difference is in the non-linear operations, where we often require some form of communication between the parties in the protocol. As explained in the communication part, doing this with player virtualisation results in an overhead of $O(m^2)$. However, we implemented it in such a way that only the shares from the virtual parties of one real party meant for *one* of the virtual parties of the receiver are grouped in one packet. If we would group all the messages for all the virtual parties of a real party together in a single packet, this would save us m packets but would cause more idle time at the CPU of the receiver. Due to the asynchronous nature of MPyC, it is very convenient for the overall run-time of our implementation to send m packets with m shares instead of one packet with m^2 shares. With this approach, we can already start performing computations for a virtual party who has received enough shares while we wait for the shares of the other virtual parties to arrive. This way, we can keep the CPU busy with useful work and reduce the practical overhead of the non-linear operations of the player virtualisation protocol. For the linear operations, we simply map the operations pairwise to all the elements in the vectors of the shares.

With our approach, the second and third task actually did not require significant changes to the runtime implementation. For the second point, sharing a secret value x now meant we did not invoke the splitting method with the number of parties n and the threshold t anymore. Instead, we now invoke the split method to split into $m*n$ shares with a threshold $t*m$. Furthermore, the invocations of the reconstruction methods for the resharing and the output phase now use the points as described in Section 6.3.2. For the third point, the runtime mainly has to perform some extra work in order to structure the messages that are received from the other parties but we believe the complexity introduced by this is insignificant compared to the other parts of the protocol.

Chapter 7

Analysis

In this chapter the performance of our compiler will be analysed. To this end, an analysis of the complexity and functionality of the design presented in Chapter 5 will be done first. After that, we will look at the practical performance of the implementation we have made in Chapter 6.

7.1 Theory

7.1.1 Deterrence Rate

The most notable difference between the watchlist approach in [DOS20], and the shared coin toss (SCT) approach of our work and [Fau+21; SSS21], lies in the way the opened executions are chosen and the resulting deterrence rates. Recall that in the watchlist approach, each party chooses the executions to open *individually*, resulting in each party picking a different, random watchlist. To now ensure that one execution remains closed, the sizes of these watchlist and the resulting deterrence rates become relatively low. On the other hand, in the shared coin toss approach the set of opened executions is shared across all the parties. By now ensuring that one execution is left out of this set, higher deterrence rates for the same number of repetitions can be obtained. In fact, the resulting deterrence rates are optimal.

Note that for a number of repetitons k and a number of opened executions t, the deterrence rate equals $\epsilon = \frac{t}{k}$. In [DOS20], To ensure (at least) one unopened execution in the watchlist approach, in the case of two parties, the size t of the watchlist is then chosen as $t < \frac{k}{2}$. However, when this approach is extended to the multi-party setting ($n > 2$), it follows that the size of these watchlists is bounded by $t < \frac{k}{n}$. In the SCT approach, all the parties pick *the same* executions to check, which results in a guaranteed $t = k - 1$. To reach higher deterrence rates in the watchlist approach, in [DOS20] it is argued that the parties can pick a watchlist of size $t > \frac{k}{n}$ and, with a constant probability, let the protocol run *fail* in case all executions have been opened by at least one of the parties. If the protocol run fails, a new run of the entire protocol is required. This way, the same deterrence rates can be obtained as in the SCT approach at the cost of the probability that we require additional runs. A comparison of the deterrence rates ϵ of the SCT approach and the standard watchlist approach ϵ' is given in Table 7.1. Furthermore, the expected number of runs in the watchlist approach to reach the same deterrence rates as the SCT approach is given. For a more detailed overview of how the expected amount of runs can be calculated, we refer to the analysis done by [Fau+21].

n	k	SCT	Watchlist approach		
		ϵ	ϵ'	or	runs
2	2	1/2	-		2
	3	2/3	1/3		3
	10	9/10	4/10		10
3	2	1/2	-		4
	4	3/4	1/4		16
	10	9/10	3/10		100
5	2	1/2	-		16
	6	5/6	1/6		1296

Table 7.1: Maximum deterrence rate or expected number of runs of the time-lock approach and the watchlist approach when ran with n parties and k executions [Fau+21]

Note that this fail mechanism only works for the input-independent compilers since allowing a protocol run to fail in the input-dependent settings implies that all the virtual parties of a real party would be opened, leaking the private input of the real party. Therefore, it must hold that $(n-1)*t < k$, and thus the deterrence rate is bounded by $\epsilon < \frac{1}{(n-1)}$ if the corruption threshold is maximal, i.e., $t = n - 1$. As we have shown in Chapter 4, this upper bound on the deterrence rate can be increased in case we assume a lower amount of corruptions, but this still does not reach the deterrence rates possible with the SCT approaches. Since the SCT approach can guarantee that always one virtual party remains hidden, they can reach a guaranteed deterrence rate of $\epsilon = \frac{k-1}{k}$.

7.1.2 Complexity

The main differences in terms of computation- and communication complexity of our compiler and the works of [Fau+21; SSS21] lie in the steps taken *after* the passively secure protocol has been executed. The goal of these steps is to ensure the parties obtain the randomness seeds used by all the other parties without the possibility of a detection-dependent abort by the adversary. In our case, this is done by the seed distribution with the publicly verifiable secret sharing scheme, the execution of Π_{open} and the possible execution of $\Pi_{\text{reconstruct}}$. In [Fau+21], the same goal is achieved by the maliciously secure puzzle generation.

Computation complexity The computation complexity of our approach is largely determined by the complexity of the underlying PVSS in the distribution of all the seed openings in step (3) of $\texttt{COMP}_{\text{PVC}}$, the verifications of *all* the seeds in step (1) of Π_{open} and the decryption, verification and reconstructions in $\Pi_{\text{reconstruct}}$, which is only executed in case any party does not receive all seed decommitments. For concreteness, we will investigate the amount of group exponentiations required if we instantiate our compiler with the state-of-the-art PVSS of Janbaz et al. [Jan+20], where the amount of group exponentiations required for each of the aforementioned steps are reported. An overview of the amount of group exponentiations for our protocol can be found in table 7.2. *Distributing* a single secret in the PVSS scheme requires $n + 1$ exponentiations, while calculating the corresponding proof requires $3 \cdot n + 1$ exponentiations. Since each party needs to distribute k seed shares, the total amount of exponentiations is $(4 \cdot n + 2) \cdot k$. *Verifying* all the shares distributed for a single secret requires $n \cdot (3 + t + 1)$ exponentiations. Here, t is the reconstruction threshold which is $\frac{n}{2}$, rounded down to the closest integer. Since each party needs to verify the seeds used in *all* the executions by *all* the parties except for themselves, this becomes

$(n \cdot (3 + \frac{n}{2} + 1)) \cdot (k \cdot n - k)$. In the optimistic case, each party now publishes all the seed openings and we are done. In the pessimistic case, some parties did not receive some of the seed openings, which will be reconstructed using $\Pi_{\text{reconstruct}}$. Say m distinct seed openings are missing in total, regardless of the parties that are missing the shares. Each party now has to *decrypt* its share of each of the m seed openings. A single decryption costs 1 exponentiation. Together with the 2 exponentiations required for computing the correct decryption proof, this results in a total of $3 \cdot m$ exponentiations for each party. Finally, each party missing some amount of shares e needs to verify the decrypted shares it receives and reconstruct the seed opening. Verifying a single share costs 4 exponentiations while reconstructing the seed opening requires $t = \frac{n}{2}$ exponentiations. In the worst case, all the n shares need to be validated in order to find $t + 1$ valid shares for each missing seed opening, resulting in a total of $(4 \cdot n + \frac{n}{2}) \cdot e$ exponentiations. In the best case, the first $t + 1$ verified shares are all valid, in which case only $4 \cdot (\frac{n}{2} + 1) + \frac{n}{2}) \cdot e$ exponentiations are required.

Step	Exponentiations
Distribution	$(4 \cdot n + 2) \cdot k$
Verification	$(n \cdot (3 + \frac{n}{2}) + 1) \cdot (k \cdot n - k)$
Decryption	$3 \cdot m$
Reconstruction	$(4 \cdot n + \frac{n}{2}) \cdot e$

Table 7.2: Overview of the amount of group exponentiations required in our protocol. Here, n is the number of parties and k the amount of repetitions. In the pessimistic case, m is the total number of missing shares while e is the amount of shares missing by a single party.

Common modular exponentiation algorithms cost $O(log(m)^{1.5} \cdot log(n))$ bit operations for a modulus m and a power n. In the PVSS of [Jan+20], both m and n have a bit length of some security parameter λ. Therefore, we can say the amount of bit operations per exponentiation is $\lambda^{2.5}$.

For the time-lock puzzle approach, [Fau+21] approximate a total number of AND gates for the maliciously secure puzzle generation functionality of:

$$\begin{aligned}&(n-1) \cdot (11|k| + 22|N| + 12)\\ &+nk \cdot (4|k| + 2\lambda + 755)\\ &+192|N|^3 + 112|N|^2 + 22|N|\end{aligned}$$

.

Here $|x|$ denotes the bit length of x and N is the RSA modulus for the time-lock puzzle. As can be seen, the asymptotic behaviour in n and k is similar to our construction, but the biggest drawback of their approach is the cubic and quadratic complexity in the length of the RSA modulus, where typical RSA modulo sizes nowadays are 1024, 2048 or even 4096 bits. For a typical security parameter of 128 bits, we believe our construction is thus much simpler compared to the approach of [Fau+21].

Per example, the total number of exponentiations required for $n = 5, t = 2, k = 2(\epsilon = \frac{1}{2})$ and $m = e = 1$ in our approach with a security parameter of 128 bits becomes 285. This results in $285 \cdot 128^3 \approx 5 \cdot 10^7$ bit operations. On the other hand, the number of AND gates in [Fau+21] with an RSA modulus of only 1024 bits (which is not very secure nowadays) becomes in the order of 10^{11}. Furthermore, if all the parties behave honestly, our approach has the benefit that the decryption and reconstruction steps can be skipped, further reducing the complexity. Comparing our work to the time-lock approach of [SSS21] is harder, but we believe the results will be similar to [Fau+21] due to the similarity of the two works.

Communication complexity We measure the communication complexity of our approach as the amount of field elements that are communicated between a pair of parties in the protocol. The total amount of field elements sent between the parties in each of the step is reported in Table 7.3. Note that normally, publicly verifiable secret sharing schemes require some form of public bulletin board to ensure that each of the parties is guaranteed to see the same encrypted shares distributed by a dealer and that all the parties receive the same decrypted shares. To this end, we use a broadcast like [Dam+13] in which first all the messages are sent to all the parties and after that pairwise comparisons are performed on the hashes of the received messages. This ensures that all the parties receive the same messages.

Step	**Field Elements**
Distribution	$(4 \cdot n + 1) \cdot k \cdot (n - 1) \cdot 2$
Optimistic case	$2 \cdot k \cdot (n - 1)$
Pessimistic case	$3 \cdot m \cdot (n - 1) \cdot 2 + e$

Table 7.3: Overview of the amount of field elements communicated between the parties in our protocol. Here, n is the number of parties and k the amount of repetitions. In the pessimistic case, m is the total number of missing shares.

Distributing a single secret in the PVSS of [Jan+20] requires publishing $3n + 1$ field elements for the proof and n for the encrypted shares themselves. Implementing the public bulletin board using the secure broadcast, these messages are first sent to the other $n - 1$ parties and then hashes of the received messages again to all the parties, hence the factor 2. Furthermore, this needs to be done for each of the k seed openings. After that, in the optimistic case, the party simply sends two seed openings and a signature on each of these seed openings to the other parties, resulting in $2 \cdot k \cdot (n - 1)$ field elements communicated in total. In the pessimistic case, a number of m seed openings are still missing. In this case, e missing messages are sent by a party missing e seed openings. After that, for a single secret, all the parties need to broadcast again their decrypted share and 2 elements for the decryption proofs. This results in a total of $3 \cdot m \cdot (n - 1) \cdot 2$ field elements. Looking at the synchronous communication model, the broadcasts cost 2 communication rounds and hence our protocol requires a constant number of 3 communication rounds in the optimistic case. In the pessimistic case, this increases to 6 communication rounds.

The communication costs of the time-lock approach from [Fau+21] can be estimated based on the number of AND gates in the circuit of the puzzle generation functionality. They mention the general-purpose actively secure MPC protocol of [YWZ20] as a possible protocol for implementing their circuit. Here, an estimated amount of 193 bytes of bandwidth is needed per party for a single multiplication triple, resulting in approximately $193 \cdot 10^{11}$ bytes required to compute the entire circuit when run with the same parameters as used in the previous paragraph. Furthermore, this MPC protcol runs in a constant number of 14 communication rounds. For comparison, if we are pessimistic, our approach requires communicating around 400 field elements per party. Assuming the size of the field elements in our protocol is 32 bits, this would result in a total number of 1600 bytes communicated in only 6 communication rounds.

Protocol execution phase As can be seen in the above analysis, the computation- and communication complexity of the additional protocols for seed generation, execution opening and potential reconstruction are independent of the size of the circuit to be computed during the protocol execution phase. Therefore, the complexity of these protocols becomes less important if we compile passively secure protocols with a large circuit sizes. As our input-independent compiler simply executes the passively secure protocol k times, the computational complexity of the protocol execution phase increases with a factor k. Since each party needs to receive each message sent during the protocol executions, the communication complexity in a single repetition of the passively secure protocol increases with a factor $n - 1$ since each message in the original protocol now needs to be sent to all the other parties. These protocol executions are all independent of each other, which means we can run them in parallel. Therefore in practice, the running time of the passively secure protocol does not have to increase if the parties have enough computational power and the network with which they are connected has enough bandwidth. In the synchronous communication model, one might say that the round complexity of the passively secure protocol is preserved.

7.2 Practice

To get a better understanding of the performance of covert security compared to passive and active security in practice, we have benchmarked our implementation of MPyC with player virtualisation and the plain version of MPyC with passive security. Furthermore, we have compared this to hypothetical executions times for an actively secure version of MPyC, which we have derived from the predecessor of MPyC called ViFF [Gei10].

7.2.1 Test Setup

The benchmarks have been performed by simulating the parties locally on a server with 20 Intel Broadwell CPUs running at 2.4GHz. Furthermore, this server boasts 63GB of RAM. We have measured the execution time of a multiplication in all three implementations. The execution time of a single multiplication is taken as the average of 100.000 multiplications, where we again took the average time of all the parties. Since the protocol is asynchronous, it might happen that some of the parties are done *slighly* earlier, but in practice this made almost no difference since the latency of the communication between the parties is so low compared to the time spent on the computations. The execution time of multiplications is a good indicator of the overall performance of an MPC protocol since these are non-linear operations, which are usually the bottleneck of MPC protocols.

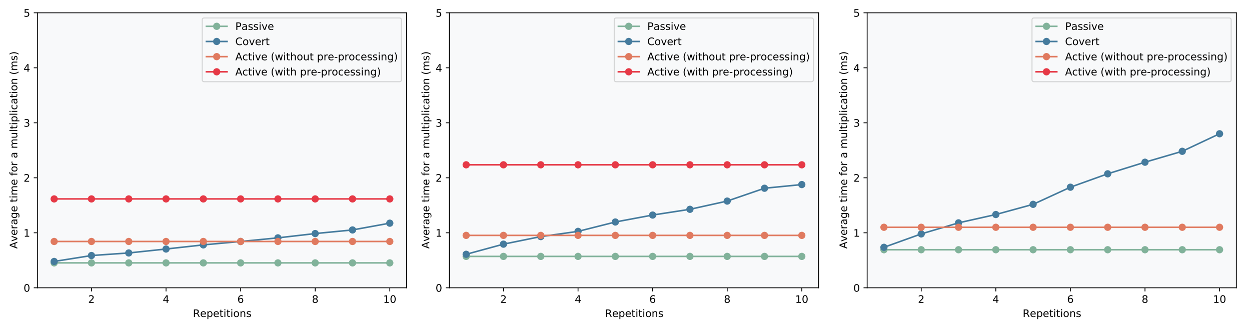
Since MPyC itself does not have an implementation of active security, we have looked at the predecessor of MPyC called ViFF, which does have an actively secure version on top of the standard passive security. Here, we looked at the reported results in terms of execution times for passive and active security in ViFF from [Gei10] and applied the same overhead to our measurements of passively secure MPyC. Since the frameworks are very similar in terms of structure and functionality, we believe this yields a fair comparison.

In these frameworks, multiplications with passive security are performed by a local multiplication followed by a reshare, similar to the approach of Gennaro at al. from Section 3.1.1.1. The actively secure implementation uses Beaver's circuit evaluation approach described in Section 3.2, with an offline phase for generating the triples. As explained in the previous chapter, our covertly secure version performs the computations in a similar fashion to the passive version, but does so with virtual parties. All of these approaches have been altered to run asynchronously. Note that for our covertly secure version, we only take the online phase of the protocol into account. However, the other parts of our compiler for covert security with public verifiability are independent of the circuit size and thus amortise over large circuits.

7.2.2 Results

n	**t**	**Passive**	***Active***		**Covert**		
		On	***On***	***On + Off***	**k = 2**	**k = 5**	**k = 8**
4	1	0.5ms	1.0ms	1.8ms	0.6ms	0.8ms	1.0ms
			(x1.9)	**(x3.6)**	**(x1.2)**	**(x1.6)**	**(x2.0)**
7	2	0.6ms	1.0ms	2.3ms	0.8ms	1.2ms	1.6ms
			(x1.7)	**(x3.9)**	**(x1.3)**	**(x2.0)**	**(x2.7)**
10	3	0.7ms	1.1ms	5.5ms	1.0ms	1.5ms	2.3ms
			(x1.6)	**(x7.9)**	**(x1.4)**	**(x2.1)**	**(x3.3)**
13	4	0.8ms	1.2ms	7.8ms	1.1ms	2.0ms	3.0ms
			(x1.5)	**(x9.8)**	**(x1.4)**	**(x2.5)**	**(x3.8)**
16	5	0.9ms	1.3ms	8.8ms	1.3ms	2.6ms	3.9ms
			(x1,4)	**(x9.8)**	**(x1.4)**	**(x2.9)**	**(x4.3)**
19	6	1.1ms	1.4ms	11.2ms	1.6ms	3.3ms	5.2ms
			(x1.3)	**(x10.2)**	**(x1.5)**	**(x3.0)**	**(x4.7)**

Table 7.4: Execution time of a single multiplication in passively secure MPyC, actively secure MPyC (split in an offline and online phase) and the online phase of our covertly secure version of MPyC for various deterrence rates.



(a) n=4, t=1 (b) n=7, t=2 (c) n=10, t=3

Figure 7.1: Benchmark of the average time for a multiplication (of 100.000 runs) for a number of repetitions. This has been plotted for 4, 7 and 10 parties with thresholds 1,2 and 3 respectively. Note that for $n = 10$, the line for active security with pre-processing is at 5.5ms and did not fit in the figure.

The results of these benchmarks can be found in Table 7.4. Here, the execution times for passive security, active security and covert security with 2, 5 and 8 repetitions ($\epsilon = \frac{1}{2}, \frac{4}{5}, \frac{7}{8}$) are presented. Visual representations for some of these results can also be found in Figure 7.1. Furthermore, the actively secure execution times are split in an offline phase which can be ran before the computations take place and an online phase in which the computations take place. ViFF provided two ways for generating these triples with active security, where one was significantly faster for lower numbers of parties while the other scaled better. In Table 7.4, only the fastest of the two for a given number of parties is taken into account.

Because of the split in an off- and online phase, we see that the introduced relative overhead of the online phase of the actively secure protocol actually goes down as the number of parties increases. This is due to the fact that the complex operations can be pushed to the offline pre-processing. However, it can be observed that both phases of active security combined introduce a lot of overhead. On the other hand, in the passively secure implementation, all the complexity is in the resharing in the online phase. This is similar for covert security, which introduces a lot less overhead compared to the passively secure protocol. For a deterrence rate of $\frac{1}{2}$, we observe that covert security is even faster than simply the online phase of active security up until 13 parties. For a larger number of parties, the circuit evaluation approach outshines the resharing approach. However, looking at the overall overhead of active security, we observe that even for large deterrence rates, the covertly secure protocol is about 50% faster. Furthermore, the additional overhead introduced by covert security grows slower for larger numbers of parties compared to to the total overhead of active security.

For protocols with a large number of parties, the expected execution times of the online phase of active security are lower than for covert security, even with low deterrence rates. Therefore, we should not focus on compiling the online phase of the protocol but find a pre-processing protocol with passive security, apply our input-independent compiler to that and then simply run an actively secure online phase. As proven in [DOS20], a protocol with a covertly secure pre-processing phase with public verifiability and an actively secure online phase constitutes an overall protocol with covert security and public verifiability. As the executions of the passive protocol in our input-independent compiler are completely independent of each other, we can run all of them in parallel with enough computation power of the parties. Therefore, the slowdown of this approach is expected to be minimal in practice. For large circuits, the vast majority of the complexity this strategy is in the pre-processing phase. Therefore, a speedup of, e.g., 30% in the pre-processing phase results in a speedup of almost 30 % in the overall protocol as well. All in all, our compiler has the potential to improve the performance of these protocols by a lot in practice.

Chapter 8

Conclusion & Discussion

8.1 Conclusion

In this thesis we have shown the potential of the notion of covert security with public verifiability for MPC protocols in the context of compilers. To this end, we have illustrated the power of these compilers in transforming arbitrary protocols to covertly secure protocols with public verifiability.

Furthermore, we have presented a new design for such a compiler, which uses publicly verifiable secret sharing for transforming a passively secure protocol to a protocol with covert security and public verifiability. Our compiler treats the passively secure protocol in a *black-box* manner, meaning our compiler will work for new MPC protocols with passive security in the future as well. This design follows the shared coin-toss approach which yields much higher deterrence rates compared to earlier ideas and also appeared in two other works on such compilers in 2021. Compared to these works, we have shown that our compiler can be instantiated with less assumptions on the network and much simpler building blocks. Both the computation- and the communication complexity of the opening mechanisms of our protocol are orders of magnitude lower compared to the work [Fau+21]. We believe our compiler is therefore more practical and a good step forward in rolling out compilers in practice.

Finally, we have shown the potential compilers in general with covert security and public verifiability by means of a proof-of-concept implementation in MPyC. This implementation constructs a related functionality for the virtual parties in a generic way. Such a related functionality is required by all such compilers known at the time of writing this thesis. To the best of our knowledge, the idea of undoing the additive secret-sharing as a first step of the virtual parties has not been presented before in this context. Benchmarks of this implementation show that the notion of covert security indeed leads to more efficient protocols compared to active security. In our theoretical construction for a new compiler as well as the implementation, the deterrence rate of the covertly secure protocol can be chosen arbitrarily, which provides a useful trade-off between security and efficiency.

8.2 Discussion & Future Work

This work has demonstrated the power of publicly verifiable secret sharing schemes in the context of compilers, but did so while making multiple assumptions that can be researched. In general, future work boils down to researching how the assumptions have influenced the design and performance of our compiler and investigate whether our techniques can be applied in situations where these assumptions may not be applicable. We will sketch some of the research directions we think will be interesting as follow-up work.

Asynchronous communication. First of all, we think and sketch that our compiler *could* work in the asynchronous communication model, but did not work this out in detail. Especially the possibility of being able to continue the protocol execution *even* if some of the parties stop responding could prove difficult for our design. These challenges might add to the complexity of our approach, but we believe they can be overcome considering that the asyncronous nature of MPyC and ViFF was already realised using the honest majority assumption.

Use of PVSS in practice. We sketch how we can implement the PVSS using a secure broadcast, but have not implemented it in practice. While we are confident this should work since a similar approach has been used in earlier works on MPC protocols, the actual implementation could prove to be non-trivial.

Honest majority. Perhaps the most important assumption of our compiler is the assumption that the majority of the participants in the resulting protocol behave honestly. If this is not the case, the security of our execution opening protocols completely breaks. In this case, an adversary might cause wrong seeds to be reconstructed and incriminate honest parties at will. Making our compiler secure against a dishonest majority is not a trivial task is the biggest drawback of our compiler compared to related works. Researching whether a PVSS can also be used to design a compiler secure against a dishonest majority would therefore be very interesting as a follow-up work.

Implementation. On the practical side, our proof-of-concept implementation currently only implements the player virtualisation strategy. To be used with covert security in practice, the PVSS, signature scheme, opening and reconstruction protocols also need to be implemented. Therefore, obvious future work is to implement and evaluate the entire compiler design presented in this work. While all the building blocks and cryptographic protocols introduced in this work have been built and used successfully before, we do not yet know the practical performance of our solution. Furthermore, our benchmarks have been performed on a single server, meaning the 'network latency' is very low. Therefore, it would be interesting to see how our solution performs in more realistic LAN or WAN networks. Lastly, the solution for finding a generic functionality for the virtual parties does not always work since it is not secure *in general*. While it works for MPyC because the entire output is released at once, this will prove to be an obstacle when trying to apply our compiler to more ad-hoc MPC protocols.

Complexity analysis. The complexity of our compiler and the related works should be researched further. Especially the asymptotic behaviour of our execution opening protocols for a large number of parties or a large number of missing seed openings need to be investigated since the complexities include some quadratic relations on these parameters. Compared to the related works, the complexity analysis of our approach look much better. We believe this is due to the circuit design of these works being rather naive. We believe their current complexities do not do justice to the potential of their solution, which can be improved with more thought. An intuition for a more efficient solution has already been sketched in

[Fau+21]. Here, the idea is to remove a complex portion of the puzzle generation step from the actively secure part of the circuit, removing the costly cubic dependency on the RSA modulus size. Currently, this solution is not secure but we believe this can be done in a secure manner in future research.

Combination with blockchain. As another line of future research, we think a combination of our work with blockchains could be interesting. A blockchain is a way of storing information (like a database) but in a distributed fashion, removing the need to trust a single party just like MPC. After all the parties agree on the data, the data is stored and becomes *immutable*[1], meaning it is impossible to change it after it has been stored. Furthermore, *anyone* participating in the blockchain may see the data stored on the blockchain. We identify two meaningful ways to combine blockchains with this work.

First, a blockchain could serve as the public bulletin board which is required for the PVSS. Here the parties can store the encrypted seed shares, decrypted seed shares and the proofs all on the blockchain. This guarantees that all the parties agree on the stored values. This seems like a more robust solution compared to the secure broadcast while possibly also reducing communication costs.

Secondly, another feature of blockchains is that they are *programmable*. Via so-called *smart contracts*, arbitrary programs may be stored on the blockchain. Example use cases of this are betting, where it is now impossible for a participant to withdraw, or for insurance contracts to automatically resolve a dispute. We believe smart contracts could be useful in combination with our work to automatically punish a detected cheater by running our `Judge` algorithm inside a smart contract. A party could then invoke the smart contract with a certificate, which automatically punishes the cheater financially if the certificate indeed proofs that a party cheated.

[1]This is true in the sense that an attacker can not alter the contents of the blockchain later on. However, if all the parties agree, the contents could still be changed.

Bibliography

[Abs+21] Mark Abspoel et al. “An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings”. In: *ACNS (2)*. Vol. 12727. Lecture Notes in Computer Science. Springer, 2021, pp. 122–152.

[AL07] Yonatan Aumann and Yehuda Lindell. “Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries”. In: *IACR Cryptol. ePrint Arch.* 2007 (2007), p. 60.

[AO12] Gilad Asharov and Claudio Orlandi. “Calling Out Cheaters: Covert Security with Public Verifiability”. In: *ASIACRYPT*. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 681–698.

[Ara+16] Toshinori Araki et al. “High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority”. In: *CCS*. ACM, 2016, pp. 805–817.

[Ara+17] Toshinori Araki et al. “Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 843–862.

[Bea91] Donald Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *CRYPTO*. Vol. 576. Lecture Notes in Computer Science. Springer, 1991, pp. 420–432.

[Ben+11] Rikke Bendlin et al. “Semi-homomorphic Encryption and Multiparty Computation”. In: *EUROCRYPT*. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 169–188.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)”. In: *STOC*. ACM, 1988, pp. 1–10.

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. “Perfectly-Secure MPC with Linear Communication Complexity”. In: *TCC*. Vol. 4948. Lecture Notes in Computer Science. Springer, 2008, pp. 213–230.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. “The Round Complexity of Secure Protocols (Extended Abstract)”. In: *STOC*. ACM, 1990, pp. 503–513.

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. “A General Composition Theorem for Secure Reactive Systems”. In: *TCC*. Vol. 2951. Lecture Notes in Computer Science. Springer, 2004, pp. 336–354.

[Can01] Ran Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *FOCS*. IEEE Computer Society, 2001, pp. 136–145.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. “Multiparty Unconditionally Secure Protocols (Extended Abstract)”. In: *STOC*. ACM, 1988, pp. 11–19.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. "Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result". In: *CRYPTO*. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 87–119.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. "General Secure Multi-party Computation from any Linear Secret-Sharing Scheme". In: *EUROCRYPT*. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 316–334.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. "Multiparty Computation from Threshold Homomorphic Encryption". In: *EUROCRYPT*. Vol. 2045. Lecture Notes in Computer Science. Springer, 2001, pp. 280–299.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[Cha+19] Harsh Chaudhari et al. "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction". In: *CCSW@CCS*. ACM, 2019, pp. 81–92.

[Chi+18] Koji Chida et al. "Fast Large-Scale Honest-Majority MPC for Malicious Adversaries". In: *CRYPTO (3)*. Vol. 10993. Lecture Notes in Computer Science. Springer, 2018, pp. 34–64.

[Cho+85] Benny Chor et al. "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults (Extended Abstract)". In: *FOCS*. IEEE Computer Society, 1985, pp. 383–395.

[Dam+09] Ivan Damgård et al. "Asynchronous Multiparty Computation: Theory and Implementation". In: *Public Key Cryptography*. Vol. 5443. Lecture Notes in Computer Science. Springer, 2009, pp. 160–179.

[Dam+12] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *CRYPTO*. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 643–662.

[Dam+13] Ivan Damgård et al. "Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits". In: *ESORICS*. Vol. 8134. Lecture Notes in Computer Science. Springer, 2013, pp. 1–18.

[Dam+17] Ivan Damgård et al. "The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited". In: *CRYPTO (1)*. Vol. 10401. Lecture Notes in Computer Science. Springer, 2017, pp. 167–187.

[Dam98] Ivan Damgård. "Commitment Schemes and Zero-Knowledge Protocols". In: *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998*. Ed. by Ivan Damgård. Vol. 1561. Lecture Notes in Computer Science. Springer, 1998, pp. 63–86. DOI: `10.1007/3-540-48969-X\_3`. URL: `https://doi.org/10.1007/3-540-48969-X%5C_3`.

[DGN10] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. "From Passive to Covert Security at Low Cost". In: *TCC*. Vol. 5978. Lecture Notes in Computer Science. Springer, 2010, pp. 128–145.

[DOS18] Ivan Damgård, Claudio Orlandi, and Mark Simkin. "Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings". In: *CRYPTO (2)*. Vol. 10992. Lecture Notes in Computer Science. Springer, 2018, pp. 799–829.

[DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. "Black-Box Transformations from Passive to Covert Security with Public Verifiability". In: *CRYPTO (2)*. Vol. 12171. Lecture Notes in Computer Science. Springer, 2020, pp. 647–676.

[DZ13] Ivan Damgård and Sarah Zakarias. "Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing". In: *TCC*. Vol. 7785. Lecture Notes in Computer Science. Springer, 2013, pp. 621–641.

[Eer+20] Hendrik Eerikson et al. "Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security". In: *ITC*. Vol. 163. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 5:1–5:24.

[Fau+21] Sebastian Faust et al. "Generic Compiler for Publicly Verifiable Covert Multi-Party Computation". In: *EUROCRYPT (2)*. Vol. 12697. Lecture Notes in Computer Science. Springer, 2021, pp. 782–811.

[FH93] Matthew K. Franklin and Stuart Haber. "Joint Encryption and Message-Efficient Secure Computation". In: *CRYPTO*. Vol. 773. Lecture Notes in Computer Science. Springer, 1993, pp. 266–277.

[FH96] Matthew K. Franklin and Stuart Haber. "Joint Encryption and Message-Efficient Secure Computation". In: *J. Cryptol.* 9.4 (1996), pp. 217–232.

[Fur+17] Jun Furukawa et al. "High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority". In: *EUROCRYPT (2)*. Vol. 10211. Lecture Notes in Computer Science. 2017, pp. 225–255.

[Gei10] Martin Geisler. "Cryptographic protocols: theory and implementation". In: *PhD thesis, University of Aarhus* (2010).

[Gen+14] Daniel Genkin et al. "Circuits resilient to additive attacks with applications to secure computation". In: *STOC*. ACM, 2014, pp. 495–504.

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *STOC*. ACM, 2009, pp. 169–178.

[GMS08] Vipul Goyal, Payman Mohassel, and Adam D. Smith. "Efficient Two Party and Multi Party Computation Against Covert Adversaries". In: *EUROCRYPT*. Vol. 4965. Lecture Notes in Computer Science. Springer, 2008, pp. 289–306.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *STOC*. ACM, 1987, pp. 218–229.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. "Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography". In: *PODC*. ACM, 1998, pp. 101–111.

[HM00] Martin Hirt and Ueli M. Maurer. "Player Simulation and General Adversary Structures in Perfect Multiparty Computation". In: *J. Cryptol.* 13.1 (2000), pp. 31–60.

[Hon+19] Cheng Hong et al. "Covert Security with Public Verifiability: Faster, Leaner, and Simpler". In: *EUROCRYPT (3)*. Vol. 11478. Lecture Notes in Computer Science. Springer, 2019, pp. 97–121.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. "Low Cost Constant Round MPC Combining BMR and Oblivious Transfer". In: *ASIACRYPT (1)*. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 598–628.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. "Founding Cryptography on Oblivious Transfer - Efficiently". In: *CRYPTO*. Vol. 5157. Lecture Notes in Computer Science. Springer, 2008, pp. 572–591.

[Ish+07] Yuval Ishai et al. "Zero-knowledge from secure multiparty computation". In: *STOC*. ACM, 2007, pp. 21–30.

[Jan+20] Shahrooz Janbaz et al. "A fast non-interactive publicly verifiable secret sharing scheme". In: *ISCISC*. IEEE, 2020, pp. 7–13.

[KM15] Vladimir Kolesnikov and Alex J. Malozemoff. "Public Verifiability in the Covert Model (Almost) for Free". In: *ASIACRYPT (2)*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 210–235.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer". In: *CCS*. ACM, 2016, pp. 830–842.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: Making SPDZ Great Again". In: *EUROCRYPT (3)*. Vol. 10822. Lecture Notes in Computer Science. Springer, 2018, pp. 158–189.

[Lin13] Yehuda Lindell. "Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries". In: *CRYPTO (2)*. Vol. 8043. Lecture Notes in Computer Science. Springer, 2013, pp. 1–17.

[Lin17] Yehuda Lindell. "How to Simulate It - A Tutorial on the Simulation Proof Technique". In: *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 277–346.

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. "The IPS Compiler: Optimizations, Variants and Concrete Efficiency". In: *CRYPTO*. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 259–276.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. "Dishonest Majority Multi-Party Computation for Binary Circuits". In: *CRYPTO (2)*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pp. 495–512.

[LP07] Yehuda Lindell and Benny Pinkas. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *EUROCRYPT*. Vol. 4515. Lecture Notes in Computer Science. Springer, 2007, pp. 52–78.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. "Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries". In: *SCN*. Vol. 5229. Lecture Notes in Computer Science. Springer, 2008, pp. 2–20.

[Nie+12] Jesper Buus Nielsen et al. "A New Approach to Practical Active-Secure Two-Party Computation". In: *CRYPTO*. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 681–700.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. "LEGO for Two-Party Secure Computation". In: *TCC*. Vol. 5444. Lecture Notes in Computer Science. Springer, 2009, pp. 368–386.

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT*. Vol. 1592. Lecture Notes in Computer Science. Springer, 1999, pp. 223–238.

[PL15] Martin Pettai and Peeter Laud. "Automatic Proofs of Privacy of Secure Multi-party Computation Protocols against Active Adversaries". In: *CSF*. IEEE Computer Society, 2015, pp. 75–89.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[Sch18] Berry Schoenmakers. *Secure Multiparty Computation in Python*. May 2018. URL: `https://www.win.tue.nl/~berry/mpyc/`.

[Sch99] Berry Schoenmakers. “A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic”. In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 148–164.

[Sha79] Adi Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (1979), pp. 612–613.

[Sma16] Nigel P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016. ISBN: 978-3-319-21935-6. DOI: 10.1007/978-3-319-21936-3. URL: https://doi.org/10.1007/978-3-319-21936-3.

[SSS21] Peter Scholl, Mark Simkin, and Luisa Siniscalchi. “Multiparty Computation with Covert Security and Public Verifiability”. In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 366.

[Sta96] Markus Stadler. “Publicly Verifiable Secret Sharing”. In: *EUROCRYPT*. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996, pp. 190–199.

[SZ13] Thomas Schneider and Michael Zohner. “GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits”. In: *Financial Cryptography*. Vol. 7859. Lecture Notes in Computer Science. Springer, 2013, pp. 275–292.

[Yao82] Andrew Chi-Chih Yao. “Protocols for Secure Computations (Extended Abstract)”. In: *FOCS*. IEEE Computer Society, 1982, pp. 160–164.

[Yao86] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *FOCS*. IEEE Computer Society, 1986, pp. 162–167.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. “More Efficient MPC from Improved Triple Generation and Authenticated Garbling”. In: *CCS*. ACM, 2020, pp. 1627–1646.