Karlsruhe Institute of Technology

KASTEL

FortifiedIPS: Increasing the Security of Multi-Party Computation by Diverse Redundancy

Master Thesis

KIT – Karlsruher Institut für Technologie
ITI – Institut für Theoretische Informatik
Forschungsruppe Kryptographie und Sicherheit

Jonas Vogl

February 22, 2021

Supervisors: Prof. Dr. Jörn Müller-Quade
Markus Raiber
Jeremias Mechler

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 22. Februar 2021

(Jonas Vogl)

Abstract

In this work we propose an approach to increase security of multi-party-computation (MPC) protocols. We consider a setting in which each party consists of several devices with different hardware, operating system and software (diverse redundancy). Under the assumption that the devices cannot all be hacked remotely due to their different setup, we construct a protocol that is secure even if the only remaining honest party is partially corrupted. To capture that assumption, we propose a corruption model that includes both attacks via physical access and more limited remote hacks that rely on exploits that are specific to hard- or software. While there is already work on the use of diverse redundancy for security, as far as we know this has not been done before with a formal security guarantee.

Many kryptographic protocols (implicitly) assume that each party consists of a single device, which is either entirely corrupted or completely honest. So necessarily we divide single parties that take part in a protocol up into several devices that collaboratively fulfill the role of one party in the protocol. To secure single points of failure within parties, we use another, already existing, MPC protocol called SPDZ [Dam+13]. We use this protocol in a very efficient way, that relies on the trust that devices within a party have initially, before any corruptions can take place. This initial trust allows us to skip the most expensive part of SPDZ, the setup phase. This approach incurs linear overhead compared to protocols that do not split up their parties.

The resulting protocol can guarantee that parties that loose up to one fourth of their devices to an adversary can still keep their input and output secret and can continue to participate in the protocol as an honest party.

Zusammenfassung

In dieser Arbeit präsentieren wir einen Ansatz, mit dem die Sicherheit von Protokollen für multi-party-computations (MPC) verbessert werden kann. Dafür gehen wir davon aus, dass Protokollteilnehmer aus mehreren Geräten mit unterschiedlicher Zusammensetzung von Hardware, Software und Betriebssystemen bestehen. Dies wird als diverse Redundanz bezeichnet. Dazu wird die Annahme getroffen, dass redundante Geräte aufgrund ihres unterschiedlichen Aufbaus nicht alle gleichzeitig korrumpiert werden können. Auf dieser Basis konstruieren wir ein MPC Protokoll, das sicher bleibt, selbst wenn die letzte ehrlich Partei teilweise korrumpiert wird.

Um die Annahme formal zu beschreiben, schlagen wir ein Korruptionsmodell vor, das zwei unterschiedliche Typen von Korruptionen vorsieht. Um Angriffe über physikalischen Zugriff auf Geräte zu beschreiben, wird der übliche aktive Angriff benutzt. Angriffe über das Netzwerk werden jedoch eingeschränkt, um zu modellieren, dass solche Angriffe auf vorhandene Sicherheitslücken angewiesen sind. Wenn Systeme in diverser Redundanz vorliegen, ist es unwahrscheinlich, dass sie alle zur selben Zeit Sicherheitslücken aufweisen. Dieser Ansatz wird in der praktischen IT-Sicherheit bereits eingesetzt, wurde, so weit wir wissen, aber noch nicht verwendet, um formale Sicherheitsgarantien zu geben.

Viele kryptographische Protokolle machen (implizit) die Annahme, dass jede Partei aus nur einem physikalischen Gerät besteht. Deshalb wird eine Partei dann entweder vollständig korrumpiert oder bleibt komplett ehrlich. Deshalb ist es für unsere Zwecke notwendig, Parteien in mehrere Geräte aufzuteilen. Diese Geräte führen dann ein Protokoll aus, mit dem eine ganze Partei realisiert wird. Um wichtige Stellen zu schützen, an denen die ganze Partei auf einmal korrumpiert werden könnte, setzen wir das MPC Protokoll SPDZ [Dam+13] ein. Hier nutzen wir aus, dass SPDZ nur innerhalb einer Partei eingesetzt wird. Hier vertrauen sich die Geräte, zumindest zu Beginn, bevor Korruptionen stattfinden können. Dieses initiale Vertrauen erlaubt es, den aufwändigsten Teil von SPDZ, die Vorverarbeitungsphase, zu überspringen. Dieser Ansatz verursacht linearen zusätzlichen Aufwand im Vergleich zu herkömmlichen Protokollen. Dafür wird sichergestellt, dass Parteien, die bis zu einem Viertel ihrer Geräte aufgrund von Korruptionen verlieren, weiter als ehrliche Parteien am Protokoll teilnehmen können. Außerdem bleibt ihre Ein- und Ausgabe geheim.

Contents

1 Introduction

In theoretic cryptography adversarial models are usually stronger than their real world counterparts would be. This does make sense because proven security guarantees only hold as long as their assumptions, for example on an adversary's capabilities, are not broken. However, this also prevents limitations of real world adversaries from being used against them. One such limitation is that remote hacks require an exploit which depends on hardware and software of the target and is not necessarily always available. Adversarial models usually do not account for that and allow them to corrupt any party they want to. While static adversaries can corrupt parties only at the start of the protocol, adaptive adversaries can corrupt any party at any time. Obviously no real adversary has that freedom with corruptions. For a corruption in the real world the adversary requires either physical access to the device or a vulnerability in the device's software. Physical access to devices is of course relatively difficult to achieve. And while vulnerabilities are common, they are not necessarily always available for any device, especially on well maintained and hardened systems.

Our goal is to use the dependency of adversaries on vulnerabilities to increase security of multi-party-computation (MPC) protocols. For that purpose we propose a modified corruption model that captures both physical and remote attacks. We then use diverse redundancy as an assumption to protect from remote attacks.

This results in a generic protocol, based on the IPS compiler [IPS10], where parties are distributed among several devices that have a different hardware and software setup (diverse redundancy). We show that with this protocol a party that looses a limited amount of devices to online corruptions is still secure and can continue to participate in the protocol as an honest party.

While this seems like a restriction for the adversary, the corruption model is defined so that overall the notion of security is stronger than the common universally-composable-security (UC). Should the assumption that online corruptions are limited be false, standard UC-security remains as a fallback.

1.1 Related Work

In this section we go over some other works with comparable ideas. The MPC in the head paradigm is a good starting point for what we want to achieve. Both fortified UC and robust combiners are comparable to what we want to achieve. The idea of fortified UC has very similar goals as we do, but uses a different approach to get there. Robust combiners also rely on the base idea of diverse redundancy, but have different goals.

1.1.1 MPC in the Head

The MPC in the head paradigm was first introduced in [Ish+07] and also inspired the IPS compiler. Protocols that follow this paradigm usually have each party imagine several servers and introduce some kind of control mechanism to ensure that an honest majority assumption on the imagined servers holds. This seems to be a good start to add diverse redundancy, instead of imagined servers we intend to place real server in hardware. The control mechanisms can also be useful because the goal is to be able to withstand a limited amount of corrupted servers. So MPC in the head protocols are natural candidates to use for our goals. Thus we choose the IPS compiler as the core of our protocol.

1.1.2 Fortified UC

Broadnax et.al. [Bro+18] show how to increase security of MPC protocols with a different approach. Their goal is to protect the secret input of parties that get corrupted during the protocol. This is achieved by using simple unhackable hardware modules. Examples for those hardware modules are data diodes, which are unidirectional channels and secure hardware modules for simple functions like a specific encryption scheme.

This is different to our approach, because we do not assume that a single piece of hardware is secure. Instead we assume that hardware is redundant and diverse so that not all of it can be hacked at the same time. Any individual server can still be hacked though. Despite the difference in approach, we share the same goal of protecting the secret input of corrupted parties.

1.1.3 Practical Security

The idea to use diverse redundancy is not new in the practical side of IT-security. Littlewood et.al. [LS04] argue that it is not only usable against random error but can also be employed against intelligent adversaries. To the best of our knowledge this approach has not yet been

used in theoretic IT-security. Previous work on diverse redundancy seems to focus on how to set up redundant systems so that they are not vulnerable to the same attacks.

Instead we use diverse redundancy as an assumption. Our goal is to provide a security guarantee under the assumption that of several redundant servers only a limited amount can be hacked. We present a proof of concept that uses diverse redundancy as an assumption to achieve a stronger notion of security.

1.1.4 Robust Combiners

As Herzberg [Hero7] describes, the concept of robust combiners also uses different components in the hope that not all of them are compromised. A simple example for a robust combiner would be a cascade of encryption schemes. Here the plaintext is encrypted by one encryption scheme, and the resulting ciphertext is then encrypted again with another encryption scheme. Herzberg shows that the resulting cascade is IND-CPA secure if just one of the used encryption schemes is IND-CPA secure.

While the base assumption of this approach is very similar to ours, we achieve a different goal. Robust combiners are used as a fallback. Should one scheme fail, there is still the other as a failsafe. In contrast we try to use a similar base assumption to protect from corruptions that would break a single system.

Another difference is that robust combiners use different protocols or algorithms and combine them. In our case all redundant devices use the same algorithms, but they run on different hardware and have different implementations. So while robust combiners can withstand a faulty protocol, our goal is to protect from faulty implementations.

1.2 Our Contributions

Here we quickly summarize our three main contributions. We present a new corruption model and both a generic and a more specific protocol that is secure against the new corruption model.

New Corruption Model We propose a new corruption model that captures the adversary's dependency on vulnerabilities to remotely corrupt devices. In this model the adversary is still able to corrupt an entire party that consists of several devices, which models adversaries that use physical access to corrupt devices. This can result in an overall stronger security guarantee that still falls back to standard UC security should new assumptions be broken.

Generic Protocol Based on the IPS compiler, we propose a generic protocol that can be used for secure MPC and is secure against adversaries that use the new corruption model. The result is an MPC protocol that remains not only secure if all but one party is completely corrupted. With the new corruption model we can also model a partial corruption of remaining honest parties. Our combined protocol remains secure even if up to $1/4th$ of the hardware-servers (HWS) of honest parties is corrupted.

Specific Protocol We propose already existing candidate protocols that can be used to implement the generic protocol and prove the security of the resulting combine protocol. Our analysis of efficiency shows linear overhead compared to the standard IPS compiler. We also analyze how parameters are best chosen based on the number of HWS used and the width of the circuit that is computed.

Overall, our work can be considered a proof of concept for the idea to use diverse redundancy to strengthen MPC protocols.

1.3 Overview

First, we provide some basic knowledge on the topic in Chapter 2. This is followed by a description of the IPS compiler in Chapter 3. The IPS compiler is a generic compiler that combines two protocols of weaker security properties into one actively secure protocol. Since parties of the IPS compiler logically already consist of several servers, it is easy to divide parties into different devices. Thus, the IPS compiler will be the core of our work. Then we will define our adversarial model in Chapter 4. This models attacks via physical access and remote hacks as two different types of attacks. In the next chapters we describe our main contributions. In Chapter 5 we modify the IPS compiler so that its parties can be divided into different hardware-server. In that chapter we still assume that a single point of failure within each party is implemented by an ideal functionality $\mathcal{F}_{Client}$. In Chapter 6 we propose an implementation for $\mathcal{F}_{Client}$ using the online phase of the SPDZ protocol. There we also analyze the overhead our approach causes compared to the plain IPS compiler.

2 Preliminaries

In this chapter we give a short overview over the basic concepts and ideas that are relevant to this work. First, we recap on the idea of MPC and related concepts such as the Real/Ideal paradigm and universal composability (UC). Then oblivious transfer and secret sharing will be introduced, two basic cryptographic buiding blocks that are used in this work. Since the IPS compiler is essential to our work, we will discuss it in more detail in Chapter 3.

2.1 MPC for any Function

As Lindell [Lin21] describe, the general goal in the field of secure multi-party-computation (MPC) is to construct protocols that allow several parties to securely compute any function together. Input can be provided by several (or all) parties. Distributed computing is often used to prevent random, natural errors, in MPC however, the goal is to protect from a malicious intelligent adversary that also may have the role of a protocol participant. MPC protocols should guarantee privacy and correctness. Privacy ensures that other protocol participants as well as any third party can not learn a party's private information including its input to the function that is computed (with the exception of what can be derived from the output). Correctness assures that other parties can not manipulate the result of computations except through their input.

2.2 Real/Ideal Paradigm

Proofs of security in the context of MPC often use the Real/Ideal paradigm. An introduction to this paradigm also referred to as simulation based proofs, was written by Lindell [Lin17]. The basic idea of proofs that follow this paradigm is to compare real protocols to ideal functionalities which are secure by definition. In the ideal world parties use ideal functionalities. Parties just provide input to them, the ideal functionality then performs the computation and sends the output back to the parties. The ideal functionality is trustworthy, so the output is correct, and it can not be hacked, so the input of each party remains private. To prove that a protocol is secure, it is then sufficient to construct a simulator that can hide the differences between real

and ideal world. The argument then is that any differences of the real protocol to the ideal functionality can be made up by the adversary without access to a real protocol. Thus, the adversary can not learn anything from the real protocol.

2.3 Secure Channels

Throughout this work we implicitly assume that any communication between two devices is done via secure channels. A secure channel guarantees both authentication and privacy of the message. However, an adversary can prevent messages from arriving.

Intuitively this means that an adversary can not read or manipulate messages, unless sender or receiver are corrupted. The adversary also can not impersonate devices unless the device is corrupted first. But the adversary is able to "cut the wire" between two honest parties and suppress their communication.

2.4 UC-Framework

Protocols usually do not run alone in an isolated environment. There are often many instances of the same protocol as well as other protocols running simultaneously on the same machines and in the same network. However, a protocol that is proven to be secure on its own is not necessarily also secure when there are other instances or protocols running as well [Lin21]. To account for the unpredictable environment in which protocols will have to run, Canetti [Can01] proposed the Universal-Composability (UC) Framework. Canetti's universal composition theorem states that protocols which are proven to be secure in the UC-Framework remain secure in an arbitrary environment, which represents other protocols and instances running at the same time. The environment is modeled as one machine that represents everything that is not part of the protocol execution. The UC-Framework requires that no environment can distinguish a real protocol execution from the ideal functionality it implements.

Parties in UC In the UC-Framework Parties are modeled as interactive Turing machines (ITM). These are Turing machines with several special tapes and instructions that are used to model network activity. To model messages sent between computers, ITMs have an outgoing message tape and an external-write instruction that writes the content of the outgoing message tape to a tape of another ITM. ITMs also have an identity tape. This is a read only tape that contains the unique ITM's identity (pid). ITMs can invoke other ITMs as subroutines. Usually protocols in the UC-Framework assume a pid-wise corruption. This means that all ITMs that are part of one party (subroutines of one main ITM) have the same id. The adversary has to

corrupt by id, so a party can only be corrupted all at once. Corruptions are represented by a backdoor tape. The adversary can write on an ITM's backdoor tape to influence (corrupt) it. For a complete overview over all special tapes and instructions please refer to Canetti's UC Paper [Can01].

Note that we describe this mainly in order to describe our modified corruption model in Chapter 4 in terms of the UC Framework. Later on we will handle parties and devices more abstract.

Environment The environment's task is to distinguish the real protocol from the ideal functionality. For that purpose, it can interact with the adversary during the protocol execution. The environment also chooses inputs of honest parties arbitrarily before the protocol and receives their output. However, it can not interact with honest parties during the protocol execution. To interact with parties during the protocol execution, the environment has to rely on the adversary.

Dummy Adversary Canetti [Can01] also shows that the adversary that is the hardest to simulate is a dummy adversary. This adversary has no adversarial strategy of its own, it simply sends all the messages it receives to the environment and forwards messages from the environment to other parties. This means, all adversarial logic is moved into the environment, the adversary only relays messages between the parties and the environment.

Since the dummy adversary is the hardest to simulate, it is sufficient to demand that no environment that can interact with a dummy adversary can distinguish the protocol instance from the ideal functionality. It is not necessary to also quantify over all adversaries because the dummy adversary is complete.

2.5 Adversaries in MPC

In the context of MPC adversaries with different capabilities are considered. The adversaries that are relevant to this work are described here, following Lindell [Lin21]. We define the adversaries capabilities in two dimensions, which are described below.

2.5.1 Adversarial Behavior

The adversarial behavior models what the adversary is able to do in order to attack a protocol. The two most common models for that are passive adversaries and the stronger active adversaries.

Passive Adversaries A passive or semi-honest adversary is relatively weak. When it corrupts a party, it receives that party's internal state including previously received messages. However, the adversary and corrupted parties continue to follow the protocol specification.

Active Adversaries In this much stronger adversarial model the adversary can essentially do whatever it wants. It does not have to follow protocol specification and can also cause corrupted parties to deviate arbitrarily from the protocol

2.5.2 Corruption Strategy

The corruption strategy defines when the adversary can corrupt honest parties. The most common versions for this are the weaker static and the stronger adaptive adversaries.

Static Adversaries A static adversary must decide before the protocol starts which parties are corrupted. As soon as the protocol begins, the adversary can not corrupt additional parties. So the adversary can not adjust its strategy based on the protocol execution.

Adaptive Adversaries An adaptive adversary can corrupt parties throughout the entire protocol execution and thus react to messages sent by honest parties. This results in a generally stronger adversarial model, compared to static adversaries.

2.6 Oblivious Transfer

A core component of the IPS compiler is oblivious transfer (OT), a cryptographic primitive for two parties. OT exists in several different flavors. One-out-of-two OT works as follows, as

described by Kurosawa et.al. [KK07]. One party, the sender, has two messages as input m_1 and m_2, while the other party, the receiver, has a choice bit b as input. The OT ensures that the receiver learns message m_b, but does not get any information about the other message, this property is called sender privacy. A second security property, the receiver privacy, ensures that the sender does not learn the choice bit b, so the sender can not learn which message arrived. A message can be a single bit or longer, depending on the OT protocol that is used.

The IPS compiler uses a different kind of OT, the Rabin-String-OT [IPS10]. In this variant, the receiver does not provide input. Only the sender inputs a bit-string, which is then either destroyed or sent to the receiver. The receiver privacy ensures in this case that the sender does not learn whether or not the message got through. Whereas the sender privacy ensures that if the message is destroyed, the receiver does not learn anything about it. Intuitively this variant of OT can be seen as a faulty communications channel, that has a fixed chance of destroying messages sent over it completely (called an erasure channel).

The IPS compiler uses OT as a black box. So for the most part of this work we assume the existence of an OT ideal functionality. We do not consider how this functionality is implemented.

2.7 Secret Sharing

Secret sharing schemes like Shamir's [Sha79] secret sharing are an important component for some MPC protocols, including the IPS protocols used in Chapter 6. They are protocols for two or more parties of which one is the dealer. The dealer holds a secret which he splits up into several shares. Each party then receives a share from which no information about the secret can be derived, unless at least t shares are combined. Secret sharing schemes such as Shamir's still allow addition and multiplication of secrets, even in their shared form. An adversary has to gain knowledge of more than t shares to be able to learn the secret.

Generally, this is done by using polynomials over a finite field, as described in [KL15]. To generate shares of a secret c, the dealer chooses a random polynomial that evaluates to c for $x = 0$. The dealer can now give out other evaluation points as shares. If the polynomial has a degree d, any $d + 1$ shares can be used to reconstruct the secret by interpolation, while a party that knows only d or less shares can not learn anything about the secret. This approach gives the dealer a lot of freedom in the choice of both the number of shares and the threshold that is required to reconstruct the secret. Note that what we described here is just the core idea. Usually some variation of this core idea is used.

For example to prevent a malicious dealer from distributing incoherent shares or a malicious party from manipulating its share a verifiable secret sharing (VSS) can be used. An example

for a VSS scheme can be found in Chapter 6.

2.8 Arithmetic Circuits

One way to model computations that is used in the context of MPC, is the concept of arithmetic circuits. Similar to normal binary circuits, arithmetic circuits also consist of wires and gates. However, the values of wires in arithmetic circuits are not just bits, but elements of a finite field. So instead of gates for binary operations, arithmetic circuits have gates that add or multiply field elements.

For example, the computations that can be performed by the SPDZ protocol are modeled as arithmetic circuits. For more details see Chapter 6 where we use SPDZ.

2.9 Preprocessing Assumption

Some MPC protocols such as SPDZ [Dam+13] require a preprocessing assumption. In a preprocessing model it is assumed that every party has access to an ideal functionality $\mathcal{F}_{PREP}$ that generates setup information for the protocol. In our case, we have a trusted server that can compute simply $\mathcal{F}_{PREP}$. If that is not the case, it can also be realized with an MPC protocol such as the SPDZ offline phase.

3 IPS Compiler

A large part of this work is essentially a modification of the IPS compiler that increases its security. So in this chapter we describe the IPS compiler, as it was introduced by Ishai et.al. [IPS10].

The IPS compiler takes two protocols. First, a multi-party protocol Π that is secure against active adversaries under an honest majority assumption. This means it is secure only as long as no more than a fixed fraction of parties is corrupted. the second one is a multi-party protocol ρ that is secure against passive adversaries. Those two are combined into one multi-party protocol that is secure against active adversaries, but no longer requires an honest majority. However, a secure oblivious transfer (OT) is required for this, so the compiled protocol is secure in the OT-hybrid model where all parties have access to an ideal functionality $\mathcal{F}_{OT}$.

3.1 Outer Protocol

The outer protocol Π realizes a functionality $\mathcal{F}$ which will also be the functionality of the compiled protocol. It consists of two different types of participants: Clients and Servers. Clients each represent a party that takes part in the protocol, they provide input and can receive output. All but one of the m Clients may be corrupted, there is no honest majority assumption regarding them. Servers are neutral devices that support the protocol's execution, but they do neither provide input nor do they receive output. The honest majority assumption only takes servers into account, it is assumed that no more than t of the n servers are corrupted (with $t = \Omega(n)$). Corruptions of both clients and servers are active. But client corruptions are static whereas servers are corrupted adaptively. Overall, this results in security against active static adversaries in the compiled protocol.

3.2 Inner Protocol

The inner protocol ρ realizes a functionality $\mathcal{G}$ securely against passive adversaries. This functionality is the functionality of a server of the outer protocol. Think of $\mathcal{G}$ as the server's next message function that takes a message sent to the server as input and outputs its response.

If the inner protocol is only statically secure it must have an additional property, called the security against two-step passive corruption. This property is necessary for the proof of security of the IPS compiler, and will also be needed in the proof for our modified variant of the IPS compiler. More information on two-step passive corruptions can be found in the proof for our modification in Chapter 5.

If a server of the outer protocol is simulated by the inner protocol, we call it a *SimS*, short for simulated server.

3.3 Combining both Protocols

The idea for combining both protocols is that the outer protocol's servers are replaced by instances of the inner protocol that are also executed by the clients. But this means that the inner protocol which is only passively secure is now exposed to the active adversary of the outer protocol. Since the outer protocol can withstand up to t server corruptions, it is ok if a few inner protocol instances are corrupted by active attacks. Still an additional security measure is necessary to ensure that the number of corrupted *SimS* remains below the threshold.

For this purpose watchlists are introduced, a mechanism that allows clients to supervise the inner protocol instances so that an adversary has only a negligible chance of cheating in enough *SimS* to break the outer protocol's threshold.

3.4 Watchlists

In order to prevent malicious clients from actively manipulating inner protocol instances, each client gets access to watchlists for some *SimS*. Each client sends all messages it receives and sends as part of the inner protocol instance that implements $SimS_j$ on the corresponding watchlist broadcast channel W_j.

Thus, any other client that can read W_j can check if $SimS_j$ is being simulated faithfully or if there are inconsistencies in its simulation.

Unfortunately, this also means that a malicious client that can read W_j can learn almost all internal state of $SimS_j$. So all *SimS* of which the adversary can read the watchlist have to be considered corrupted. This leads to an interesting tradeoff when choosing parameters for the compiled protocol. Larger watchlists make cheating in inner protocol instances more difficult because the risk of being detected by an honest party is higher. On the other hand, it provides the adversary with more free corrupted *SimS*. For more details on this tradeoff see [LOP11] or Section 6.6.

Watchlists are set up in two steps, first for each client-server pair a watchlist W_{ij} is set up. Then, from those watchlists the watchlist broadcast channels are constructed.

Watchlist Channels First, each pair of clients has to exchange n sufficiently long one-time-pads (OTP) with which the watchlist channels will be encrypted, one for each *SimS*. The OTPs for each watchlist channel are transmitted using a Rabin-String-OT. Thus, each client receives only a portion of the OTPs, so it can read only a fraction of the Watchlist channels. Every client then answers with a random permutation that assigns *SimS* to OT instances. This ensures that every client can choose which OTP is used for which of the watchlists it receives. So each client can choose which *SimS* it can watch. These permuations are necessary to make sure that a client that wants to watch $SimS_j$ can read the all the watchlist channels belonging to $SimS_j$. By the receiver privacy of the OT, the clients (and the adversary) remain oblivious of which watchlists can be read by the other clients.

Watchlist Broadcast Channels The watchlist channels can be used to construct watchlist broadcast channels. This is necessary only if there are more than 2 clients, to make sure that a malicious client can not report different internal server state to different clients.

The broadcast channels are constructed from the normal watchlist channels as follows. When a client receives a message on a watchlist channel, it also sends that message via watchlist channels to all other clients. If it can not read that channel, it still sends a message that indicates so. A client accepts a message on the broadcast channel after it received the same message from all clients via their respective watchlist channels (or the special message if other clients can not read that server's watchlist channels).

If a client receives inconsistent messages, they are discarded.

4 Color Corruption Model

In order to corrupt devices, in reality adversaries require a vulnerability to exploit. Vulnerabilities are usually found for specific hardware, software, or operating systems. This can be used to increase security by introducing redundant systems that are functionally equivalent, but use different hardware and software. Since it is unlikely that all the different systems have known vulnerabilities at the same time, the adversary has a lower success chance if the redundant systems are combined in a way that requires the adversary to corrupt all (or many) of the systems at the same time. This approach was originally intended to provide fault tolerance, but in principle it can also be used to protect against malicious adversaries [LS04]. While this approach is well known in IT-security, to the best of our knowledge, it has not yet been used in theoretical cryptography. For example, the UC-Framework does not model an adversary's dependency on exploits.

To alleviate this, we propose a corruption model that is intended to capture the necessity of exploits for remote corruptions. Overall, in this model the adversary is stronger than an active adversary, and thus we can achieve a stronger notion of security with it. Here the adversary can use a second type of corruption to partially corrupt any remaining parties that are not already corrupted as usual. If the new type of attack is not used, everything is just as in the standard UC-Framework, so UC-Security remains as a fallback.

Our intention is to propose a generic protocol that secures already existing protocols against this new corruption model. This generic protocol will use several devices to implement a single party in the original UC sense. One party in our protocol consists of several devices, but will still take the role of a single party that consists of just one device in a protocol that was designed for the plain UC-Framework.

4.1 The Setting

First, we intuitively describe the setting which we intend to model. Consider several different parties that want to execute an MPC protocol together. However, not all of them have good intentions, some might try to learn more than they should or manipulate the results. All of those parties (or at least the honest ones) have a local network with several devices used in the

protocol. Each device has a unique combination of hardware, software and operating system, so that an exploit that can corrupt one device will most likely fail on the others. This will be modeled by assigning colors to devices and limiting the adversary in the number of colors that can be corrupted. While honest parties will usually remain honest, their devices may be hacked remotely. Honest parties are set up in a way that limits the effect of remote hacks.

Our goal in the following chapters is to propose a protocol that uses this setting so that an honest party that looses some devices to hacks can continue the protocol as an honest party and keeps its inputs and outputs secret.

4.2 Modeling Diverse Redundancy

For this framework to make sense, we need at least one party to consist of several devices (modeling redundancy without redundant devices is quite pointless). Each device is assigned a color. This model is then based on the assumption that an adversary can only corrupt devices of a fixed number of colors. The adversary has no influence on how many colors it may choose. But before choosing colors, the adversary can find out which device has what color. We model each device as an ITM just as in the plain UC-Framework. A device's color is fixed in its identity tape.

4.3 Honest Parties

In the UC-Framework every party is modeled as just one device in form of an ITM. Those ITMs can invoke additional ITMs as subroutines.

Every device is an ITM just as in UC. Devices are grouped together into parties. There is some trust between devices within a party that at least initially all devices within that party have the same goal. However, devices of honest parties may be corrupted.

4.4 Adversarial Model

The adversary now has access to two different kinds of corruptions.

First, we have the corruption of entire parties. Although this now corrupts several devices at once, this is the equivalent of a corruption in the plain UC-Framework. The adversary can use this type of corruption as much as it likes. All parties may be corrupted this way, however, most protocols will not be able to give any security guarantee in this case. This kind of corruption can be either static or adaptive.

The adversary can then use the second type of corruption to partially corrupt any remaining honest parties. Instead of a pid-wise corruption this is a color-wise corruption. For this the adversary has to choose a color (or several) and then can corrupt all devices of that color. After choosing the color the adversary first learns the id of all devices of that color. It then can corrupt them at will.

Devices of any different color remain honest unless their party is corrupted by the first type of corruption. This type of corruption is always adaptive, so devices of the chosen color can still be corrupted later if they were not corrupted initially.

If a protocol is secure under colored corruptions, it can give security guarantees even if some devices of limited colors of the last remaining honest party are corrupted.

4.5 Additional Considerations

Finally, some considerations on the use of the color model. To achieve our goal parties have to be set up correctly, with different devices of different colors. Otherwise, for example if all devices have the same color this model defaults back to the normal UC-Framework with active, adaptive adversaries.

4.5.1 Internal Layout of Parties

For simplicity throughout this work, it is assumed that the devices of each party are equally important and that each color is present in only one device per party. It is not important that each color is actually present just once, as long it is hidden from other parties. A party can internally divide its devices of one color into several smaller devices as long as they present a single interface to other parties (and other devices of the same party). All the devices of a party that have the same color can be corrupted together. This is not related to security, but simply for practicality and flexibility so that each party can have a different internal setup with larger and smaller servers.

For the internal setup of each party it is very important though that the devices of each color are equally important, so that security does not depend on the adversary's pick of color. Any single points of failure must be either uncorruptable by assumption or otherwise distributed equally to the other devices, possibly by MPC.

4.5.2 Realizing Colored Devices

We also want to provide some thoughts on how to realize colored devices.

When setting up devices of different colors, it is important to make sure that all their components are completely different. If they share just one component, and that component is vulnerable to an exploit, the underlying assumptions are broken and all additional benefit of this model is lost. Standard UC-security still holds in this case, but any party that looses some of its devices then must be considered corrupted entirely.

In the real world there is only a limited number of hardware manufacturers. The number of different kernels is also very low, and development of several different implementations of the same protocol is expensive. Thus, the number of different colors used should be chosen very conservative. We would consider a number of 3 to 5 different colors fairly reasonable. Any higher number of colors would be very difficult to justify though.

4.5.3 More than one Corrupted Color

In theory it is possible to allow the adversary to corrupt devices of more than one different color as long as enough devices remain uncorrupted. However, the protocols that are considered in this work would not function properly with more than one corrupted color because of the limited amount of different colors available. Our protocol can not tolerate more than $1/4th$ of its devices to be corrupted. To be able to tolerate two corrupted colors, they need at least 8 devices of different colors. At this point the assumption that all those devices consist of entirely different components is no longer reasonable. So throughout this work we assume that just one color can be corrupted. But note that in theory our results are more generic. The protocol in Chapter 6 can withstand the corruption of $1/4th$ of its devices, also if it has more than 4 devices. The exact fraction of how many devices can be corrupted is determined by the threshold of the outer protocol that is used in our variation of the IPS compiler.

4.5.4 Composition with other Protocols

When composing different protocols in the color model it is important to consider how choices of different colors in different protocol instances interact. If for every instance a new set of colors is introduced, the global amount of colors becomes bigger and bigger, which is very unreasonable, as explained above. That is why throughout this work we assume a global fixed set of colors that all protocols or parties have to use.

5 IPS Compiler with Diverse Redundancy

Here we propose a modification of the IPS compiler where parties consist of several diverse hardware-servers (HWS) which we use to harden the IPS compiler. We initially assume every party has a secure client which the adversary can not corrupt. The client is modeled as an ideal functionality and fulfills the same role as in standard IPS, except for the simulation of the servers of the inner protocol and consistency checks.

For the HWS we use the color model of the previous chapter. In each party, every HWS has a different color assigned to it and the adversary can corrupt all HWS of one color. However, each party uses the same set of colors for its HWS. A secure instantiation of the protocol is only possible if less than t inner protocol instances run on HWS of the same color in one party. Each HWS is responsible for the simulation of an equal amount of simulated-servers (*SimS*). In this chapter we prove security under the assumption of a secure client. How to securely implement the client will be covered in a later chapter. For simplicity, we assume each party has five HWS with each a different color. Any other distribution that satisfies the above constraint will work as well, so we do not lose generality.

5.1 Initial Thoughts

First, we describe some initial thoughts on how to best achieve our goals. This is supposed to motivate and justify some important design decisions.

5.1.1 Splitting up Parties

Instead of splitting up parties, it is possible to just use an already existing MPC protocol and instanciate it with more parties. Every real party then would consist of several servers that each take the role of a party in the protocol. While this idea is very simple and obviously secure, it has some disadvantages. First, the input and output of parties would leak partially if some of its servers are corrupted. Also MPC protocols have the tendency to scale very poorly with the number of parties. For example, the IPS compiler's watchlist setup requires communication that is quadratic in the number of parties.

Thus, we chose another approach where we split up parties instead of introducing more of them. Sensitive data and single points of failure are protected with an MPC protocol. The IPS compiler is a natural choice for the core protocol since it logically already consists of different servers. So splitting up IPS parties is very straightforward.

5.1.2 Splitting Parties of the IPS Compiler

When splitting up parties of the IPS compiler, two main problems have to be solved. First, the watchlists and their consistency checks have to be distributed. Second, the client part of the outer protocol has to be secured. We discuss different approaches for both problems.

Watchlist Server The consistency checks can be done by additional HWSs that can receive data, but send only very limited output as shown in Figure 5.1. The watchlist servers receive the watchlist channels via a data diode (a unidirectional channel), so they can only receive but not send data. This ensures that while the watchlist servers can still be corrupted, they can no longer send information back to the adversary, so the watchlists remain secret. The only output the watchlist servers can provide is a shutdown signal that terminates the entire protocol. While this ensures that the party's watchlist is not leaked to the adversary, it can still suppress consistency checks by corrupting watchlist servers and turning them off. Since we want to allow only a fraction of the HWSs to be corrupted, we simply place enough watchlist servers so that the adversary can not corrupt all of them. The exact number of servers required depends on the number of corruptions we want to allow within the party.

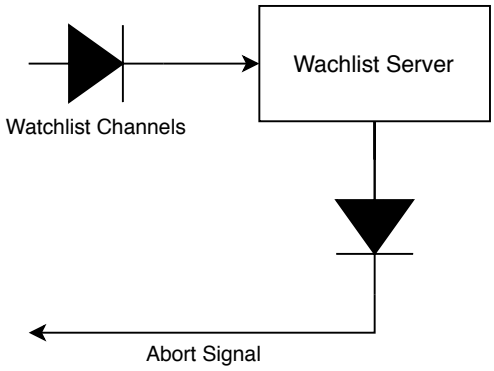

Figure 5.1: Watchlist server

It is easy to see that this approach is secure with the correct parameters. This requires a lot of additional hardware though, so a more elegant approach would be preferable.

It is possible to distribute consistency checks so that every HWS performs a part of it. We

set each party up so that each HWS of a party is responsible for the simulation of an equal amount of *SimS*. Every HWS is also responsible for an equal amount of watchlist consistency checks. So every HWS receives enough OTs via which the OTPs for watchlist channels are transmitted.

It is now important to make sure that *SimS* are checked only by the same HWS that also simulates them. We do this to reduce the number of *SimS* that are affected by the corruption of one HWS. If watchlists were assigned randomly, a corrupted HWS would not only corrupt all *SimS* that are simulated by it, the adversary also learns which OTs were successfully received by this HWS. Thus, the adversary would learn which *SimS* are watched by the corrupted HWS. The adversary could now corrupt all those additional *SimS* by cheating in the inner protocol without getting caught. So letting each HWS check on its own *SimS* makes sure that both ways of corrupting *SimS* affects the same set of *SimS*. So overall, this reduces the number of corrupted *SimS* caused by one HWS corruption.

Trust within Parties To secure the functionality of a client of the outer protocol, we use an MPC protocol. Since this MPC protocol is run only by the HWS of one party, it is reasonable to assume some trust among the participants of the MPC protocol. We leverage this trust by using SPDZ, a protocol that relies on a preprocessing assumption. Most complexity of SPDZ comes from a protocol that realizes that preprocessing assumption without trust. In our case we can use the trust to implement the preprocessing and skip a large part of SPDZ. This leaves us with a very efficient MPC protocol which we use to implement the client. We mention this here only to motivate the design of our protocol. More details on SPDZ and how we use it to implement the client can be found in Chapter 6.

5.2 The Protocol

Here we describe changes we made to the original IPS Protocol. Most noteworthy is that one party's functionality is now distributed to several devices. This has the benefit that, as long as a party does not loose too many devices, it remains honest.

5.2.1 Party Layout

Each party of the original IPS compiler is now a system that consists of several hardware components. An overview of a single party can be found in Figure 5.2. Each party has one interface client which is responsible for in and output. It also has h HWS that do most of the work, including the simulation of *SimS* and checking watchlists. Finally, each party has an

ideal functionality $\mathcal{F}_{Client}$that depends on the outer protocol that is used.

When a new protocol instance is started, its input is sent to the interface client. The input server then shares the input among the h HWS (dashed lines). We assume that the adversary can not corrupt the interface client and learn that party's input via remote corruptions. If the adversary corrupts the entire party, it can still learn its input though.

The HWS can now start the actual protocol by sending their shares of the input to the client functionality $\mathcal{F}_{Client}$. Upon receiving input $\mathcal{F}_{Client}$ answers with the first protocol messages. In terms of the IPS compiler $\mathcal{F}_{Client}$ fulfills the role of a client in the outer protocol while the HWS each simulate an equal fraction of servers and handle the corresponding watchlists.

The HWS have different colors so that partial corruptions can not be used to corrupt too many of them.

When the protocol is finished, $\mathcal{F}_{Client}$ sends shares of the output to the HWS. More details on $\mathcal{F}_{Client}$ can be found below.

The HWS then send their shares of the output to the interface client which reconstructs and outputs the result.

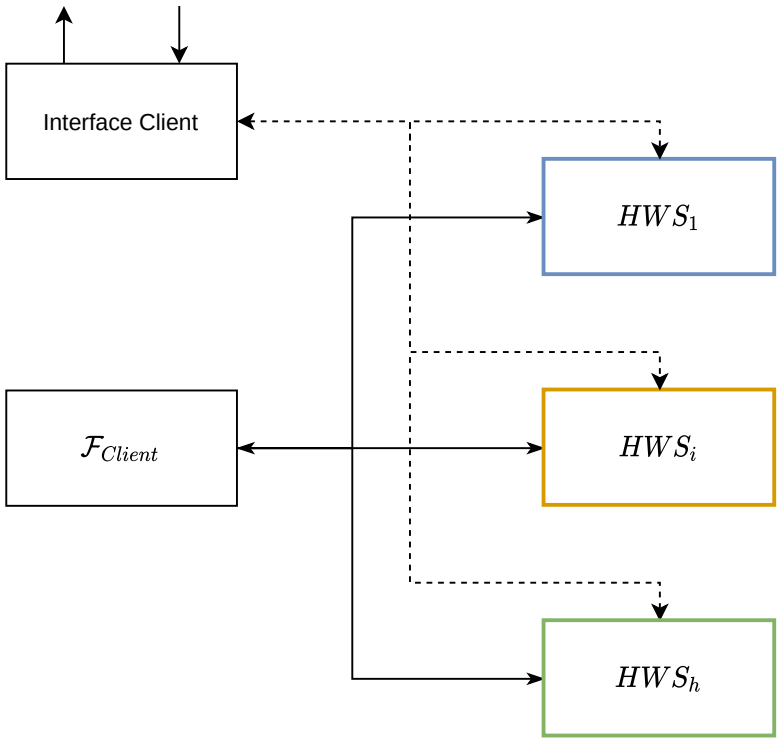

Figure 5.2: The layout of a single party in the combined protocol with an interface client for input and output, several HWS in different colors and the ideal client functionality.

5.2.2 Ideal Client Functionality

Here we shortly describe the ideal functionality $\mathcal{F}_{Client}$. The goal behind its definition is to make it as easy as possible to replace it with an MPC protocol. Input and output is relayed via the HWS so that they can use a MPC protocol to implement $\mathcal{F}_{Client}$.

$\mathcal{F}_{Client}$ is mainly defined by the outer protocol that is used. It answers exactly as a client would with just a few exceptions. Since the client functionality depends on the outer protocol used and we want to achieve a generic result, we can define $\mathcal{F}_{Client}$ only very broadly. For an example see Figure 6.4 in the next Chapter, where we use a specific outer protocol. As mentioned before, input and output are shared among the HWS. Client to client messages, both sent and received, are also shared among the HWS who relay them to other clients.

When a client wants to send a message to another client, it first splits up the message into shares. The number of shares is the same as the number h of HWS. The choice of parameters must be so that the shares from uncorrupted HWS are sufficient to reconstruct the secret. This means, there are more shares than required for reconstruction. The additional shares are used for error detection. For all additional shares, it is checked if they lie on the same polynomial as all the other shares. If not, the message was manipulated and is discarded. Alternatively, the protocol can be aborted at this moment.

Messages to and from servers of the outer protocol are sent only to the one HWS that simulates that server. $\mathcal{F}_{Client}$ is not involved in watchlists or the simulation of servers.

5.2.3 Hardware-Server

Figure 5.3 shows how two parties interact in the combined protocol. Note that HWS are displayed in their color to show their unique hardware setup. Components that work together to simulate servers of the inner protocol are outlined in red, they do not reflect hardware. Interaction between parties is all done by HWS. They are responsible for implementing the n *SimS* together with HWS of other parties using the inner protocol, outlined in red. Each HWS runs an equal fraction n/h of all the clients of the inner protocol (the a_i and b_i in the figure). Every HWS only communicates with one other HWS of each other party. Besides the communication of the inner protocol instances, HWS send messages to set up the n watchlist broadcast channels $W_1, \ldots, W_n$, send messages on the watchlists, and forward client-to-client messages.

The number of HWS should be as small as possible because the color model does become less and less reasonable the more different colors are used. Since there is a significantly larger number of *SimS* (at least a few hundred, see Section 6.6), each HWS has to implement several *SimS*. Each HWS is responsible for an equal fraction of *SimS*.

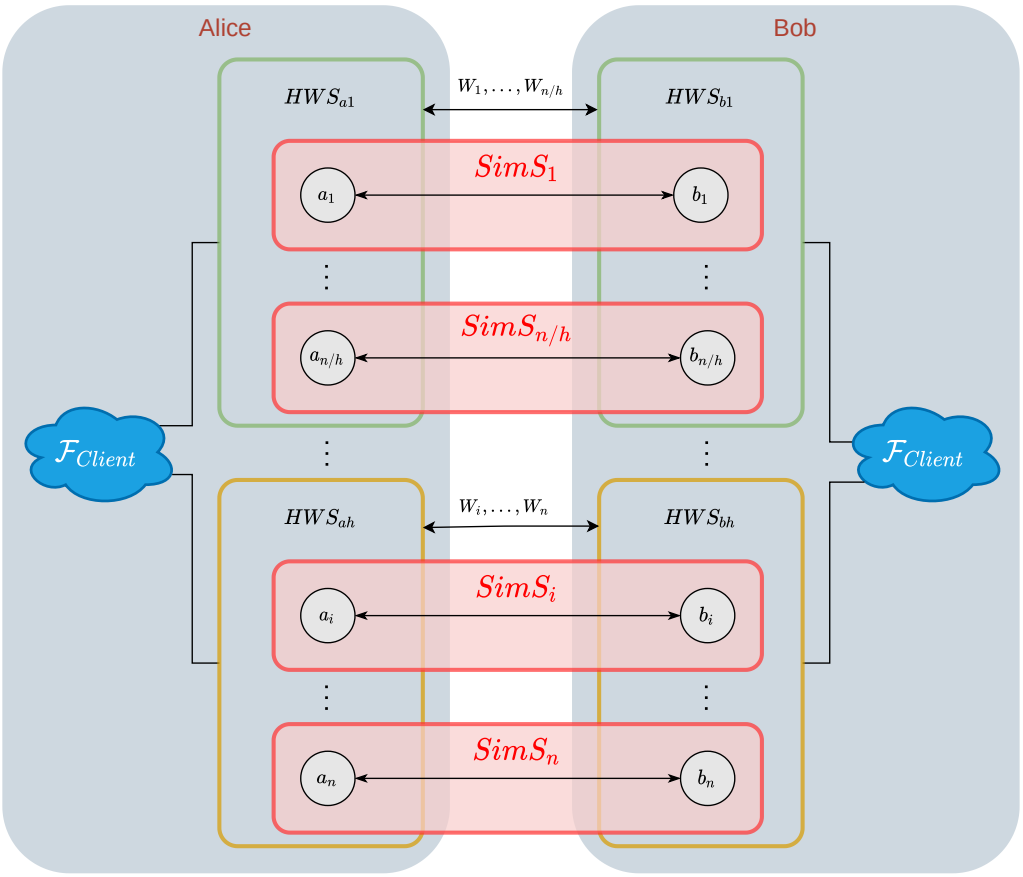

Figure 5.3: Overview of the compiled protocol, each party consists of several HWS of different colors. HWS run the inner protocol clients (the as and bs) that simulate *SimS* together. HWS of different parties that simulate *SimS* together and check each others watchlists W_i have the same color. Note that for readability the interface client, from which the HWS receive input and output is not shown here.

In the original IPS compiler the client is responsible for monitoring the watchlists which presents a few problems in a distributed setting. To check the watchlists, knowledge of the full internal state of the servers is required. This means, the HWS would have to send their full internal state to the client. This is unwieldy because of the communication overhead and might not be very useful because a corrupted HWS might not report faithfully in order to hide cheating in a *SimS*. Also, since we assume the client is secure, we receive a stronger result if we move functionality from the client to a HWS which can be corrupted.

Thus, we let each HWS monitor the watchlist of its partner HWS with which it executes the inner protocol instances. Watchlists are set up as described in Chapter 3 but not between clients but between HWS that implement servers together. This simplifies the protocol and reduces overhead at no cost in security.

Note that the threshold t of the outer protocol sets a minimum number of HWS that are required to allow at least one corruption. This is the case, because all of the *SimS* that run on a corrupted HWS are corrupted as well. For example consider an outer protocol with a threshold of $t = n/4$. This would require more than four HWS as the corruption of just one HWS would immediately reach the threshold.

This leads to another tradeoff in the choice of parameters because more HWS allow more efficient parameters to be used. On the other hand, more HWS make the assumption that just one of them can be corrupted less and less reasonable. More details on the choice of parameters can be found in Section 6.6 where we analyze efficiency and parameters.

5.2.4 Security of the Interface Client

Here we just assume that the interface client is secure. In the following proof it is important that the interface client is there because the protocol has to present a single interface to the environment.

We argue that this assumption is justified because the interface client can be realized easily with an air gap.

The party is disconnected from the network until input shares are distributed to the HWS. Only then the party is connected to other potentially malicious networks. A party's output can be protected in the same way. The party closes its connections to other networks before attempting to reconstruct the final output. As long as some HWS remain uncorrupted, the adversary can not extract enough shares from that party to learn its input or output.

5.3 Proof of Security

To prove security of the protocol described above, we first need to define the notion of security we want to prove. It can be found in Theorem 5.1 which is naturally very similar in structure to the definition of security of the IPS complier. Inner and outer protocol are unchanged, but the inner structure of parties is different. Because of all theses similarities Theorem 5.1 is a modification of Theorem 1 described by Ishai et.al. in [IPS10]. This is also the case for the overall structure of the proof as well as many of the arguments used in it.

Theorem 5.1 *Let $\mathcal{F}$ be a m-party functionality that is realized by an MPC honest majority protocol Π with $n = \Theta(m^2k)$ and threshold $t = \Theta(n)$ for a statistical security parameter k. Let $\mathcal{G}$ be the functionality of the servers in Π, realized by the inner protocol $\rho^{\mathcal{F}_{OT}}$ in the $\mathcal{F}_{OT}$ hybrid model against two-step passive corruptions. Then the compiled and distributed protocol $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ described above, in the $\mathcal{F}_{OT}, \mathcal{F}_{Client}$ hybrid model securely realizes $\mathcal{F}$ against active static corruptions of entire parties and active adaptive corruptions of HWS that are limited by color as described in the previous chapter. If both Π and $\rho^{\mathcal{F}_{OT}}$ are statistically/computationally secure, then the compiled protocol inherits the same kind of security.*

5.3.1 Notation

First, we give an overview of the used variables and their meaning and dependencies. Most of the variables defined here will also be used in later chapters.

- n the number of *SimS*.
- k the number of *SimS* observed by the watchlist, must be strictly smaller than n.
- d the degree of the polynomial used for secret sharing. According to Lindell et.al. [LOP11] can be no more than $n/4$. Otherwise d should be as large as possible so we set $d = n/4$.
- l the block size of the secret sharing scheme.
- t the number of *SimS* that can safely be corrupted. $t = d - l - 1$.
- h the number of HWS.
- $\mathcal{T}$ the simulator for the combined protocol.
- $\mathcal{S}$ the simulator for the inner protocol.
- $\mathcal{G}$ is the ideal functionality of servers of the outer protocol, realized by the inner protocol ρ.

- Π denotes the outer protocol.
- P denotes servers of the outer protocol.
- $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ is the combined protocol for which security is proven in this chapter.

5.3.2 Overview

The general idea of the proof is the same idea used for the IPS compiler. The security of $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ is reduced to the security of the outer protocol Π. This is done by constructing a simulator that shows that $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ is indistinguishable from Π for any environment. The simulator $\mathcal{T}$ simulates an instance of $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ to an adversary $\mathcal{A}$ that can corrupt parties in it. $\mathcal{T}$ takes the role of a malicious party in an instance of the outer protocol Π. The outer protocol is then changed via a few indistinguishable hybrids to an $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ instance. Thus, no environment can distinguish $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ from the outer protocol Π which proves that $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ is as secure as the outer protocol.

Here it is important to keep in mind that Π is a honest-majority protocol. So it is essential that $\mathcal{T}$ does not corrupt more that t of the servers in Π during the simulation because it is only secure as long as a majority of the servers are honest. When more than t servers are corrupted, Π is no longer secure, and reducing $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$'s security to Π becomes pointless.

During the proof the additional property of security against *two-step passive corruption* of the inner protocol ρ^{OT} is required. Another important tool for $\mathcal{T}$ is that it provides the OT functionality for the adversary. Thus, the simulator knows all inputs including choice bits of those OTs.

Since $\mathcal{A}$ corrupts protocol participants, it expects to see combined protocol messages which are sent via HWS. In the plain outer protocol, in which $\mathcal{T}$ is the adversary, each party consists of just one device so mapping communication from the outer protocol to the correct HWS in $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ will be one of the main differences to the original IPS compiler.

Also, $\mathcal{A}$ expects to be able to do two different corruptions, entire parties and single HWS, as described in the previous chapter.

5.3.3 The Simulator

To begin with, we describe the simulator in the first setting, with an adversary against the compiled protocol $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ on one side and the plain outer protocol Π on the other side. Π will then be changed into the compiled protocol in a few hybrids.

Overview Simulation and Setting First, we give an overview over the simulator and its surrounding parties, also see Figure 5.4. On the left hand, there is the dummy adversary $\mathcal{A}$. A dummy adversary is fully controlled by the environment and only forwards messages to and from the environment.

The adversary's structure, as presented in Figure 5.4, is taken over from parties that were corrupted entirely. Via that interface, $\mathcal{A}$ (or the environment) expects normal messages of the combined protocol.

On the right side, there is the outer protocol instance with the client and several servers. The environment sets the input for all parties and receives their output via the interface client but can not directly interact with the servers or $\mathcal{F}_{Client}$because they do not provide input to any protocol.

One important tool for the simulator is that because we are in the OT-hybrid model it can simulate the OT functionality for the adversary. This means that $\mathcal{T}$ learns all input and output that $\mathcal{A}$ sends to and receives from any OT.

The simulators main difficulty is to translate messages between the two different protocols. Three main differences have to be simulated for that.

First, servers of the outer protocol are implemented by the inner protocol $\rho^{\mathcal{F}_{OT}}$ on the left side. The entities, outlined in red, cooperate to implement servers. Instead of actually executing the inner protocol, $\mathcal{T}$ uses simulators for the inner protocol to simulate $\rho^{\mathcal{F}_{OT}}$ instances for $\mathcal{A}$ and translate messages between protocols.

Second, the outer protocol does not have HWS whereas on the left side the many instances of the inner protocol are running on different HWS. To hide this difference, $\mathcal{T}$ has to assign the servers of the outer protocols to simulated HWS and corrupt them if that HWS is corrupted by the adversary. The Simulated HWS are outlined by dashed lines in Figure 5.4.

Finally, $\mathcal{A}$ expects to receive the internal states of inner protocol clients via the watchlists. To receive the necessary information to simulate this, $\mathcal{T}$ corrupts the corresponding servers in the outer protocol.

All these steps are described in more detail below.

Watchlist Setup At the start of the compiled protocol first, the watchlists are set up. Since each HWS receives watchlists only from the HWS of the other parties with which it simulates the servers using $\rho^{\mathcal{F}_{OT}}$, the servers of the outer protocol have to be assigned to HWS at this point. This is completely transparent for both $\mathcal{A}$ and the clients of the outer protocol because this only determines which servers are corrupted if the adversary chooses to corrupt an entire HWS.

The watchlist channels are set up by exchanging keys for the channels via OT so that each

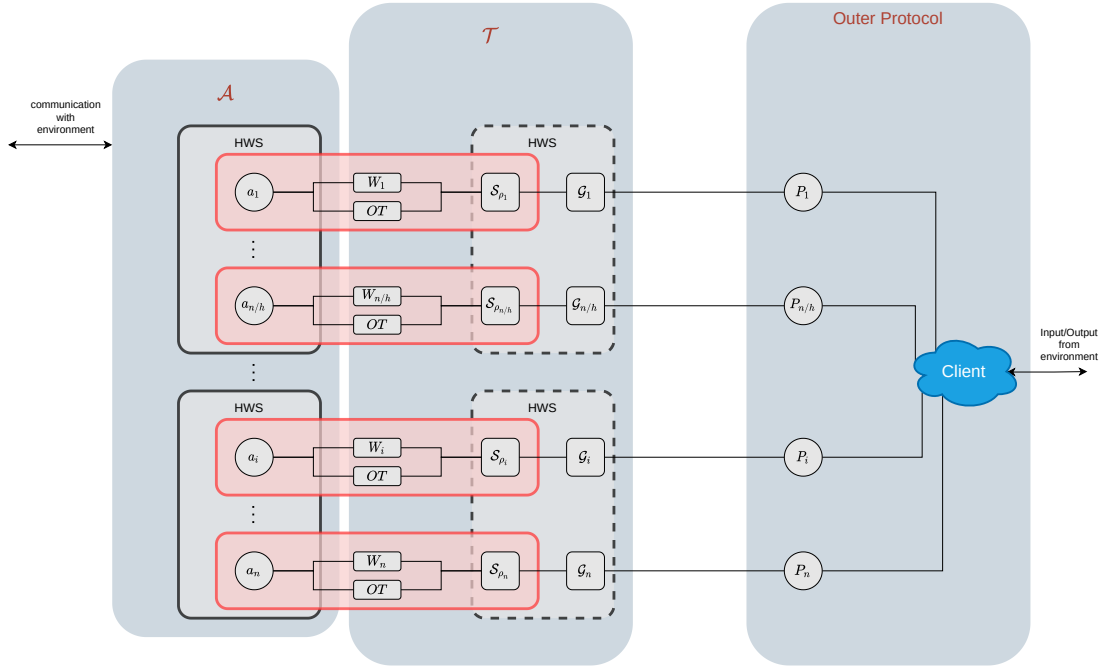

Figure 5.4: Overview of the Simulator with dummy adversary and HWS of corrupted parties on the left. The plain outer protocol instance is on the right.

party can only decipher k watchlists, just as in the original IPS compiler described in Chapter 3. However, the watchlist setup is now evenly distributed among a party's HWS. Since there are no HWS in the outer protocol and the simulator simulates them, it also takes over the watchlist setup and exchanges keys via OT with the adversary. Here the simulator uses the fact that it simulates the OT functionality for the adversary to learn which watchlist channels $\mathcal{A}$ has access to. $\mathcal{T}$ then corrupts the corresponding servers in the outer protocol to learn their internal state and continues to execute them according to protocol but uses their internal state to simulate a $\rho^{\mathcal{F}_{OT}}$ instance which is used to feed the watchlist channels for $\mathcal{A}$. The servers that are not watched by $\mathcal{A}$ are not corrupted. Since $\mathcal{T}$ does not know their internal state, it can not be reported over the watchlist. So the simulator just sends random ciphertexts on all the watchlist channels the adversary can not read. At this point, k servers in the outer protocol are corrupted which leaves a remaining threshold of $t - k$ servers that may be corrupted in the remainder of the proof.

Simulation of Protocol Messages During the protocol, four different kinds of messages have to be simulated by $\mathcal{T}$.

First, to simulate client-to-client messages, $\mathcal{T}$ has to split them up into shares or reconstruct

them depending on the direction in which they are sent. Client-to-Client messages coming from $\mathcal{A}$ have to be reconstructed from the shares sent by $\mathcal{A}$'s HWS. $\mathcal{T}$ then performs a consistency check and passes the message on to the client in the outer protocol. Messages sent to the adversary's client, first, have to be split up into shares by $\mathcal{T}$ who then passes on sets of shares to $\mathcal{A}$'s HWS, as described above.

Second, to generate messages of the inner protocol that emulates servers of the outer protocol, $\mathcal{T}$ uses simulators $\mathcal{S}_i$ of the inner protocol. Messages from the outer protocol are translated to the combined protocol using $\mathcal{S}$ and then passed on to $\mathcal{A}$ and vice versa. $\mathcal{S}$ expects communication with an adversary and an ideal functionality of the inner protocol $\mathcal{G}$. The simulator $\mathcal{T}$ has to emulate them both to properly use $\mathcal{S}$by simply forwarding messages. The adversary's messages are emulated by forwarding $\mathcal{A}$'s inner protocol messages to $\mathcal{S}_i$, and $\mathcal{G}_i$ is emulated by relaying messages to and from server in the outer protocol, as shown in Figure 5.4.

The third type of messages that have to be simulated are the watchlist channels. Their simulation is described above, together with their setup.

Finally, $\mathcal{T}$has to be able to abort if $\mathcal{A}$ is detected when cheating in a $\rho^{\mathcal{F}_{OT}}$ instance. For this, $\mathcal{T}$ uses its full access to OTs. Since all keys for watchlists are transmitted via OT, the simulator learns all watchlist keys. So $\mathcal{T}$also learns everything $\mathcal{A}$ sends on its watchlists. With this information, $\mathcal{T}$ can perform full consistency checks on all messages sent by $\mathcal{A}$ because the Simulator sees both messages and the choice bits of OTs. Honest clients however, see only one input message of OTs they receive so if only one input message is inconsistent, they only have a $1/2$ chance to detect the inconsistency if for example an 1 out of 2 OT is used. An honest client also can not detect inconsistencies if it does not have access to the corresponding watchlist.

So $\mathcal{T}$ can detect more inconsistencies than an honest client and thus, also has to simulate a clients consistency checks to determine whether or not they would detect an inconsistency. This mainly depends on which watchlists that client could read. $\mathcal{T}$ can simulate this by choosing watchlists for clients initially. Then, any inconsistencies in servers that are not watched by any client are ignored.

Note that a client that does not detect an inconsistency because it can not read the watchlist, will not be able to detect future inconsistencies in that server as well. If the clients would detect an inconsistency, $\mathcal{T}$ aborts the protocol. If the clients in the outer protocol do not detect the inconsistency, $\mathcal{T}$ corrupts the corresponding server to continue the simulation.

Corrupted inner Protocol Instances If no client detects inconsistencies in $\mathcal{A}$'s inner protocol messages, we have a passively secure protocol instance that faces an active adversary.

At this point, we have no guarantees that $\mathcal{S}$ can simulate this. To continue the simulation, we need the special property of ρ^{OT}, the security against *two-step passive corruption*. This property allows a static adversary to corrupt all remaining honest parties (of that inner protocol instance) at once, at any time in the protocol. If that happens, $\mathcal{G}$ sends the inputs of all honest parties to the simulator $\mathcal{S}$ which then has to simulate a consistent internal state of the parties.

Since $\mathcal{T}$ simulates $\mathcal{G}$ for $\mathcal{S}$, $\mathcal{T}$ now sends all state of that server to $\mathcal{S}$ (via $\mathcal{G}$). This includes inputs and previous messages.

With that input, $\mathcal{S}$ simulates an inner protocol instance with according state. $\mathcal{T}$ then lets the adversary interact with that inner protocol instance. Through this interaction, the adversary potentially learns all internal state of the corresponding server. But, by the security of the inner protocol, it can not distinguish the reconstructed inner protocol instance from a real one.

This allows $\mathcal{T}$ to simulate consistent states of servers when the adversary cheats undetected.

HWS Corruptions By the definition of the color model, $\mathcal{A}$ can corrupt an entire HWS. If that happens, the adversary gains full control over that HWSits further behavior is in the adversary's hands and does not have to be simulated. It is sufficient to simulate the internal state of that HWS, which can then be turned over to the adversary.

Simulation of inner protocol instances is done in the same way as when they are corrupted by undetected cheating.

The adversary also expects to learn the watchlist keys of those servers and can use them to decrypt messages, sent previously on their watchlist channels. So $\mathcal{T}$ has to make them up so that they are consistent with the internal state of the server and ciphertexts that were sent on their watchlists before. This requires the use of a *non-committing* encryption scheme. This is why OTPs are used to encrypt watchlists. Since OTPs are perfectly secure, we do not need a secure encryption scheme as an additional assumption for Theorem 5.1.

This allows $\mathcal{T}$ to make up keys so that the previously sent random ciphertexts decrypt to consistent reports of internal state of the corresponding servers.

Finally, shares of client-to-client messages that were relayed via the HWS have to be simulated. For client-to-client messages where the adversary was either sender or receiver, $\mathcal{T}$ already knows those shares as it generated them when simulating those messages. Shares of messages between honest clients can simply be made up in this moment if the adversary is static. To detect this, the adversary would have to either corrupt a corresponding client or break the share's privacy.

5.3.4 Hybrid Argument

In the next step, we use a hybrid argument to show that the scenario described above with simulator $\mathcal{T}$ is indistinguishable from an actual instance of the combined protocol for any environment. As mentioned before, the environment can interact with the protocol in two ways. It has full control over the dummy adversary, and it chooses all the client's inputs and receives their output.

Hybrid 0: Replacing Inner Servers with Ideal Functionalities The first hybrid is almost the same to the setting described above, with the only difference that all the servers $P_i, i \in \{1, ..., n\}$ in the outer protocol are replaced with their ideal functionalities $\mathcal{G}_i$. So $\mathcal{T}$ no longer needs to emulate the $\mathcal{G}_i$ for $\mathcal{S}_i$. Also, the clients need to be changed so that they interact with $\mathcal{G}_i$s instead of P_is. The environment can not detect this change because servers in the outer protocol do not provide input or output and thus, the environment can not directly interact with them. $\mathcal{G}_i$ and P_i are completely identical with the exception of how the adversary controls when messages are sent (directly or via the communication channel). The output behavior of clients is not changed. $\mathcal{T}$ can mask this difference because it simulates communication channels for $\mathcal{A}$. Since $\mathcal{T}$ can not corrupt an ideal functionality $\mathcal{G}$, it now directly uses the corruption interface of the $\mathcal{G}_i$s to get the internal state of corrupted servers. This corruption interface is still guaranteed by the security against two-step passive corruptions of the inner protocol. The internal state, that is provided by the ideal functionality, is still sent to $\mathcal{S}$ for translation into the combined protocol as before.

Hybrid 1: Introducing HWS In the next hybrid, HWS are introduced in the outer protocol. The servers of the outer protocol are now grouped together into several HWS. Each HWS simply encapsulates an equal fraction of the $\mathcal{G}_i$ but still allows other parties to interact with them as before by forwarding messages.

These changes can not be detected by any environment because first $\mathcal{T}$ can continue its simulation unchanged, and second, both servers and HWS do not provide input to the outer protocol so the environment can not directly interact with them.

The simulation of watchlists and corruptions of entire HWS are still simulated as described above. $\mathcal{T}$ does not need to actually corrupt any HWS to simulate its corruption. All information $\mathcal{T}$ needs to simulate the corruption of an entire HWScan be learned by corrupting all servers (or $\mathcal{G}_i$) that run on that server. So now, that there are real HWS present, the simulator can still simulate HWS corruptions as in previous hybrids where $\mathcal{T}$ completely made up the HWS.

When the adversary corrupts an entire HWS, it expects to learn all internal state of the

HWS which is more than just the state of all servers that run on that HWS. The state of a HWS also contains watchlist keys and messages that were previously sent and received on watchlist channels. Messages sent on watchlist channels exist in encrypted and unencrypted form (if the HWS knows the corresponding key).

The only way to distinguish simulated HWS from real ones, is to detect that messages sent on watchlists that the adversary can not read, are not reports of internal server state. There are two ways to accomplish that. First, the adversary might simply break the encryption and see just randomness on the watchlist. This is obviously in contrast to the security of the used encryption scheme. The second way to find out, would be that the adversary might corrupt a HWS, learn its encryption key for watchlists and use it to decrypt previously sent watchlist messages. Since watchlist channels are encrypted with OTPs which are both perfectly secure and *non-committing*, neither of the two ways can be successful.

Note that the number of HWS the adversary can corrupt is limited by the color corruption model.

In this hybrid, we also have to introduce the interface client along with the ideal client functionality $\mathcal{F}_{Client}$. The environment now sends input and receives output from the interface client. The interface client then shares input to the HWS of that party. The HWS simulate inner servers, manage watchlists and use the ideal client functionality to execute the outer protocol. For that, the HWS forward messages between the servers/$\mathcal{G}_i$s and $\mathcal{F}_{Client}$. When the outer protocol ends, the HWS receive shares of the final output from $\mathcal{F}_{Client}$. They forward those shares to the interface client, which reconstructs the final output and sends it to the environment. The ideal functionality is by definition indistinguishable from a real client, so the environment can not detect a difference in the final output. The interface client to which the environment provides input and receives output is also indistinguishable, as it is a device that takes input and then sends output to the environment. But $\mathcal{T}$ also has to be able to simulate the shares that the HWS hold. Here we use that shamir shares have perfect privacy as long as there are not enough shares for reconstruction. Since, by assumption the adversary can not corrupt enough HWS to learn enough shares to reconstruct them, $\mathcal{T}$ can just make up random shares when it has to simulate internal state of a HWS.

Hybrid 2 - n+2: Replacing Ideal Functionalities with the Inner Protocol The next n hybrids replace the $\mathcal{G}_i$ one by one with instances of the inner protocol ρ_i^{OT} such that in the jth hybrid $\mathcal{G}_1$ through to $\mathcal{G}_j$ are replaced by ρ_1^{OT} through to ρ_j^{OT}. So the hybrids j and $j+1$ differ in how server $j+1$ of the outer protocol is implemented, either by an ideal functionality or by the inner protocol.

For servers that are implemented by an ideal functionality $\mathcal{T}$ simulates corruptions and

watchlists as before. For servers implemented by the inner protocol it is no longer necessary to simulate the watchlists because the corresponding HWS expects to send on and receive watchlist channels for its $\rho^{\mathcal{F}_{OT}}$ instances. $\mathcal{T}$ just forwards those messages between the adversary and the HWS in the outer protocol.

As we already argued above, any adversary that can distinguish between simulated watchlists and actual watchlists sent by a HWS, can be used to break either the *non-committing* property or the privacy of the encryption scheme used for the watchlists.

If the adversary now requests the corruption of an entire HWS that is part of at least one $\rho^{\mathcal{F}_{OT}}$ instance, $\mathcal{T}$ corrupts that HWS to learn the internal state of the inner protocol instances as well as that HWS's keys for watchlist channels. If that HWS also still consists of some ideal functionalities, their internal state must be simulated by $\mathcal{T}$ as described above.

Two consecutive hybrids now only differ in one instance of a server. As long as the differing server is attacked only by passive corruptions, two neighboring hybrids are indistinguishable because ρ^{OT} realizes functionality $\mathcal{G}$ secure against passive corruptions. But since we want to show that the compiled protocol is secure against active corruptions, we also have to consider active corruptions. Since a passively secure protocol does not make any security guarantees in the presence of an active adversary, the adversary can potentially learn all its internal state here. Previously this had to be simulated by $\mathcal{T}$ using $\mathcal{S}$. Since there is an actual protocol instance of $\rho^{\mathcal{F}_{OT}}$, now $\mathcal{T}$ only has to forward protocol messages to $\mathcal{A}$. The *two-step passive corruption* property of ρ^{OT} guarantees that these forwarded messages are indistinguishable from messages generated by $\mathcal{S}$.

Since two consecutive hybrids only differ in one instance of $\rho^{\mathcal{F}_{OT}}$, any machine that can distinguish two neighboring hybrids could be used to distinguish interactions with $\mathcal{S}$ from interactions with ρ^{OT}, which is in conflict with the security against *two-step passive corruption* of ρ^{OT}.

All in all these steps make it easier to simulate because the simulator can just forward messages instead of simulating servers that are now implemented by $\rho^{\mathcal{F}_{OT}}$. The last hybrid then is exactly the combined protocol $\Phi_{\Pi,\rho}^{\mathcal{F}_{OT}\mathcal{F}_{Client}}$ and requires no simulation at all, only the forwarding of messages, which is exactly what we want.

So overall, we make the following conclusions. Because all the hybrids in between them are indistinguishable, the setting of hybrid 0 which consists of the outer protocol and $\mathcal{T}$ is indistinguishable from the last hybrid which consists only of an adversary and the real combined protocol. Since we assume that the outer protocol is secure, any attack on the combined protocol could be used to distinguish some consecutive hybrids (and thus, also the first and last hybrid). However, we have just shown that said hybrids are indistinguishable, thus, no such attack can exist.

5.3.5 The Simulator Corrupts no more than t Servers

In a last step, it has to be shown that $\mathcal{T}$ does not need to corrupt more servers of the outer protocol than its honest majority assumption allows. If the threshold t of the outer protocol is exceeded in the simulation, the entire proof does become worthless because then the security of the compiled protocol is reduced to the security of an honest majority protocol with a broken honest majority assumption. No security guarantee for the compiled protocol is gained from such a reduction.

There are three sources of server corruptions that have to be considered.

First, the simulator has to corrupt k servers to simulate watchlists for the adversary.

Second, up to n/h servers have to be corrupted because their corresponding HWS is corrupted by the adversary. The first two categories overlap since the watchlists are distributed evenly between the HWS. For simplicity, we assume they do not overlap, which results in a higher upper bound of the number of corrupted inner servers.

Third, we have to consider the servers that have to be corrupted because the adversary cheats undetected in inner protocol instances.

The number of corruptions due to the second category can be exactly determined before the protocol even starts. Of the n inner servers n/h run on each HWS. So if one HWS is corrupted, n/h inner servers are corrupted along with it. After considering those corruptions, we have a remaining threshold of $t' = t - (\frac{n}{h})$ to account for corruptions from sources one and three.

From here on, we can follow the same argument as in the original IPS paper. Half of t' is used to compensate for corruption via watchlist the other half is used for corruptions via undetected cheating.

Now we fix the parameter h to a constant. We also use that $t \in \Theta(n)$ because the outer protocol's threshold t is a constant fraction of the number of servers n. So t' is also in $\Theta(n)$ and thus we can make the following approximations.

The probability that the adversary can cheat undetected in too many servers is:

$$(1 - k/n)^{t'/2} \tag{5.1}$$

because $t' \in \Theta(n)$ we can use the fact that

$$\lim_{y \to \infty} (1 - x/y)^y = e^{-x}$$

so

$$\lim_{n \to \infty} (1 - k/n)^{\Theta(n)} = e^{-\Omega(k)} \tag{5.2}$$

Thus, there exists an n such that the probability that $\mathcal{T}$ has to corrupt more than t *SimS* is negligible in k. So it is possible to securely instantiate the combined protocol. More details on instantiation, choice of parameters and efficiency can be found in Chapter 6.

5.3.6 On fully Adaptive Security

The above proof works for static corruption of entire parties and adaptive remote hacks of HWS which are restricted by the color model. We make the conjecture that if all its pieces, including the inner and outer protocols, are secure against adaptive attacks, the combined protocol will be secure against adaptive corruptions of entire parties as well. For this it will probably be necessary that the protocol that implements the client can be simulated even if all parties are corrupted in the same sense of the two-step passive corruption of the inner protocol, so that shares that the HWS received during the protocol can be simulated in a coherent way.

6 Implementing the Client

In the previous chapter, we managed to split up parties of the IPS compiler so that they can be partially corrupted and remain secure. In that first step, the implementation of the client was ignored for the sake of simplicity the client was modeled as an ideal functionality.

This has the advantage that we can use any MPC protocol that is secure in the UC-Framework to implement the client. In this chapter, we propose using the MPC protocol SPDZ to implement the client, and we analyze its efficiency. For an adequate efficiency analysis, we also have to choose inner and outer protocols. For that, we use the protocols that Ishai et.al. propose along with the IPS compiler because they are designed to work well with it [IPS10]. We choose SPDZ mainly for the efficiency of its online phase.

6.1 SPDZ

SPDZ is an MPC protocol that uses a preprocessing assumption to achieve good efficiency [Dam+13]. Its online phase can compute arithmetic circuits fast but requires a complicated offline phase to set up. We use SPDZ in a context where initially devices trust each other so we can skip the offline phase and only use the online phase with the preprocessing done by just one trusted party. The 2013 version of SPDZ [Dam+13] is more efficient than its earlier version [Dam+12] at the cost of being secure only against covert adversaries. This is because the offline phase is secure only against covert adversaries while the online phase remains actively and adaptively secure. So skipping the offline phase also means that this version of SPDZ is still actively, adaptively secure.

Since we use SPDZ only within a party to realize its client functionality, assuming a trusted setup is very reasonable. We do not assume trust between the different parties of the combined protocol.

Here we briefly describe SPDZ as far as necessary, for a detailed description see [Dam+13].

6.1.1 Preprocessing Phase

The SPDZ online phase requires auxiliary data that is consumed in the online phase during the actual computation. The auxiliary data is essentially beaver triples with a MAC. The

Initialize: On input $(init, p^k)$ from all parties, the functionality activates and stores the modulus p.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with varid a fresh identifier, the functionality stores $(varid, x)$.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y \mod p)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y \mod p)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, x)$ and outputs it to the environment. If the environment inputs OK, then x is output to all players. Otherwise $\perp$ is output to all players.

Figure 6.1: Ideal functionality for $\mathcal{F}_{AMPC}$ exactly as defined in [Dam+12]

preprocessing phase is defined as an ideal functionality $\mathcal{F}_{PREP}$. For its exact definition see Damgard et.al. [Dam+13]. While they propose a protocol that securely realizes $\mathcal{F}_{PREP}$, we do not need it. Since all participants of one SPDZ instance will be devices of the same party, we can use the initial trust within that party to securely set up the auxiliary data. After receiving input, the interface client will execute $\mathcal{F}_{PREP}$ and send enough auxiliary data to the HWS along with their input shares. The HWS will then use the auxiliary data in the SPDZ online phase to realize $\mathcal{F}_{Client}$.

6.1.2 Arithmetic Multi-Party-Computation

SPDZ realizes MPC in form of arithmetic MPC. Damgard et.al.[Dam+12] define that via an ideal functionality $\mathcal{F}_{AMPC}$ which can be found in Figure 6.1.

This functionality allows its parties to compute additions and multiplications over a field of size p so any arithmetic circuit can be implemented with SPDZ.

This means that the loss of efficiency by implementing the client via SPDZ can be measured in additions and multiplications.

6.1.3 Online Phase

On a high level, the SPDZ online phase uses multiplication triples to locally multiply shared values. A MAC is added to achieve active security.

First, each party P_i needs a MAC key which is a share α_i of a secret value α. The share for party P_i of a value a is now defined as the tuple $(a_i, \gamma(a)_i)$ with $\gamma(a) := \alpha * a$. The MAC for a public value a is defined as $\gamma(a)_i = a * \alpha_i$. Values can be partially opened by revealing the a_is but not the shares of the MAC.

Multiplications require multiplication triples. These are shared triples a, b and c with $c = a*b$. Multiplication triples are given by the preprocessing and are consumed by multiplications.

To compute a multiplication $z_i = x_i * y_i$ with the shared triple (a_i, b_i, c_i), first the values $\varepsilon = x - a$ and $\gamma = y - b$ are partially opened. With those values, each party can now locally compute $z_i = c_i + \varepsilon * b_i + \gamma * a_i + \varepsilon * \gamma$. The MACs are used to verify results.

Addition and multiplications by constants can be performed locally on these shares. This means they require no communication between protocol participants and no auxiliary information, which makes them essentially "free".

Damgard et.al. [Dam+13] also mention that it is possible to use SPDZ for some operations that go beyond the operations of $\mathcal{F}_{AMPC}$. They explain that with some additional preprocessed data it is possible to realize the protocols described in [DI06]. Of those protocols, we will use the constant round equality check. This will allow us to perform equality checks in SPDZ with constant communication overhead.

6.2 IPS Outer Protocol

Since efficiency will mainly depend on the number of multiplications the simulation of the client of the outer protocol requires, we must fix the outer protocol for a meaningful analysis. The IPS outer protocol, proposed along with the IPS compiler by Ishai et.al [IPS10], is an obvious choice since it is designed to work well with the IPS compiler.

6.2.1 Verifiable Secret Sharing

The IPS outer protocol requires a verifiable secret sharing scheme as a building block [IPS10]. In addition to shares, the dealer also provides a proof that ensures all shares are consistent. To do that, the dealer shares a set of random blinding vectors $r^1, \ldots r^\sigma$, as depicted in Figure 6.2. The servers then draw randomness by a Coin-Toss among each other. That randomness $R^h \in \{0,1\}^m$ defines a linear combination of the shared vectors and is sent back to the dealer. The dealer now broadcasts those σ linear combinations masked by the blinding vectors

$u^h = r^h + \sum_{i=1}^{m} R_i^h v_i$. Each server can now verify that every u^h is consistent. If a server detects an inconsistency, a complaint is broadcast. If the number of complaints is within the threshold, the shares are accepted and otherwise rejected.

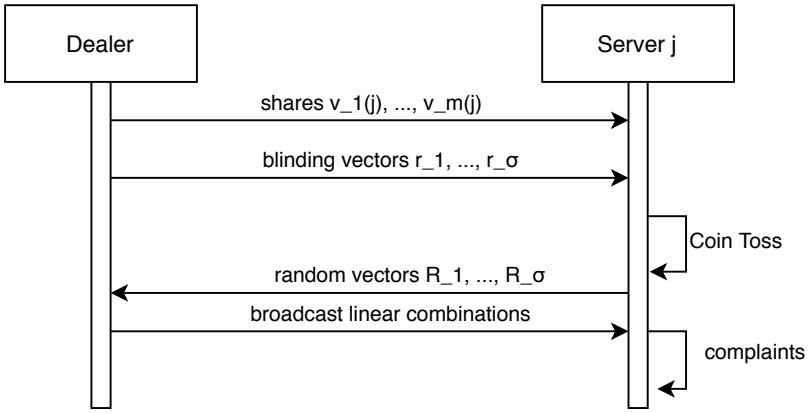

Figure 6.2: The VSS scheme used in the combined protocol. The scheme was proposed in [IPS10].

6.2.2 IPS Protocol

Here we describe the IPS-Protocol from [IPS10]. First, both parties use the VSS scheme to share their input. The input is arranged in blocks, in a way that neighboring blocks are NANDed together. The circuit is restricted to NAND gates because they can easily be computed by a local multiplication, see Figure 6.3. Bob also shares a random mask for later use. The servers then locally compute the NAND operation on their shares and add the mask. This multiplication doubles the degree of the polynomials of the shares. So parameters for the VSS have to be chosen so, that even with doubled degree they can still be reconstructed. All shares of those intermediate results are then sent to Alice. She then reconstructs the shares. Since each server sends a share, there are more shares than necessary to reconstruct a degree $2d$ polynomial, which is used for error detection. The additional shares are used to reconstruct the secret multiple times from different shares. The protocol is only continued if all reconstructions yield the same result. This prevents the adversary from manipulating values undetected.

The resulting value is then shared again with reduced degree of d (instead of $2d$) and arranged so the blocks can be used for the next layer of the circuit.

The servers now locally remove the mask with the unblinding blocks provided by Bob and are then ready to compute the next layer.

Algorithm 1 IPS protocol

This is the the IPS outer protocol, almost as described in protocol A.2 in [IPS10]. Also see Figure 6.3. *SimS* run on several HWS. Messages to and from *SimS* got to/come from their respective HWS. One important change we made here is that the client receives its input and sends the final output in shares. Also see Figure 6.4 for the definition of the ideal functionality the client of this protocol has to realize.

1. **Initialization:**
   - The client receives its input in form of shares.
   - The client recovers the shares.
2. **Sharing Input to *SimS***
   - First, the client draws enough randomness for the mask.
   - The client uses the VSS to share input and mask for the next layer of the circuit.
3. **Computing next circuit layer**
   - *SimS* compute NAND and add mask locally on their share
   - *SimS* send their shares to the client
   - This is not verified the client accepts a sufficient amount of any evaluation points of fitting polynomials as input.
4. **Preparation of the next circuit layer by client**
   - The client recovers the intermediate result by reconstructing the shares since there are more shares than necessary, additional shares are used for error detection. If all shares are not points on the same polynomial, the protocol is aborted.
   - The blocks of the intermediate result are rearranged for the next round by the client.
   - The client shares the rearranged intermediate result to the *SimS*using teh VSS scheme.
   - The client shares unblinding blocks to *SimS*.
   - *SimS* remove mask and are now ready for the next layer.
   - If there are more layers to compute, go to 3.
5. **Output**
   - The *SimS* send their unmasked share to all clients that are supposed to receive output.
   - The client recovers the result.
   - The final output of the client is shares of the result.

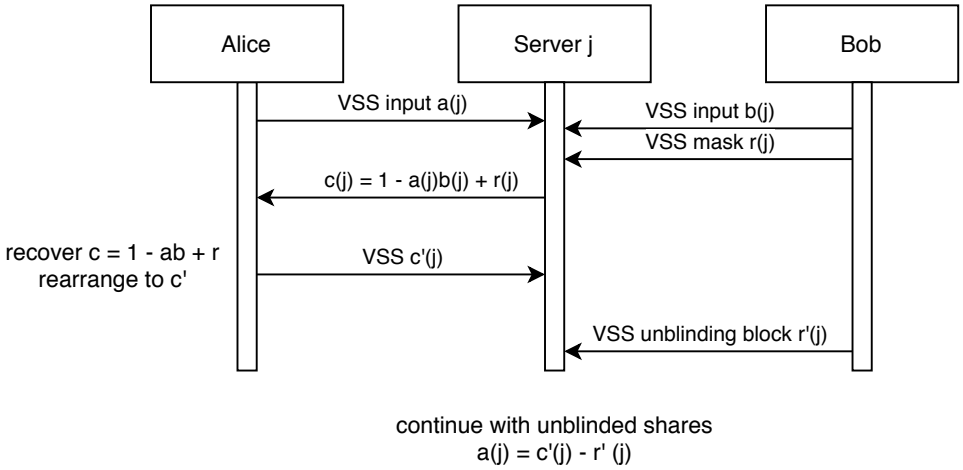

Figure 6.3: The IPS Protocol as proposed in [IPS10]. It serves as outer protocol in the combined protocol.

6.3 The Combined Protocol

The combined protocol consists of a short setup phase followed by the online phase that is mainly the same as in the previous chapter. An overview can be found in Figure 6.5. In the setup phase, the interface client receives the party's input, generates auxiliary data for the HWS and sends input shares and auxiliary data to the HWS. We assume that the interface client is secure as long as not the entire party is corrupted. The client functionality of each party is now realized by an SPDZ instance that is run by the HWS of that party.

Initialization First, all HWS that belong to a party have to collaborate to initialize the full protocol. They receive their input shares and auxiliary data from the interface client. With that they can start a SPDZ instance to realize $\mathcal{F}_{Client}$.

Client-to-Client Messages Client-to-Client messages are forwarded via the HWS, as described in the previous chapter. To make sure that the corrupted HWS can not manipulate the message, secret shares are used.

Main Protocol Execution After the initialization, the main protocol starts. The HWS now start a SPDZ instance to realize $\mathcal{F}_{AMPC}$ with what $\mathcal{F}_{Client}$ is realized. That requires some communication between the HWS which is marked by the green arrows in Figure 6.5.

The HWS provide input, execute and receive output from $\mathcal{F}_{Client}$. Note that different HWS will often have to get different outputs from the SPDZ instance because it is often used to

1. Init: The functionality receives parameters from all participants (HWS). This includes the number n of shares that will be needed, the block size l of the shares and the degree d of polynomials used to share. The functionality also receives the number of HWS h that will participate in this instance and the prime modulus p that defines the field in which calculations will be done. If all participants agree, the functionality outputs success and failure otherwise.

2. Input: The functionality expects one share from each participant as input. So it receives h evaluation points of a polynomial of degree $h-2$. So there are more shares than required to reconstruct the secret. The functionality reconstructs the secret with $h-1$ shares and checks if the last evaluation point is on the polynomial. If not it aborts. This corresponds to step 1 in the IPS protocol.

3. Share Input: After receiving input and reconstructing input, the functionality re-shares the input into n verifiable shares and sends them to the participants. So each participant receives n/h evaluation point of polynomials and corresponding verification data. From this point on, the functionality performs steps 4 and 5 in rounds until the circuit is completely executed. In each round, it waits for all required input from all participants before continuing.

4. Share mask: The functionality draws a random mask and sends n verifiable shares of it to the participants, as in step 3.

5. Preparation of the next layer: If there are more layers to compute on input of n shares from the participants, the functionality checks if all n shares are points on a degree $2d$ polynomial. If not the functionality aborts. Otherwise, it recovers the secret and rearranges the recovered value according to the circuit that is to be computed. The functionality expects an equal amount of shares from each participant and will not accept more shares from a single participant. But shares are not verified, anything that looks like a evaluation point of a polynomial will be accepted. The functionality's output in this step is once again, n evaluation points of a degree d polynomial along with corresponding verification data. Each participant receives n/h of the verifiable shares. The functionality then continues with steps 3 and 4. This corresponds to step 4 of the IPS protocol.

6. Output: If there are no more layers to compute, the functionality also expects n shares as in step 5. They are checked for errors recovered as before. The result is then split into h shares of which each participants gets one as final output. So the final output of the functionality is h evaluation point of a degree $h-2$ polynomial. This corresponds to step 5 of the IPS protocol.

Figure 6.4: The ideal functionality $\mathcal{F}_{Client}$ that a client of the IPS protocol has to realize. This functionality works in rounds. It awaits input for a round from all participants, then computes the round and provides output. Then again the next round is only started after input from all participants has arrived.

replace the dealer of a VSS scheme, where each HWS must receive different shares. While $\mathcal{F}_{AMPC}$ sends output to all parties, output can be encrypted with a OTP that is chosen by the party that should receive the output alone. This is specified later.

The communication that belongs to the inner protocol that simulates servers for the outer protocol is shown in red.

Finally, communication between the *SimS* and the clients is shown in blue. Note that both the client and the *SimS* are only logical entities, simulated by the inner protocol (*SimS*) or SPDZ ($\mathcal{F}_{Client}$). The communication between those logical entities is forwarded via HWS. To forward a message between *SimS* and $\mathcal{F}_{Client}$ a HWS receives the message as output from a protocol and uses it as input for the protocol that realizes the other entity.

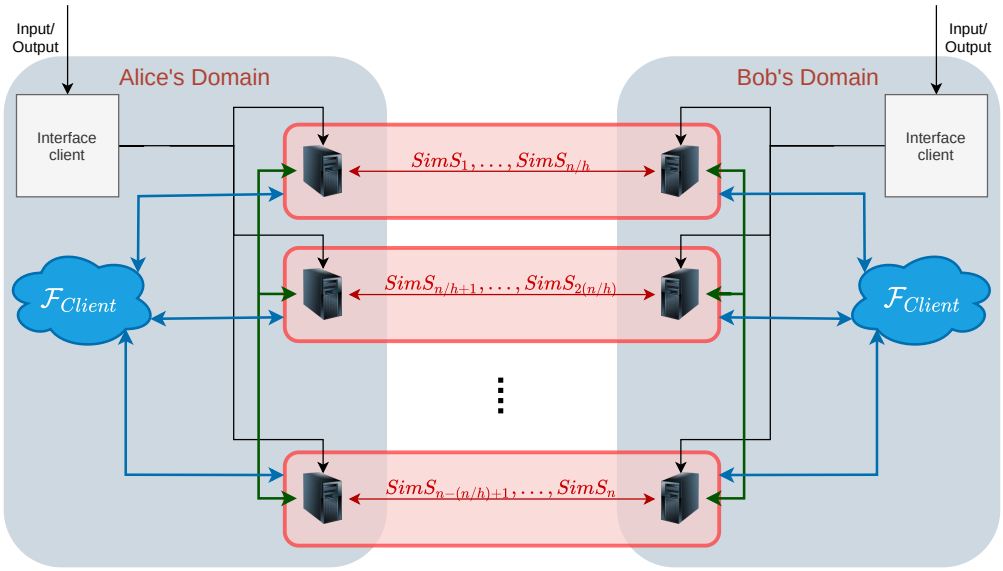

Figure 6.5: The IPS protocol combined with SPDZ Client

Additional Modifications to the IPS protocol In the original IPS outer protocol the masks that hide intermediate results are provided only by the clients that do not do the preparations for the next layer. In our case however, all clients will add the mask to prevent the adversary from learning anything from corrupted HWS.

Threshold Since SPDZ is secure as long as at least one party is honest, the threshold is not dependent on SPDZ. Every corrupted HWS allows the adversary to also corrupt the corresponding *SimS*. Thus, the upper bound on possible corruptions of HWS is determined by the outer protocol that is used.

Algorithm 2 VSS with SPDZ Dealer

The VSS protocol from Figure 6.2 with the dealer implemented by HWS using SPDZ. The HWS have previously distributed multiplication triples. Receivers are the *SimS*. This follows closely protocol 2 from [DI06] with some modifications due to the dealer being a SPDZ instance. Since the dealer is implemented by SPDZ we describe teh dealer's operations in terms of $\mathcal{F}_{AMPC}$ as defined in Figure 6.1 and the subroutines described below.

1. **Re-Share Input**
   - If the value that is supposed to be shared to the *SimS* is not yet input to $\mathcal{F}_{AMPC}$, the HWS input it now.
   - The HWS use the Share subroutine to generate shares for all *SimS*.
   - The HWS use the SingleOutput subroutine to learn the shares for *SimS* that they simulate and forward the shares to their *SimS*.
2. **Share Blinding-Vectors**
   - HWS use the DrawRandom subroutine to draw blinding vectors $r^1, ..., r^\sigma$.
   - The HWS use the Share and SingleOutput subroutines to share blinding vectors to the *SimS* as in step 1.
3. ***SimS* generate random vectors** $R^1, ..., R^\sigma$
   - The *SimS*Coin-Flip among themselves to generate random vectors.
   - The random vectors are sent to the HWS.
   - The HWS input the random vectors into $\mathcal{F}_{AMPC}$.
4. **Proving**
   - HWS compute $u^h = r^h + \sum_{i=1}^{m} R_i^h v_i$ with $h \in 1, ..., \sigma$ in $\mathcal{F}_{AMPC}$.
   - The HWS use the output command of $\mathcal{F}_{AMPC}$ to send all the u^h to all *SimS*.
5. **Complaining**
   - Each *SimS* checks if all u^h are consistent with R^h and the shares it received.
   - For all inconsistencies found, a complaint is broadcast to the other *SimS*.
   - If the number of complaints does not exceed the threshold, the *SimS* accept their shares

6.4 A Protocol for the Client

We now describe an MPC protocol that allows the HWS to realize the ideal functionality $\mathcal{F}_{Client}$ of the IPS outer protocol defined in Figure 6.4. The IPS outer protocol is described in algorithm 1. Since we want to use SPDZ to implement $\mathcal{F}_{Client}$ and SPDZ realizes $\mathcal{F}_{AMPC}$ as defined in Figure 6.1, throughout this chapter we use $\mathcal{F}_{AMPC}$ as a Black-Box to realize $\mathcal{F}_{Client}$. By the UC-security of SPDZ and the composition theorem this remains secure when $\mathcal{F}_{AMPC}$ is replaced by SPDZ.

This protocol uses a few assumptions that are directly implicated by assumptions we already mentioned before.

- Preprocessing: All HWS receive enough SPDZ preprocessing Data and shares of input to the circuit that is to be computed.
- Arithmetic circuit evaluation: All HWS have access to an ideal functionality $\mathcal{F}_{AMPC}$. This functionality is realized by a SPDZ instance within each party.
- Each honest party consists of several HWS of different colors of which at most 1/4th is corrupted via remote corruptions.

Besides the functions of $\mathcal{F}_{AMPC}$, the protocol will use additional subroutines which are easily obtained from Black-Box calls to functions of $\mathcal{F}_{AMPC}$.

- **SingleOutput:** On input $(SingleOutput, varid, P_i, otp)$ from one HWS P_i and input $(SingleOutput, varid, P_i, ?)$ from the other HWS, the functionality outputs the value $(varid + otp \mod p)$ to all HWS. This requires one input, one add and one output operation, and achieves output that just HWS P_i can read.
- **RecoverShares:** On input $(recover, varid_1, ..., varid_i)$ from each HWS P_i, the shares stored in $varid_1, ..., varid_i$ are recovered using lagrange interpolation. This requires several add and multiply operations. This subroutine will usually be given more shares than necessary to reconstruct the secret. The additional shares will be used for error detection. For that, the subroutine checks if the additional shares lie on the same polynomial that is defined by the other shares.
- **Share:** On input $(share, varid_s, g)$ from each HWS, the value stored in $varid_s$ is shared into g shares which are stored in new varids $(varid_1, ..., varid_g)$. The VSS scheme described above is used for this.

- **DrawRandom:** On input $(DrawRandom, varid_1)$ every HWS locally draws long enough random value. The HWS then use the input and add operation to add up the random values of all HWS and store the result in $varid_1$.

Below we describe the protocol that implements the client functionality.

Algorithm 3 client protocol

This protocol implements the client using SPDZ to realize $\mathcal{F}_{AMPC}$.

1. **Initialization:**
   - All HWS call the Initialize function of $\mathcal{F}_{AMPC}$
   - All HWS input their input shares into $\mathcal{F}_{AMPC}$.
   - All HWS call the RecoverShare subroutine to reconstruct the input. For the error detection it is checked if the additional shares lie on the same polynomial. To realize that in SPDZ the previously mentioned constant round equality checks are used.
2. **Sharing Input to *SimS***
   - All HWS call the Share operation to share the input into enough shares for all *SimS*.
   - All HWS use n SingleOutput operations to output the shares to the HWS that simulate the corresponding *SimS*.
3. **Sharing mask to *SimS***
   - Each HWS uses the Input operation to input enough randomness for the mask into $\mathcal{F}_{AMPC}$.
   - The mask of each HWS is added together to one mask, using 5 Add operations.
   - The mask is shared and sent as output as in step 2.
4. **Computing next circuit layer**
   - The HWS wait for the *SimS* to compute NAND and add mask locally on their share.
   - The HWS receive masked results from their *SimS* and input them into $\mathcal{F}_{AMPC}$.
   - All HWS use the RecoverShare operation to recover masked results as before. This includes the error detection that requires equality checks.
5. **Preparation of the next layer by client**
   - HWS rearrange intermediate result for the next round. This is done locally by permutating the $varid$s that will be input into future $\mathcal{F}_{AMPC}$ commands.
   - The intermediate results are shared to the *SimS* as in step 2.
   - The HWS share unblinding blocks to *SimS* as in step 2.
   - The *SimS* can remove the mask from the result with this information.
   - If there are more circuit layers to be computed, continue with step 3.
6. **Output**
   - If there are no more layers, the *SimS* remove the mask from their share and send them to their HWS.
   - The HWS input the shares of their *SimS* into $\mathcal{F}_{AMPC}$.
   - The HWS use the reconstruct, share and SingleOutput commands to re-share the final output to the HWS.
   - Technically, the functionality of $\mathcal{F}_{Client}$ is fulfilled at this point. Obviously, the HWS still have to send their shares of the output to the interface client to conclude the entire computation.

6.5 Proof of Security

We now argue that the above protocol securely realizes the client functionality of the combined protocol when all calls to $\mathcal{F}_{AMPC}$ are realized by SPDZ. For that, we briefly describe a simulator to show that the SPDZ implemented client in algorithm 3 is indistinguishable from the client of the plain outer protocol in algorithm 1 which defines $\mathcal{F}_{Client}$.

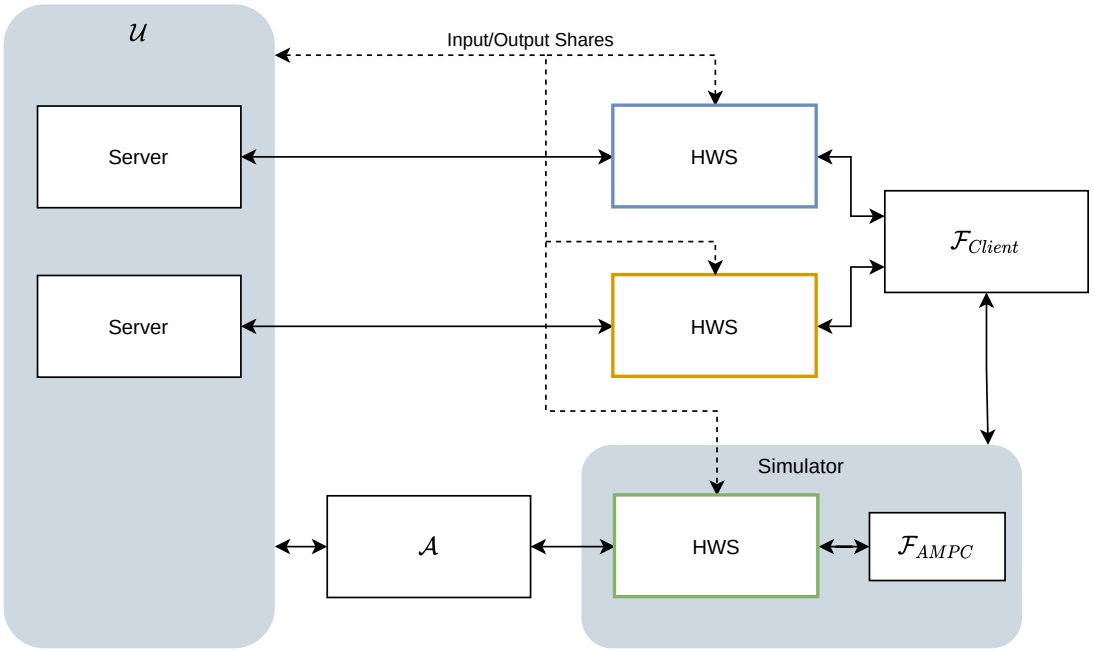

Figure 6.6: The setting of the proof with the environment $\mathcal{U}$ on the left. It sets initial input of HWS and receives their final output. The environment als controls the dummy adversary $\mathcal{A}$. The simulator serves as adapter between $\mathcal{A}$ and $\mathcal{F}_{Client}$.

Consider the following setting, which is also depicted in Figure 6.6. We only look at the protocol that realizes the client. The outer protocol is context surrounding the client, which is handled by the environment. Since we can not make assumptions on the environment, there may or may not be an outer protocol. It consists of several HWS, although only two are depicted in the figure, at least 4 are required in the combined protocol. Some HWS may get adaptively corrupted by the dummy adversary that is fully controlled by the environment.

The environment also chooses the input which is sent directly to the HWS because we only look at $\mathcal{F}_{Client}$, not at the combined protocol. Honest input for the HWS would be shares of input for the outer protocol. Every round of the outer protocol the environment expects messages from the HWS to the servers as output. Answers of the servers are sent from the

environment to the HWS as input. In the end, the environment expects output from each HWS which consists of shares of output of the outer protocol or an abort message if at any point input to $\mathcal{F}_{Client}$ was rejected by it.

Besides taking input and sending output, the honest HWS basically just relay messages between servers of the outer protocol and $\mathcal{F}_{Client}$. The outer protocol servers are not simulated by the HWS, instead they are run by the environment, so there are no watchlists.

The order of computations in $\mathcal{F}_{Client}$ as defined in Figure 6.4, is very static. Each round, the HWS can only provide input and then receive output, but they can not influence what computations are performed. The environment controls the inputs to $\mathcal{F}_{Client}$ since it provides input to the HWS, and they will just forward it.

The simulator serves as an adapter between corrupted HWS and the adversary. And it has to convince the environment that the corrupted HWS interact with the real protocol, not with $\mathcal{F}_{Client}$. The real protocol uses $\mathcal{F}_{AMPC}$ as subroutines, so the simulator runs $\mathcal{F}_{AMPC}$ by itself. If the adversary wants to corrupt an honest HWS, the simulator corrupts that HWS first to learn its internal state. Then the simulator uses that internal state to simulate the internal state of a HWS that interacts with $\mathcal{F}_{AMPC}$ and then sends that to the adversary. The simulator can then interact with $\mathcal{F}_{Client}$, but only in the role of the corrupted HWS. Meaning, it sends input in place of the adversary and receives output meant for the adversary, but it has no access to messages exchanged between $\mathcal{F}_{Client}$ and honest HWS.

We now have to argue that the simulator can simulate all the steps necessary to implement the client, by using $\mathcal{F}_{AMPC}$.

When a HWS is newly corrupted, the simulator learns its input shares and all messages sent to and from $\mathcal{F}_{Client}$. All operations already done via $\mathcal{F}_{Client}$ are now simulated via $\mathcal{F}_{AMPC}$, as described below. With this, the simulator can generate a view of the corrupted HWS as if it had interacted with the real protocol and not $\mathcal{F}_{Client}$. This view is sent to the adversary. The simulator then continues the simulation and lets the adversary control the behavior of the corrupted HWS.

The simulator can not interact with the honest HWS, so it must make up their inputs to $\mathcal{F}_{AMPC}$. Their input does not have to be completely simulated since it is not sent to the adversary by $\mathcal{F}_{AMPC}$. But $\mathcal{F}_{AMPC}$ always awaits the same command with fitting parameters from all participants before a command is executed, so the simulator must keep track of what type of command would be called next by honest HWS. Simulating this is no problem since the circuit that is to be computed as well as the code for the client are public knowledge.

Because of that, the simulator only has to simulate calls to $\mathcal{F}_{AMPC}$ and subroutines in such an order that implements $\mathcal{F}_{Client}$, as described above. As long as at least one HWS is honest, only commands that implement $\mathcal{F}_{Client}$ will be called by all HWS. Since $\mathcal{F}_{AMPC}$ just ignores

any commands that are not called by all parties, the simulator can just ignore calls to $\mathcal{F}_{AMPC}$ that are not protocol conform. The same argument can be made for all *varid* and server id P_i parts of inputs the adversary makes to $\mathcal{F}_{AMPC}$. Thus, the simulator must only be able to simulate if the input of the adversary is malicious. If the adversary deviates from that order of commands, the simulator just ignores those commands since no honest HWS would agree with that order of commands.

Since the environment can not change the order of commands, its only way of distinguishing $\mathcal{F}_{AMPC}$ from $\mathcal{F}_{Client}$ is by manipulating inputs and comparing outputs.

At this point, we observe that all input to $\mathcal{F}_{Client}$ is always (non verifiable) shares. All of the output of $\mathcal{F}_{Client}$ is verifiable shares. The only exception is input that is malformed, so that it can not be interpreted as a share. In this case, $\mathcal{F}_{Client}$ outputs an error message. The client protocol has almost the same input and output behavior, with the only exception being the DrawRandom command that allows HWS to input some randomness for a coin toss.

The simulation of $\mathcal{F}_{Client}$ now works as follows. As explained above, we only have to consider a faithful order of commands. Anything else is just ignored and does not have to be simulated.

When the adversary sends any input to $\mathcal{F}_{AMPC}$, the simulator just saves it but does not take action or send output for now.

The adversary can continue to send different commands for which the simulator only checks if they are valid, if not they are ignored. The input of the adversary to $\mathcal{F}_{AMPC}$ consists of important parts which is the actual input to the input command (which are shares if the input is honest). The rest of the adversary's input we consider unimportant. The unimportant input includes varids and ids of HWS, see Figure 6.1 for details. The simulator uses the unimportant inputs only to check if the adversary's inputs are valid or if they can be ignored. Important parts of the input will have to be forwarded to $\mathcal{F}_{Client}$ if the previous commands are valid.

The simulator only has to do some work when the adversary calls the output command in a valid way. If the previous commands were valid, the simulator now sends the previously saved important part of the input to $\mathcal{F}_{Client}$ which will answer with a verifiable share or ignores it if the input was malformed.

The simulator then simply forwards that answer to the adversary after adding some cosmetics, so that it looks like the real protocol answer to the adversary's output command. The important part of the output looks also as if it came from the real protocol because a series of valid commands performs the same computation as $\mathcal{F}_{Client}$, and only valid series of commands result in output to the adversary. Here it does not matter if the important part of the input was honest or not, as long as it can be interpreted as shares somehow. If the input can be interpreted as shares, both the real protocol and $\mathcal{F}_{Client}$ will accept it.

However, the environment might try to detect that the adversary's random input was ignored. It might wait until $\mathcal{F}_{Client}$ sends the unblinding blocks to the servers. Since the environment controls all servers, at this point it can reconstruct both the mask and the intermediate result.

But since in the client protocol the values of all HWS are added modulus p, any one of the inputs can completely determine the result. So, to detect that the adversary's randomness was not used to generate the mask, the environment would have to learn the random inputs of all HWS of a party. By assumption, not all HWS can be corrupted if the party is not corrupted entirely, thus, this can never happen.

So no environment can distinguish the client protocol in algorithm 3 from $\mathcal{F}_{Client}$ as long as at least one HWS remains honest.

6.5.1 On Adaptive Corruptions Part 2

We finally want to follow up on our conjecture on fully adaptive corruption in the previous chapter. For this, we argue that simulation is still possible if all HWS are corrupted. In this case the simulator first corrupts all remaining honest HWS. This means that the simulator has access to all shares that were sent to and output by by $\mathcal{F}_{Client}$. So the simulator can reconstruct those shares and learns not only the real input of that party but also the random masks that were generated by $\mathcal{F}_{Client}$. Because just one random input of a HWS can completely determine the result of the generateRandom subroutine, the simulator can make up that random input of the freshly corrupted HWS now in a way that is consistent with everything the adversary might have learned already. Thus, the simulator can send a consistent internal view of HWS to the adversary, even if all HWS are corrupted.

A party that looses all its HWS is of course no longer an honest party. The adversary learns its input and output, and can fully control its behavior. But since this case can still be simulated, it does not break the combined protocol. Thus, we believe that the combined protocol is secure against adaptive corruptions of entire parties as well as adaptive corruptions based on color.

6.6 Efficiency Analysis

More security usually comes at a cost, and the modification of the IPS ompiler proposed in this work is no exception. In this section, we analyze the additional overhead caused by our modifications compared to the standard IPS ompiler. We begin with a rough analysis in the Big-O notation.

6.6.1 Analysis in Big-O Notation

The additional cost caused by SPDZ is dominated by the number of multiplications that have to be calculated to implement the client. While technically all computations in SPDZ (addition, multiplication by constant and multiplication) are done locally, only multiplications require multiplication triples and additional network communication. From this perspective, all overhead of SPDZ is network and memory overhead.

Thus, we consider additions and multiplications by constant to be free. To determine Network and memory overhead, we count the multiplications that have to be calculated by SPDZ.

The client has to perform the following computations during the execution of one round of the outer protocol which executes one layer of the circuit.

Generating and Reconstructing Shares This is both done by a lagrange interpolation. Usually, lagrange interpolation is quadratic in the polynomial's degree. However, in our case, all interpolation points and the points in which the polynomials have to be evaluated are constant and publicly known. So the fundamental polynomials only have to be computed once in the entire protocol, or alternatively they can be pre-computed. This leaves only additions and multiplications by constant which makes lagrange interpolations "free".

However, when dealing a VSS a linear combination has to be computed, this requires multiplications linear in the number of *SimS n*.

When reconstructing shares, the error detection also requires constant round equality check. This also incurs overhead that is linear in n.

Overall, that means the communication and memory overhead of both generation and reconstruction of shares is linear in n.

Permutation of Input The client also has to permutate the input blocks in order to prepare them for the next layer of the circuit. This can be seen as a vector of input blocks being multiplied by a permutation matrix. Since the circuit that is to be calculated is fixed in advance, the permutation matrices can be pre-computed as well. If the permuation matrixes are precomputed, this requires only multiplications by constant, so no network and memory overhead is caused by this.

It is also possible to implement this by having each HWS permutate the varids used in calls to $\mathcal{F}_{AMPC}$, accordingly, which also incurs no relevant overhead.

Overall Overhead of Computing a Circuit We now combine the factors mentioned above to determine the overall overhead of computing a circuit C of size $|C|$. When computing one gate of a circuit, the client has to perform the following actions.

1 VSS the input of that layer to the *SimS* ($O(n)$).

2 VSS the mask for the intermediate result to the *SimS* ($O(n)$).

3 reconstruct the result it receives from the *SimS* ($O(n)$).

4 permutate as preparation for the next layer ($O(1)$)

5 VSS new input ($O(n)$)

This yields a combined overhead of $O(4n)$ in communication and memory per layer. Also note that only one client has to perform the reconstruction, permutation and VSS of the new input steps in each layer. The other parties only have to provide their mask, so for all but one party the overhead is reduced to $O(2n)$. In Big-O notation the difference is not important, though.

Overall, this leaves us with a communication and memory overhead linear in the number of *SimS* and the size of the circuit $O(4n \cdot |C|)$.

Computation Overhead While computation overhead is almost negligible in comparison to even local network overhead, we still want to give a rough estimate.

While operations that can be done locally do not cause network or memory overhead, the operations also have to be performed on both the data and the MAC. Thus, all operations in SPDZ require roughly twice the computations.

6.6.2 Choice of Parameter

As shown above, the overhead of implementing the client in SPDZ is mainly dependent on the number n of *SimS*. So, to fully understand the overhead caused by SPDZ, we have to analyze how the parameter n has to be chosen to achieve a given security, and how our modification to the IPS compiler affect that choice.

Since we compensate for *SimS* that are corrupted because their HWS are corrupted with the threshold of the outer protocol, the number of *SimS* and the size of the watchlists have to be bigger, compared to standard IPS in order to achieve the same level of security.

Here we analyze how parameters have to be chosen to achieve a statistical success chance of 2^{-40} for the adversary. This number seems both arbitrary and too low. However, note that the given success rate for the adversary is statistical, not computational. This means, the adversary

can not improve its chances with more computational power. This type of security of the combine protocol is inherited from the inner and outer protocol. Also Lindell et.al. [LOP11] use the same number for a similar analysis of the plain IPS compiler.

Overview Parameters The combined Protocol has several different parameters that can also depend on each other. Thus, those parameters have to be chosen carefully to make the protocol work. In this section, we describe our findings on reasonable ranges of parameters.

First, we consider findings by Lindell et.al. [LOP11] on the choice of parameters of the original IPS protocol without SPDZ and HWS.

There are four parameters to consider: The degree d of the polynomials used for secret sharing, the block size l of the secret sharing scheme, the number of tolerable corruptions t and the number of servers simulated by the inner protocol n.

The degree d of the polynomials used for secret sharing is set to $d = n/3$ by Ishai et.al. [IPS10]. This is because during multiplications the degree is doubled but must still be small enough to be recovered from n shares. Lindell et.al. propose an optimization of the IPS compiler that tightens this constraint to $d < n/4$. In the case of our combined protocol, this optimization is outweighed by the increased overhead caused by the tightened constraint. We will describe this in more detail below. Besides those limitations, d should be as large as possible because that allows for a larger block size or a higher corruption threshold. So we will continue with $d = n/3$.

The block size l of the secret sharing scheme has to be smaller than d, but otherwise increases the efficiency of the protocol with higher values. But since the threshold is $d - l$, a larger l also decreases the amount of corruptions the outer protocol can tolerate. This increases the number of servers n that are required what again reduces efficiency. Also note that setting l higher than the number of gates in a circuit layer does not provide any efficiency benefits when computing that layer. This dependency on the circuit layout makes it hard to determine generally good values for parameters. However, for a fixed l we can compute optimal choices for n and k that achieve a given statistical success chance for the adversary (2^{-40} in this case).

The threshold t is entirely dependent on the number of shares the adversary can know without being able to reconstruct the secret. This determines $t = d - l - 1$.

Analysis of Parameters In the IPS compiler several parameters interact with each other, so we can not give one ideal choice for all parameters. So instead we analyze how the parameters interact and give some advice on how to choose them, depending on the context in which the protocol is used. In this section, we only consider the two party case, but the same approach can be used to analyze higher numbers of parties. So in the following, m is generally set to 2.

Since $n = O(mk)$ for the number of parties m and watchlist size k, we first try to find the factor a so that $n = amk$. We then can find an optimal ratio between a and k that satisfies a given error probability while minimizing n for efficiency.

To break the protocol, the adversary has to corrupt more than t *SimS*. Because of the watchlists, the adversary has already $(m-1)k$ servers corrupted (assuming a worst-case scenario where the adversary has full control over all but one party). So the adversary has to corrupt

$$(t/amk)amk - (m-1)k$$

additional *SimS*. Since there are $n = amk$ servers and the one honest client has k of them on its watchlist, the adversary can break the protocol without being detected with probability

$$\frac{\binom{amk-k}{(t/amk)amk-k}}{\binom{n}{(t/amk)amk-k}}. \tag{6.1}$$

The number of clients is fixed already, which leaves us with three parameters to choose. If we fix t and a error propability, we can use eq. (6.1) to determine values for a and k that minimize k while achieving the desired error probability.

With this approach Lindell et.al. conclude that for an error probability of 2^{-40},

$$a \approx 2n/t * (1 - 1/m)$$

is optimal in the original IPS compiler. Their numeric experiments confirmed their conclusion.

Numeric Experiments We conduct similar numeric experiments for the compiled protocol to be able to compare both versions of the IPS compiler with the following results, shown in Figure 6.7.

The Plot shows the required k on the X-axis and n on the Y-axis to achieve a statistical security of 2^{-40} for standard IPS in green and our combined protocol with 5 HWS of which one can be corrupted in red. Within both colors, smaller values of l are the lower curves.

For a fixed l, the optimal choices of n and k are their values at the local minimum of the curve.

For standard IPS, depending on l, n is in ranges between 1500 and 2000 and k in ranges of 150 to 170.

With 5 HWS n needs to be roughly between 6000 and 8000 while k is about 400, rising with higher l.

So using 5 different HWS increases the overhead of simulating *SimS* roughly by a factor of 6.

Also, roughly 3 to 4 times more watchlists have to be checked every round.

Note here that overhead of setting up the watchlist channels is only dependent on n, not on k since all parties have to set up a watchlist channel for every *SimS* with every other party, whether or not that channel can be read. The watchlist size k only changes the success rate of the erasure channels used to set up watchlists, not the number of OTs that have to be performed.

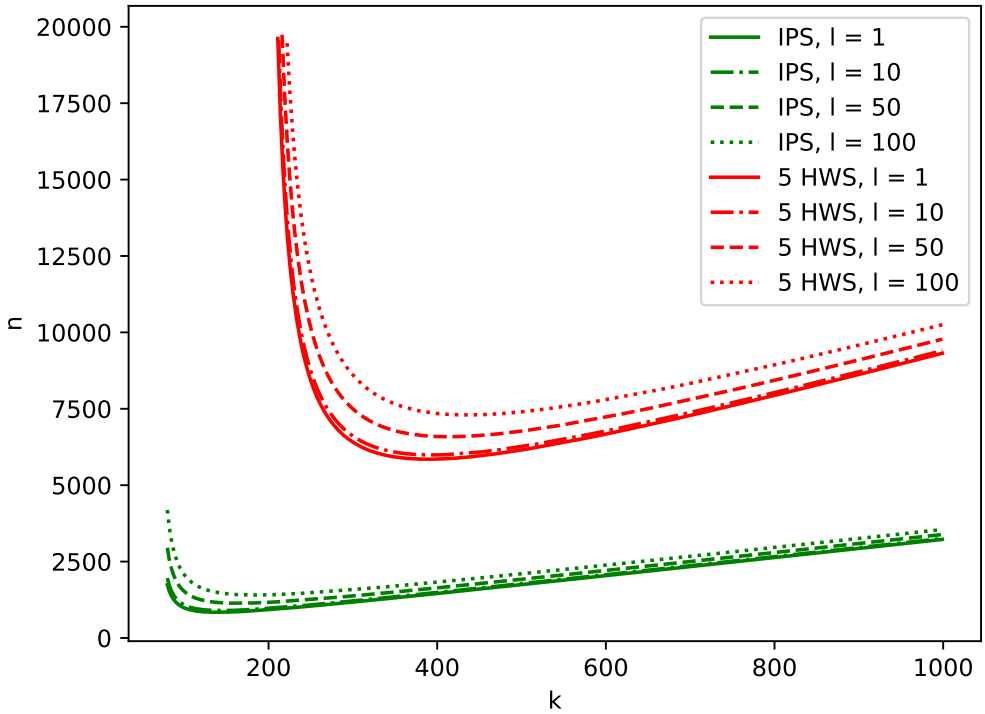

Figure 6.7: Comparison Standard IPS to distributed IPS with 5 HWS. n is the number of *SimS*, k is the size of the watchlist and l is the block size of the used secret sharing scheme.

Since the additional overhead from implementing the client in SPDZ is linear in n, this adds an additional constant factor of about 6 to the overhead.

On the Number of HWS Taking into account, the number of HWS results in another tradeoff. Assuming just one HWS can be corrupted, we still have to make the choice of how many HWS should be placed per party. More HWS result in a lower fraction of all the *SimS* getting corrupted if one HWS is lost. This in turn allows for a smaller number of n and k while

achieving the same security. So more HWS result in a more efficient protocol. For comparison see Figure 6.8 which compares required parameters with 4, 5 and 6 HWS. It can be seen that when using only 4 HWS n and k have to be more than twice than that required for 5 or 6 HWS.

On the other hand, a setting with less HWS is easier to set up and maintain. It is also more reasonable to rely on fewer different hardware and software.

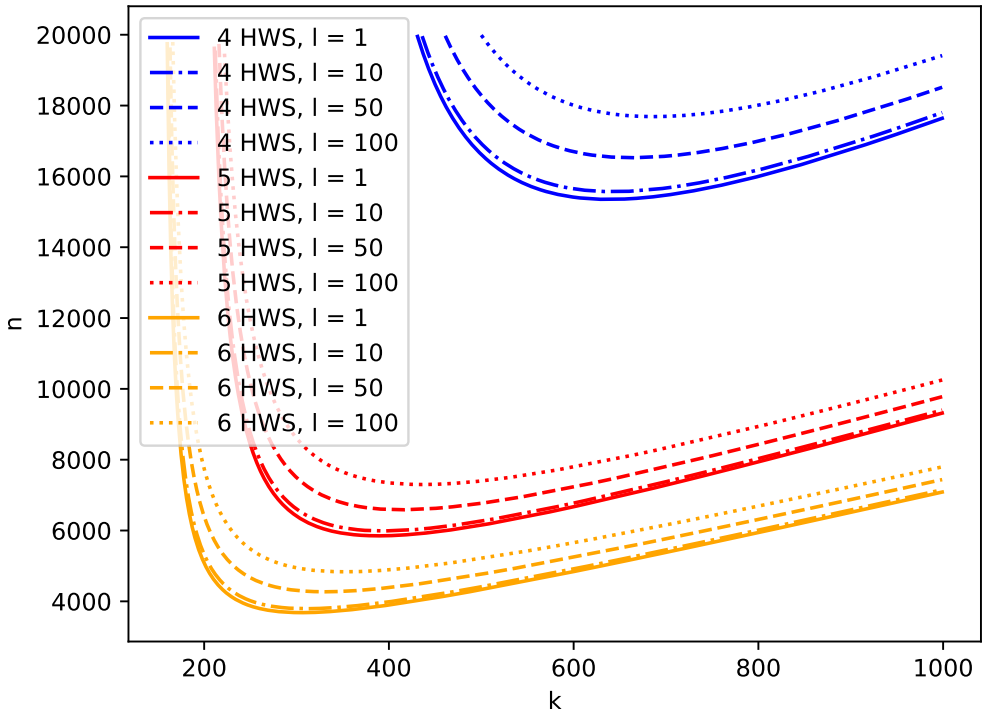

Figure 6.8: Comparison of parameters with 4, 5 and 6 HWS. n is the number of *SimS*, k is the size of the watchlist and l is the block size of the used secret sharing scheme.

6.6.3 Overhead from Redundancy

When considering the overhead of this approach, one must not forget the additional cost of ensuring that the hardware assumptions hold. Not only does each party require more hardware, they must also make sure that the assumptions of the color model hold. Each server must be made up of different hardware and operating systems, which does not necessarily make them more expensive but the system overall is more difficult to maintain. Each HWS also needs

independent implementations of all the protocols that are used which requires a significant initial investment. Maintaining several implementations is also much more expensive.

6.6.4 Scaling with the Number of Parties

We have not yet taken into account how the number of parties affects efficiency. The steps we list above that have to be performed for every layer of the circuit have to be done by all parties, but they can all perform them in parallel. So the changes we made to the IPS compiler should scale well with the number of parties.

However, setting up the watchlists requires communication in $O(n * m^2)$. Since our combined protocol requires larger n than the standard IPS compiler setting up the watchlists is also more expensive.

Another more subtle side effect is in the choice of parameters. Our analysis of parameters above is considers the two-party case. For more parties, parameters have to be chosen higher, because the adversary can corrupt more parties, and thus get access to more watchlists, so the number of servers has to be chosen higher and watchlist size has to be increased as well. This effect is already present in the standard IPS compiler.

But, as mentioned before Lindell et.al. [LOP11] present some optimizations to the IPS compiler. One effect of those optimizations is that the adversary does not get access to more watchlists if there are more parties. As explained above, those optimizations do not scale well with our changes. It is still unclear how exactly all those optimizations interact with our changes. More work is needed to understand this tradeoff and when the optimizations are worth it in combination with our changes to the IPS compiler.

7 Conclusion

The main goal of this work is to achieve a stronger notion of security for MPC protocols. For this purpose, we propose a new, stronger corruption model for adversaries in the UC framework. This corruption model is meant to be used in a setting where parties consist of several different devices. In the new model, the adversary has two different corruption types. The first type is a standard active corruption that corrupts an entire party at once, even if it consists of several devices. This is supposed to model very strong but expensive attacks, where the adversary has, for example, physical access to its targets. Corruptions that are caused via remote hacks are modeled differently, here the adversary has to corrupt each device separately. These corruptions are limited by the hardware and software of the devices. We model this by assigning colors to devices. If two devices have overlapping hardware or software components they get the same color. If they consist of completely different components they are assigned different colors. This models that for a remote hack the adversary needs an open vulnerability, which only works for specific hardware or software. So in our model the adversary can only corrupt a limited number of different colors via remote hacks.

Then we use the assumption that the adversary can not corrupt several diverse, redundant systems, to build a generic protocol that is secure against the new corruption model. This protocol has the benefit that an honest party that consists of several devices can loose a limited amount of devices to the adversary. Even with some corrupted devices, the party is still able to participate as an honest party in the protocol and its in- and output remains private. With this new corruption model we can show that our generic protocol can still give security guarantees even if the last remaining honest party is partially corrupted.

Previous MPC protocols such as the IPS compiler or SPDZ remain secure only as long as at least one party remains honest. We improve this threshold in the sense that even the last remaining honest party can be partially corrupted. So overall, this approach allows a larger percentage of parties to be corrupted while still performing secure computations.

While increased security comes at a price, we were able to show that this modification of the IPS compiler causes only linear overhead, compared to the original IPS compiler.

Additionally, most of the linear overhead is memory and communication in the local network where communication is much faster and cheaper compared to a wide area network. Wide

area network overhead caused by the increased number of servers that have to be simulated is only constant.

Building on the generic protocol we propose a more specific protocol that uses already existing protocols as components for the generic protocol. We also provide an analysis on the choice of parameters for a secure instantiation with those candidates. While there is no set of parameters that is optimal for every use case, we describe how parameters should be chosen, given a fixed circuit that shall be computed.

All in all, we achieve our goal of modeling a stronger adversary and we provide both a generic and a more specific protocol, that allows secure MPC in the presence of that adversary. With linear overhead, this is at least a good proof-of-concept upon which can be improved in future work.

In conclusion this work shows that it is possible to use diverse redundancy to give formal security guarantees for MPC protocols.

8 Future Work

As discussed in the efficiency analysis, some optimizations for the IPS compiler do not work well with our modifications. However, those modifications are interesting, especially if more than two parties are present. More work is required to fully understand if, and under what circumstances those optimizations are worth it.

Other ideas worth exploring are alternatives for the IPS compiler. Our general approach of using MPC to fortify individual parties of MPC protocols should work with any MPC Protocol. However, a detailed analysis would be necessary to determine the practicality of other protocols.

Even if other protocols are shown to be more efficient than the IPS compiler, we believe that SPDZ is a very good choice when it comes to the distribution of single points of failure within parties. To the best of our knowledge, SPDZ is currently among the most efficient MPC protocols. In our case, we can even skip the most expensive part of SPDZ, the offline phase, because we can reasonably assume initial trust within parties.

An exception to this might be an outer protocol with a client that can not be easily implemented in terms of $\mathcal{F}_{AMPC}$. While theoretically every computation can be done in $\mathcal{F}_{AMPC}$, it can get very inefficient if the computation is difficult to do with just multiplications and additions. In that case it would probably be beneficial to replace SPDZ with another protocol that realizes another computation model. So if the outer protocol is changed, alternatives to SPDZ should also be explored.

Bibliography

[Bro+18] Brandon Broadnax et al. “Fortified Universal Composability: Taking Advantage of Simple Secure Hardware Modules”. In: (2018).

[Can01] R. Canetti. “Universally composable security: A new paradigm for cryptographic protocols”. In: *Annual Symposium on Foundations of Computer Science - Proceedings* (2001).

[Dam+12] Ivan Damgård et al. “Multiparty computation from somewhat homomorphic encryption”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2012).

[Dam+13] Ivan Damgård et al. “Practical Covertly Secure MPC for Dishonest Majority – or : Breaking the SPDZ Limits Secure Multi-Party Computation Goal : compute”. In: *Esorics* (2013).

[DI06] Ivan Damgård and Yuval Ishai. “Scalable secure multiparty computation”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2006).

[Her07] Amir Herzberg. “Folklore, practice and theory of robust combiners”. In: *Journal of Computer Security* 17 (2007), pp. 159–189.

[IPS10] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. “Founding cryptography on oblivious transfer”. In: *Proceedings of the Annual ACM Symposium on Theory of Computing* (2010).

[Ish+07] Yuval Ishai et al. “Zero-Knowledge from Secure Multiparty Computation”. In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 21–30.

[KK07] Kaoru Kurosawa and Takeshi Koshiba. “Direct Reduction of String (1, 2)-OT to Rabin’s OT.” In: *IACR Cryptology ePrint Archive* 2007 (2007).

[KL15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman and Hall Crc Cryptography and Network Security Series)*. Chapman and Hall CRC, 2015.

[Lin17] Yehuda Lindell. "How to simulate it – A tutorial on the simulation proof technique". In: *Information Security and Cryptography* (2017).

[Lin21] Yehuda Lindell. "Secure Multiparty Computation (MPC)". In: *Communications of the ACM* (2021).

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. "The IPS compiler: Optimizations, variants and concrete efficiency". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2011).

[LS04] Bev Littlewood and Lorenzo Strigini. "Redundancy and diversity in security". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3193 (2004), pp. 423–438.

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