

Hardening the Security of Server-Aided MPC Using Remotely Unhackable Hardware Modules

Dominik Doerner,¹ Jeremias Mechler,² Jörn Müller-Quade³

Abstract:

Garbling schemes are useful building blocks for enabling secure multi-party computation (MPC), but require considerable computational resources both for the garbler and the evaluator. Thus, they cannot be easily used in a resource-restricted setting, e.g. on mobile devices. To circumvent this problem, server-aided MPC can be used, where circuit garbling and evaluation are performed by one or more servers.

However, such a setting introduces additional points of failure: The servers, being accessible over the network, are susceptible to remote hacks. By hacking the servers, an adversary may learn all secrets, even if the parties participating in the MPC are honest.

In this work, we investigate how the susceptibility for such remote hacks in the server-aided setting can be reduced. To this end, we modularize the servers performing the computationally intensive tasks. By using data diodes, air-gap switches and other simple remotely unhackable hardware modules, we can isolate individual components during large parts of the protocol execution, making remote hacks impossible at these times. Interestingly, this reduction of the attack surface comes without a loss of efficiency.

Keywords: multi-party computation; garbling schemes; universal composability; fortified universal composability

1 Introduction

Secure Multi-Party Computation (MPC) allows mutually distrusting parties to jointly perform tasks on private data via a protocol π . Examples for such tasks include

- the secure evaluation of a function f , where protocol parties learn nothing but the result [GMW87],
- first-price sealed-bid auctions, where nothing but the winner and the highest bid is revealed or, for example,
- COVID contact tracing [Be21].

¹ Constreo Systems, dominik.doerner@student.kit.edu

² KASTEL, Karlsruhe Institute of Technology, jeremias.mechler@kit.edu

³ KASTEL, Karlsruhe Institute of Technology, joern.mueller-quade@kit.edu

More generally, MPC can be used for every efficiently computable task while guaranteeing properties such as *correctness*, *privacy* or the *independence of inputs*. These guarantees hold even if some parties are corrupted and deviate from the protocol in an arbitrary manner.

For MPC, there is a long list of feasibility results (e.g. [GMW87; Ya86]) that give a protocol for *any* efficiently computable function (*general MPC*). To this end, the function-to-be-evaluated f is transformed into a circuit C that is jointly evaluated. For *secret-sharing* MPC (e.g. [Da12; GMW87]), this evaluation happens online and gate by gate, incurring a round complexity that is linear in the circuit's depth. For high performance, low latency between the protocol parties is desirable.

For MPC with *garbled circuits*, the circuit C is “garbled” into a garbled circuit C' by a garbler. After an interaction phase for distributing the inputs (which is independent of the circuit depth), C' can be evaluated by the evaluator without further interaction. While garbled circuit protocols are generally constant-round, the garbled circuit C' is usually much larger than C , leading to a high communication overhead. In this setting, a high bandwidth is more important than a low latency.

Common to virtually all solutions for general MPC is the high computational or communication overhead. In a setting where the parties participating in such a protocol have very limited resources, e.g. on mobile devices, this overhead is prohibitive. As such, MPC has not seen wide-spread adoption, despite its promising security guarantees.

A partial solution to the limited efficiency is to outsource the most expensive parts of an MPC to one or more *servers*, e.g. in the cloud. If they can be partially trusted, for example in the sense that some kind of corruption threshold is tolerated by the protocol, security in such a setting can be shown. In practice, the assumption of e.g. a honest majority of the serves is often justified by placing the servers at different (competing) cloud providers. In such an outsourcing setting, the resources of the clients' devices are mostly irrelevant. Assuming the use of different cloud providers, it may be plausible to assume high bandwidths together with latency that is higher than e.g. within the same data center. In the following, we thus consider MPC based on garbled circuits, which is suitable for this particular setting.

Depending on the protocol, cloud-based MPC may not require the clients to communicate with each other. This may be highly desirable, e.g. for security reasons.

Possible applications include private set intersection, e.g. for contact discovery with messengers on smartphones or privacy-preserving analytics on e.g. smart-meter data. Generally, a setting with a complex function to be evaluated and small in- and outputs is favored.

1.1 Related Work

Garbling-based multi-party computation. Garbling schemes have proven to be a valuable tool in implementing MPC protocols. Goyal et al. [GMS08] designed an efficient MPC protocol using the cut-and-choose principle while Ben-David et al. [BNP08] designed a system called FairplayMP, in which the function to be executed is compiled to a boolean circuit and executed jointly and securely against semi-honest adversaries.

Server-aided MPC. More practice-oriented variations include the works of Alam et al. [Al18], for server-aided privacy-aware access control, and Baldimtsi et al. [Ba17], for online queries using data of offline users in the use case of social networks. Another subclass of the research has focused on improving security by utilizing multiple servers, as used by Jakobsen et al. [JNO14], Kerschbaum [Ke15] and Bugiel et al. [Bu11]. Thirdly, many MPC protocols utilize the cut-and-choose principle to ensure that the circuit is garbled correctly. Notable examples include the series of single-server-aided MPC protocols by Kamara et al. [KMR11] and the Salus system by Kamara et al. [KMR12].

MPC using secure hardware. Additional hardware assumptions provide a way to enable security under majority collusion. Some hardware assumptions, such as Trusted Execution Environments (TEE), provide great possibilities, but require the assumption of complete tamper-proof processors that can execute arbitrary functions. These have been used by Gupta [Gu16], Bahmani et al. [Ba16] and Benenson et al. [Be06]. Less restrictive assumptions include tamper-proof hardware tokens for cryptographic operations, as used by Katz [Ka07] or Dowsley et al. [DMN15].

Fortified Multi-Party Computation. In the established adaptive corruption model, an adversary may corrupt parties during any point of a protocol execution, being able to learn and modify a corrupted party's inputs and outputs. As such, even protocol parties that are initially honest are not afforded any kind of protection from the consequences of adaptive corruptions. This problem is addressed by Broadnax et al. [Br21], who introduce a new corruption model that distinguishes between *physical attacks* and *remote hacks*. By using simple remotely unhackable hardware modules like data diodes or air-gap switches, a protocol party can (temporarily) isolate itself, preventing an adversary to remotely hack it. By using an appropriate architecture, they construct a protocol where initially honest protocol parties are protected against the *consequences* of remote hacks: Unless all parties are corrupted or a party is corrupted at the very beginning of a protocol execution, the remote hacking of a party will *not* result in the adversary learning or being able to modify the hacked party's inputs or outputs. Being a theoretical feasibility result, their construction is not practically efficient.

1.2 Our Contribution

In previous solutions for server-aided MPC, the problem of server corruptions is usually addressed by defining a corruption threshold up to which security is guaranteed. However, no

technical measures are taken to justify such corruption thresholds. In practice, the problem is exacerbated by the fact that a server, presumably used for many MPC executions, may be an attractive target for hackers. It is thus very important to reduce the attack surface of such servers as much as possible.

Inspired by the work of Broadnax et al. [Br21], we investigate the use of simple remotely unhackable hardware modules like data diodes (which allow unidirectional communication only) or air-gap switches (which allow a party to temporarily disconnect itself) in the context of server-aided MPC in order to reduce the servers' attack surface. We present a server-aided general MPC protocol where the most expensive computations are outsourced. We split the servers into several parts with different responsibilities. By isolating parts of the servers from the network, they are not susceptible to remote hacks anymore. We prove the security of our construction in the Fortified UC framework [Br21].

To the best of our knowledge, we are the first to explore the use of isolation assumptions such as data diodes or air-gap switches in the field of server-aided MPC.

1.3 Guide for the Reader

In Sect. 2, we give a short introduction into the concepts used throughout the paper. In Sect. 3, we present our construction, along with a proof sketch together with a short theoretical evaluation of its performance. For more details, see the full version.

2 Preliminaries

In the following, we give a short and informal overview of important concepts and building blocks used in this paper. A detailed version can be found in in the full version.

Oblivious transfer (OT) [EGL85; Ra81] allows a receiver with an input $b \in \{0, 1\}$ to interact with a sender with inputs m_0 and m_1 . At the end of the interaction, the receiver learns the value m_b , but nothing about m_{1-b} . Conversely, the sender does not learn the bit b . This notion of 1-out-of-2-OT can be extended, e.g. to the case of n -out-of- k -OT.

When combined with an OT protocol, a *garbling scheme* [BHR12; Ya86] allows two parties P_1, P_2 to jointly evaluate a circuit C on their respective private inputs x_1 and x_2 . To this end, the *garbler* creates an obfuscated variant of C with its input x_1 hard-coded into the circuit. The *evaluator* P_2 can obtain “keys” for its input x_2 via OT from the garbler P_1 . Having these keys, it can evaluate C on inputs x_1 and x_2 to learn the result $y = C(x_1, x_2)$. With respect to security, it is guaranteed that P_1 learns nothing about x_2 and P_2 learns nothing about x_1 that it cannot compute from y and x_2 . Also, P_2 cannot evaluate the circuit repeatedly on different inputs.



Universal Composability [Ca01] is a security notion for MPC protocols in a setting where multiple (possibly adversarially chosen) protocols are executed concurrently. As an extension of the *real-ideal paradigm*, the security of a protocol is defined through an ideal functionality it (presumptively) realizes. *Fortified Universal Composability* (Fortified UC) [Br21] extends UC security to capture isolation properties of remotely unhackable hardware modules such as data diodes or air-gap switches and introduces a new corruption model that distinguishes between *physical attacks* and *remote hacks*.

3 Our Protocol

In the following, we will describe our MPC protocol, which is cast in the Fortified UC Framework [Br21]. We begin with a description of the protocol and its architecture, followed by the ideal functionality we realize and the formal protocol definition.

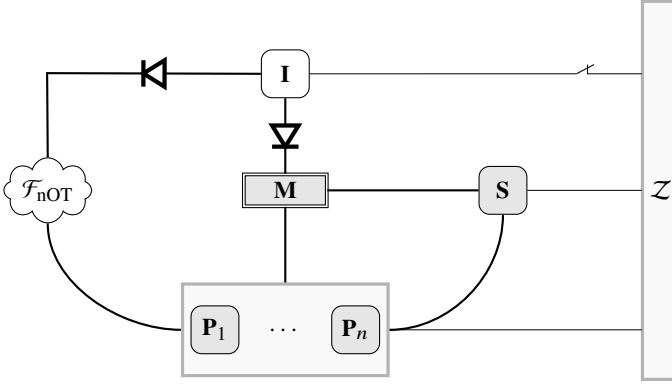
3.1 Protocol Architecture

Our protocol π_{MPC} allows a set of n players (\mathcal{P}), each with their own private input $x_i \in X$, to securely compute a globally known circuit $C : X^n \rightarrow Y^n$ and yet keep their inputs x_i and respective output $y_i \in Y$ private. Apart from the players, the parties include an initializer \mathbf{I} , a server \mathbf{S} , a hybrid functionality \mathcal{F}_{NOT} , a remotely unhackable module \mathbf{M} and the environment \mathcal{Z} .

The complete architecture is depicted in Fig. 1. Using the conventions for graphical depictions as defined by Broadnax et al. [Br21], rounded boxes represent main parties while sub-parties are represented by sharp corners. A cloud shape is meant to represent hybrid functionalities, a gray background that the party is online during the protocol execution and bold borders that a (sub-)party is unhackable. Air-gap switches, represented with , can be controlled by the party next to the hinge, allowing it to disconnect itself via this connection. Similarly, data diodes, represented with , allow unidirectional communication only.

The initializer \mathbf{I} is connected to the environment via an initially connected air-gap switch. To the other entities, it is connected via data diodes. After it has received initialization messages from the environment, the initializer can disconnect the air-gap switch. Then, it is no longer able to receive messages and thus cannot be hacked remotely. At the same time, it can still send protocol messages to the outside via its data diodes.

Connected to the initializer is a remotely unhackable module \mathbf{M} that will act as a surrogate for \mathbf{I} for any interaction with the other parties. As we will see later, this is necessary as this interaction cannot be performed by uni-directional messages from behind a data diode. Due to its simplicity, we assume that the module \mathbf{M} can be implemented in a way that makes it immune to hacks, e.g. as a fixed-function circuit that can be verified for correctness.

Fig. 1: Architecture of π_{MPC}

Finally, the server **S** interacts with the module **M** and the players \mathcal{P} . Due to the communication required in the protocol, we cannot protect it via data diodes or air-gap switches. Also, due to its complexity, we do not assume that it is remotely unhackable.

The protocol makes use of a single hybrid functionality \mathcal{F}_{nOT} that interacts with **I** and the players \mathcal{P} :

Definition 3.1 (Ideal Functionality for $n \times \text{OT}_2^1$). *The ideal functionality \mathcal{F}_{nOT} for non-adaptive n -fold 1-out-of-2 1-bit oblivious transfer ($n \times \text{OT}_2^1$) interacts with a sender **S**, a receiver **R** and an adversary \mathcal{A} .*

It is parameterized with the number of repetitions n as well as the word length l . These are fixed and known to all parties involved.

*On receiving $(\text{sid}, \text{initiate}, m = ((m_1^0, m_1^1), \dots, (m_n^0, m_n^1)))$ from **S**, with $m_i^v \in \{0, 1\}^l$:*

- *If no message $(\text{sid}, \text{initiate}, \cdot)$ has yet been received, store (m_1, \dots, m_n) .*
- *Send $(\text{sid}, \text{initiate})$ to \mathcal{A} .*

*On receiving $(\text{sid}, \text{request}, b = (b_1, \dots, b_n))$ from **R**, with $b_i \in [0, 1]$:*

- *Send $(\text{sid}, \text{output-request})$ to \mathcal{A} and wait for an answer $(\text{sid}, \text{output-response}, \phi)$ with $\phi \in \{\text{pass}, \text{abort}\}$.*
 - *if $\phi = \text{pass}$, continue.*
 - *if $\phi = \text{abort}$, abort.*
- *Check if a message $(\text{sid}, \text{request}, \cdot)$ was previously received and answered.*
 - *if not, continue.*
 - *if yes, abort.*
- *Check if a message $(\text{sid}, \text{initiate}, \cdot)$ was previously received.*
 - *if not, abort.*

- if yes, continue.
- Send $(sid, response, m^* = (m_1^{b_1}, \dots, m_n^{b_n}))$ to \mathbf{R} .

All main parties (namely \mathbf{I} , \mathbf{S} and \mathcal{P}) are connected to the environment \mathcal{Z} , although \mathbf{I} is connected only via an air-gap switch and will disconnect it upon first activation.

The only parties to receive any input from the environment are the players \mathcal{P} , each of which receives an input value $x_i \in X$.

As we will see later, the above architecture is suitable to provide security against adversaries that adhere to the following corruption model.

3.2 Corruption Model

We assume that the initializer \mathbf{I} can only be corrupted semi-honestly, meaning that an adversary learns all information, but does not deviate from the protocol. We make this assumption for the following reasons: i) We envision that \mathbf{I} , \mathbf{M} and \mathbf{S} are placed in the cloud. Cloud providers are often assumed to be honest-but-curious. This is captured by semi-honest corruption. ii) As \mathbf{I} is placed behind data diodes and an air-gap switch, the attack surface from the “outside” is greatly reduced, making remote hacks infeasible. Moreover, semi-honest corruption also captures e.g. side-channel attacks, which are still possible in this setting. They would only leak the internal state of the party and not allow for an active manipulation of the protocol flow.

Additionally, we assume that an adversary can only corrupt either \mathbf{I} or \mathbf{S} , but not both at the same time. This assumption is practically motivated: Standard security practices would dictate to use different providers to host both parties, such as two different major cloud providers.

The server \mathbf{S} as well as the players are subject to adaptive byzantine corruptions throughout the whole protocol execution.

3.3 Ideal Functionality

The ideal functionality \mathcal{F}_{MPC} for secure multi-party computation interacts with an initializer \mathbf{I} , a server \mathbf{S} , a given set of players $\mathcal{P} := \{\mathbf{P}_1, \dots, \mathbf{P}_n\}$ and an adversary \mathcal{A} .

It is parameterized with a circuit $C : (\{0, 1\}^l)^n \rightarrow (\{0, 1\}^l)^n$ to execute as well as the word length l . These are fixed and known to all parties involved.

Every player \mathbf{P}_i holds an input $x_i \in \{0, 1\}^l$.

Phase 1: Input

On receiving $(sid, input, x_i)$ from $\mathbf{P}_i \in \mathcal{P}$, with $x_i \in \{0, 1\}^l$:

- If no message $(\text{sid}, \text{input}, \cdot)$ has yet been received from \mathbf{P}_i , store x_i and ignore all further input messages from P_i .
- Send $(\text{sid}, \text{input-received}, \mathbf{P}_i)$ to \mathcal{A} .
- Once a message $(\text{sid}, \text{input}, \cdot)$ has been received from every player $\mathbf{P}_j \in \mathcal{P}$, mark this phase as complete.

Phase 2: Calculation

- Store $(y_1, \dots, y_n) := C(x_1, \dots, x_n)$ and mark this phase as complete.

Phase 3: Output

- For every player $\mathbf{P}_j \in \mathcal{P}$, send $(\text{sid}, \text{output-request}, \mathbf{P}_i)$ to \mathcal{A} and wait for an answer $(\text{sid}, \text{output-response}, \phi)$ with $\phi \in \{\text{pass}, \text{abort}\}$. If $\phi = \text{pass}$, send $(\text{sid}, \text{output}, y_i)$ to \mathbf{P}_i . If $\phi = \text{abort}$, send $(\text{sid}, \text{output}, \text{abort})$ to \mathbf{P}_i .

Corruption

- Upon corruption of a player \mathbf{P}_i , send its input x_i to the adversary (if existent) and let it replace it with an input x'_i .
- Upon corruption of the server \mathbf{S} , do nothing.
- Upon corruption of the initializer \mathbf{I} , do nothing.

3.4 Protocol

In the following, we present our construction.

Roughly, the protocol works as follows: First, the initializer \mathbf{I} creates a garbling of the circuit C and distributes the input labels to the players via the ideal functionality for oblivious transfer \mathcal{F}_{NOT} . The garbled circuit and the output labels are given to the module \mathbf{M} , which sends the garbled circuit to the server \mathbf{S} . According to their inputs, the players will obtain their input labels via oblivious transfer and forward them to the server. With these labels, \mathbf{S} can evaluate the garbled circuit, leading to an encrypted and authenticated result that it distributes to the parties. Using the output labels provided by \mathbf{M} , a player \mathbf{P}_i can reconstruct its output y_i .

Unless an adversary controls both the initializer and the server, this implies the following (informal) security guarantees:

- **Privacy:** For an honest player \mathbf{P}_i , the adversary is unable to learn its input or output. If the server is corrupted, the adversary only sees a player's input labels, but does not know the corresponding input. Conversely, the output is encrypted. If the initializer is corrupted, the adversary learns all input and output labels. However, as we assume that the server is honest in this case, it does not learn which labels are actually used.

- **Integrity:** The adversary cannot modify the computation’s result. To this end, it would need private information held by the initializer together with the ability to modify the server’s messages to the parties. As the adversary can only corrupt the server or the initializer, this is not possible.
- **Independence of Inputs:** In order to choose the input of a corrupted player \mathbf{P}_j in dependence of an honest player \mathbf{P}_i , the adversary would have to perform the OT of \mathbf{P}_j with choice bits depending on the choice bits of \mathbf{P}_i . By the definition of \mathcal{F}_{OT} , this is not possible. Alternatively, even if the initializer is corrupted, the adversary will not learn the choice bits of \mathbf{P}_i in the OT, making it impossible to use the corresponding input labels for \mathbf{P}_j (which are then known to the adversary for all possible inputs of \mathbf{P}_j).
- **No Repeated Circuit Evaluations:** In order to evaluate the circuit multiple times in the case of a corrupted server (and one or more corrupted players), the adversary needs to know additional input labels. Due to the security of the OT and the fact that the initializer must not be corrupted together with the server, this is not possible.

These only very informal security guarantees are implied by the ideal functionality \mathcal{F}_{MPC} , which is provably realized by our construction.

In the following, we give the formal protocol description.

Protocol 1 π_{MPC} , a secure server-aided multi-party computation protocol

Parties. Initializer \mathbf{I} , secure module \mathbf{M} , server \mathbf{S} and set of players $\mathcal{P} := \{\mathbf{P}_1, \dots, \mathbf{P}_n\}$.

Parameterization. The protocol is parameterized with a session identifier sid , a circuit $C : (\{0, 1\}^l)^n \rightarrow (\{0, 1\}^l)^n$, a security parameter κ , an input length l and a projective, output-projective and output-key-secure garbling scheme⁴ $\mathcal{G} = (\text{Gb}_{\mathcal{G}}, \text{En}_{\mathcal{G}}, \text{Ev}_{\mathcal{G}}, \text{De}_{\mathcal{G}})$. In the following, we use \vec{x} to abbreviate x_1, \dots, x_n .

Inputs. Every $\mathbf{P}_i \in \mathcal{P}$ holds an input $x_i \in \{0, 1\}^l$. The other parties hold no initial input.

Hybrid functionality. The protocol makes use of the ideal functionality \mathcal{F}_{OT} for n -fold oblivious transfer (see Definition 3.1).

Channels. \mathbf{I} , \mathbf{S} and all $\mathbf{P}_i \in \mathcal{P}$ are connected to \mathcal{Z} via air-gap switches that are initially connected. \mathbf{I} is connected to \mathbf{M} via a data diode. Every player $\mathbf{P}_i \in \mathcal{P}$ is connected via a (secure) standard channel to both \mathbf{M} and \mathbf{S} .

⁴ A garbling scheme \mathcal{G} is called *projective* if the encoding function consists of $2nm$ input keys (also called “wire labels”), where n is the number of input values and m is the number of bits in each input value.

We call a garbling scheme *output-projective* if the decoding information consists of 2 wire labels for each output bit in each output value, one corresponding to each possible value of that bit.

Informally, output-key secure (OKS) garbling schemes, introduced by Jafargholi et al. [JSW17], maintain their security properties even if the adversary is provided with an unordered set of output keys, i.e. the possible labels that can be produced by $\text{Ev}_{\mathcal{G}}(\cdot)$ for each output bit, but without knowledge which bit value they each encode.

Protocol Phase 1: Garbling

- **MGarble**(C), executed by **I** on first activation
 1. Close the air-gap switch to \mathcal{Z} .
 2. Set $(\tilde{C}, e, d) \leftarrow \text{Gb}_{\mathcal{G}}(1^\kappa, C)$.
 3. Parse $e =: \{L_{in,i,j}^v\}_{i \in [n], j \in [l], v \in \{0,1\}}$ and $d =: \{L_{out,i,j}^v\}_{i \in [n], j \in [l], v \in \{0,1\}}$.
 4. For each player $\mathbf{P}_i \in \mathcal{P}$:
 - a) Initialize a new instance of \mathcal{F}_{NOT} with session identifier $\text{sid}_{ot,i}$.
 - b) Send $(\text{sid}_{ot,i}, \text{initiate}, L_{in,i} = ((L_{in,i,1}^0, L_{in,i,1}^1), \dots, (L_{in,i,l}^0, L_{in,i,l}^1)))$ to \mathcal{F}_{NOT} .
 5. Send $(\text{init}, \tilde{C}, \vec{\text{sid}}_{ot}, \vec{L}_{out})$ to **M**.

Protocol Phase 2: Distribution

- **MDistReq**($\tilde{C}, \vec{\text{sid}}_{ot}, \vec{L}_{out}$), executed by **M** after receiving a message $(\text{init}, \tilde{C}, \vec{\text{sid}}_{ot}, \vec{L}_{out})$ from **I**
 1. Send $(\text{circuit}, \tilde{C})$ to **S**.
 2. Save \vec{L}_{out} .
 3. For each player $\mathbf{P}_i \in \mathcal{P}$, send $(\text{labels}, \text{sid}_{ot,i})$ to \mathbf{P}_i .
- **MDistServer**(\tilde{C}), executed by **S** after receiving a message $(\text{circuit}, \tilde{C})$ from **M**
 1. Save \tilde{C} .
- **MDistPlayer**($x_i, \text{sid}_{ot,i}$), executed by \mathbf{P}_i after receiving a message $(\text{labels}, \text{sid}_{ot,i})$ from **M** and an input (x_i) from \mathcal{Z}
 1. Send $(\text{sid}_{ot,i}, \text{request}, x_i)$ to \mathcal{F}_{NOT} and receive \tilde{x}_i .
 2. Send $(\text{input}, \tilde{x}_i)$ to **S**.

Protocol Phase 3: Computation

- **MCalcReq**($\mathbf{P}_i, \tilde{x}_i$), executed by **S** after receiving a message $(\text{input}, \tilde{x}_i)$ from \mathbf{P}_i .
 1. Save $(\mathbf{P}_i, \tilde{x}_i)$.
 2. If an entry has been saved for every $\mathbf{P}_j \in \mathcal{P}$:
 - a) Set $\vec{y} = \text{Ev}_{\mathcal{G}}(\tilde{C}, \vec{x} = (\tilde{x}_1, \dots, \tilde{x}_{|\mathcal{P}|}))$.
 - b) If the execution of $\text{Ev}_{\mathcal{G}}$ aborts, send $(\text{output}, \text{abort})$ to every player $\mathbf{P}_i \in \mathcal{P}$.
 - c) Else, send $(\text{output}, \tilde{y}_i)$ to every player $\mathbf{P}_i \in \mathcal{P}$.
- **MCalcRes**(\tilde{y}_i), executed by \mathbf{P}_i after receiving a message $(\text{output}, \tilde{y}_i)$ from **S**
 1. Save \tilde{y}_i .
 2. Send (decode-request) to **M**.

Protocol Phase 4: Output Decoding

- **MDecSend**($\vec{L}_{out}, \mathbf{P}_i$), executed by **M** after receiving a message (decode-request) from $\mathbf{P}_i \in \mathcal{P}$.
 1. Save $(\mathbf{P}_i, \text{decode})$.
 2. If an entry has been saved for every player $\mathbf{P}_j \in \mathcal{P}$:

- a) Send (decode-response, $L_{out,i}$) to every player $\mathbf{P}_i \in \mathcal{P}$.
- **MDecEx**($\tilde{y}_i, L_{out,i}$), executed by \mathbf{P}_i after receiving a message (decode-response, $L_{out,i}$) from \mathbf{M} .
 1. Set $y_i := \text{De}_{\mathcal{G}}(L_{out,i}, i, \tilde{y}_i)$.
 2. Return y_i .

We can now state our main theorem:

Theorem 3.1. *The protocol π_{MPC} Fortified-UC-realizes the ideal functionality \mathcal{F}_{MPC} in the \mathcal{F}_{NOT} -hybrid model for adversaries adhering to the corruption model of Sect. 3.2.*

3.5 Proof Sketch

In the following, we will give a short proof sketch for Theorem 3.1. The complete proof can be found in the full version.

In order to show Theorem 3.1, we have to show the indistinguishability of a real execution of π_{MPC} and the execution of \mathcal{F}_{MPC} with a simulator that “simulates” the execution of π_{MPC} . To this end, the simulator must provide an *interactive distinguisher* (the environment) with protocol messages that are indistinguishable from messages of a real execution. The environment may influence the execution, e.g. by (adaptively) giving inputs to parties, learning the outputs of honest parties and by communicating with the adversary.

The architecture has been specifically designed to allow simulation:

If the initializer is corrupted, it must follow the protocol flow and expose its secrets to the simulator by passing them to \mathbf{M} (which is simulated by the simulator). Since \mathbf{I} and \mathbf{S} cannot be corrupted simultaneously according to our corruption model, the adversary cannot decrypt (fake) values sent by the simulator or check their validity. With at least one of both being controlled by the simulator, the environment has not enough information to verify the circuit execution.

If any player is corrupted, they need to garble their inputs before passing them to the server \mathbf{S} . As such, the simulator can learn their input from \mathcal{F}_{NOT} , as players would only be able to garble their inputs themselves if \mathbf{I} is corrupted as well. Following the reasoning above, the simulator would learn the input labels from \mathbf{M} and can then decrypt the input given to the simulated \mathbf{S} .

If the simulator has committed itself to encrypted random inputs for honest players, which the environment then learns by adaptively corrupting \mathbf{S} , then the simulator is still able to decrypt the encrypted circuit output to the correct output values by manipulating the output labels.

We now state the simulator for one considered corruption pattern. The other corruption cases are handled similarly. For the sake of simplicity, we consider static corruptions only.

Definition 3.2 (Simulator for corrupted players and a corrupted server). *The simulator \mathcal{S} works as follows:*

1. Set $(\tilde{C}, \vec{L}_{in}, \vec{L}_{out}) \leftarrow \text{Gb}_{\mathcal{G}}(1^\kappa, C)$.
2. Send the message $(\text{circuit}, \tilde{C})$ to \mathbf{S}^* .
3. For each corrupted $\mathbf{P}_i^* \in \mathcal{P}^*$:
 - a) Create a new dummy instance of \mathcal{F}_{NOT} with session identifier $\text{sid}_{ot,i}$.
 - b) Send the message $(\text{labels}, \text{sid}_{ot,i})$ to \mathbf{P}_i^* .
4. When receiving a message $(\text{sid}, \text{input-received}, \mathbf{P}_i)$ from \mathcal{F}_{MPC} for an honest \mathbf{P}_i :
 - a) Choose \tilde{x}_i at random.
 - b) Set $\tilde{x}_i \leftarrow \text{En}_{\mathcal{G}}(L_{in,i}, i, x_i)$.
 - c) Send the message $(\text{input}, \tilde{x}_i)$ to \mathbf{S}^* .
5. When a corrupted \mathbf{P}_i^* sends a message $(\text{sid}_{ot,i}, \text{request}, x_i)$ meant for \mathcal{F}_{NOT} :
 - a) Set $\tilde{x}_i \leftarrow \text{En}_{\mathcal{G}}(L_{in,i}, i, x_i)$.
 - b) Have \mathcal{F}_{NOT} return $(\text{sid}_{ot,i}, \text{response}, \tilde{x}_i)$ to \mathbf{P}_i^* .
6. When receiving the first message (output, \cdot) from \mathbf{S}^* meant for an honest player
 - a) For every corrupted player $\mathbf{P}_i^* \in \mathcal{P}^*$
 - i. Let x_i be the input that \mathbf{P}_i^* has sent to \mathcal{F}_{NOT} . Set it randomly ($x_i \stackrel{R}{\leftarrow} X$) if \mathbf{P}_i^* has not interacted with \mathcal{F}_{NOT} .
 - ii. Send the message $(\text{sid}, \text{input}, x_i)$ as \mathbf{P}_i^* to \mathcal{F}_{MPC} .
7. When \mathcal{F}_{MPC} sends a message $(\text{sid}, \text{output-request}, \mathbf{P}_i)$ to an honest player $\mathbf{P}_i \notin \mathcal{P}^*$, delay sending a response until later on. Let all output-requests to corrupted players pass.
8. When \mathbf{S}^* sends a message (output, ϕ) to an honest player \mathbf{P}_i
 - a) If $\phi = \text{abort}$, send $(\text{sid}, \text{output-response}, \text{abort})$ to \mathcal{F}_{MPC} .
 - b) If ϕ is an encoded value $\tilde{y}_i^{\text{server}}$, mark that a decode-request-message was received from \mathbf{P}_i .
9. When a decode-request-message was received from every honest and corrupted player in \mathcal{P}
 - a) Evaluate $y^{\text{sim}} = C(x)$, with x being the input of the corrupted players as it was given to \mathcal{F}_{MPC} and the input of the honest players as it was given (in an encoded form) to \mathbf{S}^* .
 - b) For every corrupted player \mathbf{P}_i^*
 - i. Wait for a message $(\text{sid}, \text{output}, y_i^{\text{ideal}})$ from \mathcal{F}_{MPC} meant for \mathbf{P}_i^* .
 - ii. Create a copy $\vec{L}_{out,i}$ of $L_{out,i}$ as follows:
 - For each $j \in [l]$
 - If $y_{i,j}^{\text{sim}} = y_{i,j}^{\text{ideal}}$, set $\vec{L}_{out,i,j}^v := L_{out,i,j}^v, v \in [0, 1]$.
 - Else, set $\vec{L}_{out,i,j}^v := L_{out,i,j}^{1-v}, v \in [0, 1]$.
 - iii. Send $(\text{decode-response}, \vec{L}_{out,i})$ to \mathbf{P}_i^* .
 - c) For every honest player \mathbf{P}_i
 - If $\tilde{y}_i^{\text{server}}$ can be decoded successfully, send the response $(\text{sid}, \text{output-response}, \text{pass})$ to \mathcal{F}_{MPC} to let the output pass to (this single) \mathbf{P}_i .
 - Else, send the response $(\text{sid}, \text{output-response}, \text{abort})$ to \mathcal{F}_{MPC} .

We will now briefly argue why the environment's view is computationally indistinguishable between an execution of π_{MPC} and a real-world adversary and the execution of \mathcal{F}_{MPC} and the simulator of Definition 3.2 for the case of corrupted players and a corrupted server.

To this end, we define a series of games, starting with the real execution. Finally, we reach the ideal execution through indistinguishable changes.

Game 1 We now consider an execution with an ideal functionality $\langle \mathcal{F}_{\text{MPC}} \rangle$ that is identical to \mathcal{F}_{MPC} , but also tells the simulator all inputs and lets it determine all outputs. In this execution, the simulator performs the protocol for the honest parties, using the inputs provided by the functionality and making outputs through it. Also, \mathcal{F}_{NOT} is simulated honestly. As the changes are only syntactical and oblivious for the environment, it is easy to see that the environment's view is identically distributed.

Game 2 In Game 2, the simulator initializes an internal vector $\$INPUT$ that saves the inputs x_i of all players. Initially set all inputs to random values. If a corrupted player \mathbf{P}_i sends an input to \mathcal{F}_{NOT} , replace the saved input with the given value.

When \mathbf{S}^* first sends out a output-message to any (honest or corrupted) player, send $(\text{sid}, \text{input}, x_i)$ to $\langle \mathcal{F}_{\text{MPC}} \rangle$ for all players with $x_i = \$INPUT[i]$ for every \mathbf{P}_i .

Replace the output labels $L_{out,i}$ in the message (decode-response, $L_{out,i}$) with a copy $\bar{L}_{out,i}$, where the output labels for the same bit ($L_{out,i,j}^0, L_{out,i,j}^1$) are conditionally switched so that the garbled output produced by $\text{Ev}_{\mathcal{G}}(\tilde{C}, \text{En}_{\mathcal{G}}(L_{in}, \$INPUT))$ decodes to the output returned by $\langle \mathcal{F}_{\text{MPC}} \rangle$.

It is obvious that when $\$INPUT$ holds the correct server input of every player it follows that the result produced by the ideal functionality is identical to the one to be expected from the server and thus no changes to the output labels will be made. Note that the actual behavior of the corrupted server has no effect on this change. Thus, the games are indistinguishable.

Game 3 Let us assume that the simulator does not know all inputs. We define a changed game where an honest player \mathbf{P}_i is chosen at random by the simulator. \mathbf{P}_i is mostly simulated honestly, but will choose a (random) input $\hat{x}_i \neq x_i$ such that the resulting $\hat{y}_i := \text{Ev}_{\mathcal{G}}(\hat{x}_i, \cdot) \neq \text{Ev}_{\mathcal{G}}(x_i, \cdot)$. \hat{x}_i is given to \mathcal{F}_{NOT} and (in an encoded form) to \mathbf{S}^* and thus saved in $\$INPUT$. The original input x_i is still saved and sent in a message $(\text{sid}, \text{input}, x_i)$ to $\langle \mathcal{F}_{\text{MPC}} \rangle$. This results in at least one row of $L_{out,i}$ to be rearranged due to the simulator instructions as to preserve the correctness of the player outputs.

If a player in Game 3 were to send different inputs to the hybrid functionality \mathcal{F}_{NOT} and the ideal functionality $\langle \mathcal{F}_{\text{MPC}} \rangle$, then the resulting game is still indistinguishable from Game 2 under the output-key secure obliviousness of \mathcal{G} .

Let us assume that there exists an environment \mathcal{Z} that can distinguish between Games 2 and 3, thereby determining for at least one tuple $(L_{out,i,j}^v, i, j, w)$ that $v \neq w$. Such an environment \mathcal{Z} can be used to construct a new adversary \mathcal{B} that can break the output-key secure obliviousness of \mathcal{G} . Due to the security of \mathcal{G} , we thus conclude that Game 2 and Game 3 are indistinguishable.

Game 4 We now replace all (simulated) honest players in $\mathcal{P} \setminus \mathcal{P}^*$ with their ideal world counterparts as in Definition 3.2.

When $\langle \mathcal{F}_{\text{MPC}} \rangle$ sends an input-received-message to the simulator informing about the input of an honest player $\mathbf{P}_i \notin \mathcal{P}^*$, send the message $(\text{input}, \tilde{x}_i)$ to \mathbf{S}^* , where \tilde{x}_i is the encoding of a randomly chosen input x_i .

When \mathbf{S}^* sends an abort to an honest \mathbf{P}_i , respond with abort to any output-request by $\langle \mathcal{F}_{\text{MPC}} \rangle$ for \mathbf{P}_i . If \mathbf{S}^* instead sends an output \tilde{y}_i^* to \mathbf{P}_i , mark that a decode-request-message from \mathbf{P}_i was received.

After sending out decode-response-messages, try to decode the result \tilde{y}_i^* received from the server for any honest \mathbf{P}_i and respond with pass to any output-request by $\langle \mathcal{F}_{\text{MPC}} \rangle$ for \mathbf{P}_i if that is the case and abort otherwise.

Game 4 is indistinguishable from Game 3 for any environment \mathcal{Z} under the output-key secure obliviousness, authenticity and the output-projectiveness of \mathcal{G} .

Game 5 Replace the ideal functionality $\langle \mathcal{F}_{\text{MPC}} \rangle$ with the ideal functionality \mathcal{F}_{MPC} . Game 5 is perfectly indistinguishable from Game 4 for any environment \mathcal{Z} .

We remark that \mathcal{S} in the latest adaptation is no longer dependent on the input or output of any party. Since these values are not used in the simulator's instructions and both ideal functionalities are identical apart from that, this change is naturally perfectly indistinguishable.

Note that the protocol described in Game 5 is identical to our simulator instructions. The indistinguishability of the presented games proves our claim.

For the complete proof, see the full version.

3.6 Efficiency

Regarding the efficiency of the protocol we have multiple aspects of complexity that we can differentiate. In the following, we will present a summary of our findings regarding the communication cost, total information flow and computational complexity of the protocol. The full efficiency analysis can be found in the full version.

The communication cost of the protocol, with which we denote the total number of messages sent between protocol participants, with n being the number of players, is $2 + 5n$ for the execution of π_{MPC} . As such, the communication cost is linear in the number of players.

The information flow is our designation for the total number of bits transferred between parties and is more difficult to determine than the communication cost as it depends on details of the garbling scheme and the circuit to be evaluated. Using variables for the number of input bits per player l , the maximum size of input and output keys m and the information size of the garbled circuit \tilde{c} , we can determine an upper bound of $\Theta(\tilde{c} + n \cdot (\log(n) + ml))$ bit for the execution of π_{MPC} alone, which is quasi-linear in the number of players n .

The computation complexity heavily depends not only on the implementation of the protocol and its schemes but also the size of the used circuit and the underlying hardware. We note that only three sub-tasks incur noteworthy computation efforts, namely the garbling of the circuit, the execution of \mathcal{F}_{NOT} and the evaluation of the garbled circuit. It can be seen that our protocol has the advantage that all players are burdened identically and have outsourced both the garbling and execution to other participants, namely **S** and **I**. Furthermore, even **I** can realistically be presented by low-performance hardware, since the initial garbling can be pre-processed and stored securely before initiation of the protocol with concrete players if the circuit is known beforehand.

4 Conclusion

While many solutions for server-aided MPC exist, they do not aim to protect the servers from remote hacks performed by an adversary.

In this paper, we take a first step towards addressing this problem: We have presented a composable general MPC protocol in the setting of server-aided MPC. By modularizing the servers and using remotely unhackable hardware modules such as data diodes and air-gap switches, we can greatly reduce the attack surface.

The resulting protocol is plausibly efficient and provides security going beyond the state of the art.

Acknowledgements

We would like to thank Lukas Beeck, Markus Raiber and Rebecca Schwerdt for helpful discussions on a related but unpublished project.

Jeremias Mechler, Jörn Müller-Quade: This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

References

- [Al18] Alam, M.; Emmanuel, N.; Khan, T.; Khan, A.; Javaid, N.; Choo, K. K. R.; Buyya, R.: Secure policy execution using reusable garbled circuit in the cloud. *Futur. Gener. Comput. Syst.* 87/, pp. 488–501, 2018.
- [Ba16] Bahmani, R.; Barbosa, M.; Brassier, F.; Portela, B.; Sadeghi, A.-r.: Secure Multiparty Computation from SGX, tech. rep., Cryptology ePrint Archive, 2016.
- [Ba17] Baldimtsi, F.; Papadopoulos, D.; Papadopoulos, S.; Scafuro, A.; Triandopoulos, N.: Server-Aided Secure Computation with Off-line Parties. *ESORICS Part 1/LNCS 10492*, pp. 103–123, 2017, arXiv: 9780201398298.
- [Be06] Benenson, Z.; Fort, M.; Freiling, F.; Kesdogan, D.; Penso, L. D.: TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. *Eur. Symp. Res. Comput. Secur./*, pp. 34–48, 2006.
- [Be21] Beskorovajnov, W.; Dörre, F.; Hartung, G.; Koch, A.; Müller-Quade, J.; Strufe, T.: ConTra Corona: Contact Tracing against the Coronavirus by Bridging the Centralized-Decentralized Divide for Stronger Privacy. In (Tibouchi, M.; Wang, H., eds.): *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security*, Singapore, December 6-10, 2021, Proceedings, Part II. Vol. 13091. *Lecture Notes in Computer Science*, Springer, pp. 665–695, 2021.
- [BHR12] Bellare, M.; Hoang, V. T.; Rogaway, P.: Foundations of garbled circuits. *Proc. ACM Conf. Comput. Commun. Secur./*, pp. 784–796, 2012.
- [BNP08] Ben-David, A.; Nisan, N.; Pinkas, B.: FairplayMP - A system for secure multiparty computation. *Proc. ACM Conf. Comput. Commun. Secur./*, pp. 257–266, 2008.
- [Br21] Broadnax, B.; Koch, A.; Mechler, J.; Müller, T.; Müller-Quade, J.; Nagel, M.: Fortified Multi-Party Computation: Taking Advantage of Simple Secure Hardware Modules. *Proc. Priv. Enhancing Technol.* 2021/4, pp. 312–338, 2021.
- [Bu11] Bugiel, S.; Stefan, N.; Sadeghi, A.-r.; Schneider, T.: Twin Clouds : Secure Cloud Computing with Low Latency./, pp. 32–44, 2011.
- [Ca01] Canetti, R.: Universally Composable Security: a New Paradigm for Cryptographic Protocols. In: *42nd Found. Comput. Sci. Conf.* Vol. 16, 2001.
- [Da12] Damgård, I.; Pastro, V.; Smart, N. P.; Zakarias, S.: Multiparty Computation from Somewhat Homomorphic Encryption. In (Safavi-Naini, R.; Canetti, R., eds.): *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference*, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Vol. 7417. *Lecture Notes in Computer Science*, Springer, pp. 643–662, 2012.

- [DMN15] Dowsley, R.; Müller-Quade, J.; Nilges, T.: Weakening the Isolation Assumption of Tamper-proof Hardware Tokens./1, 2015, arXiv: 1502.03487.
- [EGL85] Even, S.; Goldreich, O.; Lempel, A.: A Randomized Protocol for Signing Contracts. *Commun. ACM* 28/6, pp. 637–647, 1985, arXiv: arXiv:1011.1669v3.
- [GMS08] Goyal, V.; Mohassel, P.; Smith, A.: Efficient two party and multi party computation against covert adversaries. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 4965 LNCS/, pp. 289–306, 2008.
- [GMW87] Goldreich, O.; Micali, S.; Wigderson, A.: How To Play Any Mental Game OR A Completeness Theorem for Protocols with Honest Majority. In: *Proc. ACM STOC*. Pp. 218–229, 1987, ISBN: 0897912217.
- [Gu16] Gupta, D.: *Practical and Deployable Secure Multi-Party Computation*, Yale University, 2016.
- [JNO14] Jakobsen, T. P.; Nielsen, J. B.; Orlandi, C.: A Framework for Outsourcing of Secure Computation. *Proc. 6th Ed. ACM Workshop Cloud Comput. Secur. CCSW '14/*, pp. 81–92, 2014.
- [JSW17] Jafargholi, Z.; Scafuro, A.; Wichs, D.: Adaptively indistinguishable garbled circuits. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 10678 LNCS/, pp. 40–71, 2017.
- [Ka07] Katz, J.: Universally composable multi-party computation using tamper-proof hardware. *EUROCRYPT 2007* 4515/, pp. 115–128, 2007.
- [Ke15] Kerschbaum, F.: Oblivious outsourcing of garbled circuit generation. *Proc. ACM Symp. Appl. Comput. 13-17-April/*, pp. 2134–2140, 2015.
- [KMR11] Kamara, S.; Mohassel, P.; Raykova, M.: *Outsourcing Multi-Party Computation./*, 2011.
- [KMR12] Kamara, S.; Mohassel, P.; Riva, B.: *Salus : A System for Server-Aided Secure Function Evaluation./*, 2012.
- [Ra81] Rabin, M. O.: *How To Exchange Secrets with Oblivious Transfer*, 1981.
- [Ya86] Yao, A. C. C.: *How To Generate and Exchange Secrets. Annu. Symp. Found. Comput. Sci./1*, pp. 162–167, 1986.