cryptography

MDPI

Article

Foundations of Programmable Secure Computation

Sven Laur [1,†] and Pille Pullonen-Raudvere [2,*,†]

[1] Institute of Computer Science, University of Tartu, Narva mnt 18, 51009 Tartu, Estonia; sven.laur@ut.ee
[2] Cybernetica AS, Mäealuse 2/1, 12618 Tallinn, Estonia
* Correspondence: pille.pullonen-raudvere@cyber.ee
† Both authors contributed equally to this work.

Abstract: This paper formalises the security of programmable secure computation focusing on simplifying security proofs of new algorithms for existing computation frameworks. Security of the frameworks is usually well established but the security proofs of the algorithms are often more intuitive than rigorous. This work specifies a transformation from the usual hybrid execution model to an abstract model that is closer to the intuition. We establish various preconditions that are satisfied by natural secure computation frameworks and protocols, thus showing that mostly the intuitive proofs suffice. More elaborate protocols might still need additional proof details.

Keywords: secure multiparty computation; security proofs; universal composability; reactive simulatability; secret sharing; provable security

Citation: Laur, S.; Pullonen-Raudvere, P. Foundations of Programmable Secure Computation. *Cryptography* **2021**, *5*, 22. https://doi.org/10.3390/cryptography5030022

Academic Editor: Josef Pieprzyk

Received: 30 April 2021
Accepted: 18 August 2021
Published: 21 August 2021

1. Introduction

Over the years of secure multiparty computation (MPC) research many different frameworks [1–7] and applications [8–13] have been developed. Among the applications, some are tailored for a specific framework, others are more general and simply assume some underlying computation capabilities. Especially, when developing a new application or algorithm for MPC, it would be best if we had standard notions to use to specify the requirements (types of functionalities, data types and security assumptions) that the algorithm has on the MPC frameworks. This would give basis for the applicability of the algorithm as well as the security proof of the algorithm.

The security proofs and claims of the programmable MPC frameworks are usually well documented and follow the best practices of universally composable security [14]. Therefore, we are given guarantees that everything from protocol inputs until protocol outputs remains secure independently of the context where the protocol is being used. However, the standard set of operations in MPC frameworks is quite small, essentially supporting linear combinations, multiplication, giving inputs and getting outputs. In addition, these frameworks are often specified as one monolithic secure functionality, for example, arithmetic black box (ABB) [15]. ABB is essentially a representation of a secure computer where you can put values and give computation commands.

It is a separate task to build all other necessary algorithms and building blocks in order to achieve bigger applications like secure machine learning. Building full-fledged applications, like the equality check in Algorithm 1, that do not release intermediate values, is straightforward to model in ABB. In this case, you can give this code as commands to the ABB that would give out the desired outcome. However, note that instead of the shared values, the inputs would be private inputs of the participants. If the ABB is secure, then the output z is computed securely and, for example, if the ABB operates on finite fields, then this protocol is also correct. Formally, there is no good way to add primitive operations inside the ABB as there is no access to the internal representation of the intermediate data. Whenever a new operation is added, we should formally prove the security of the whole ABB. Still, ABB is the best abstraction to define quite generic new primitives for secure computation, for real-world uses see [16,17].

Many algorithms can be sped up by releasing intermediate values. For example, consider the sorting algorithm in Algorithm 2 where the comparison result b is published in the middle of the algorithm and elements are ordered based on this. The value b can be seen as given out from the ABB and then a new command can be given to reorder the values as necessary. In addition, there is no way in ABB to actually return the intermediate representation of the secure values $[\![k]\!]$ and $[\![m]\!]$. Therefore, the effect of such a protocol is such that there are intermediate representations of m and k inside the ABB but the order and its use is defined by the follow up commands sent to the ABB. In this case, the security of the ABB is not necessarily sufficient to give security guarantees. Additional reasoning must be carried out, as the published values and actions based on them happen outside of the ABB.

Algorithm 1 Equality Check

Input: Two shared values $[\![x]\!]$ and $[\![y]\!]$
Output: 0 if $x = y$, non-zero value otherwise
Generate a shared random non-zero value $[\![r]\!]$.
Compute $[\![z]\!] = ([\![x]\!] - [\![y]\!]) \cdot [\![r]\!]$.
Publish $[\![z]\!]$ as z.
return z

Our view of secure computation adds an explicit way to consider such choices and published values as well as to consider each operation such as comparison or addition individually, not just as one functionality. In short, our goal is to define an abstract execution environment for secure protocols where the only details that are relevant for the security analysis are necessary. These details are those that can be easily seen from algorithms written down in pseudocode like our two examples. One considers secure values $[\![x]\!]$, different computations can be carried out with them and a special focus is on the published values x. For example, if some comparison-based sorting algorithm is defined similarly to Algorithm 2, then special care should be taken to analyse what the published values can leak. For example, they may leak something about the number of equal values in the input.

Algorithm 2 Two-element Comparison-Based Sorting

Input: Two shared values $[\![x]\!]$ and $[\![y]\!]$
Output: Return fresh shares of x and y so that larger is the first
Shuffle $[\![x]\!]$ and $[\![y]\!]$ to learn $[\![k]\!]$, $[\![m]\!]$ where $\{m, k\} = \{x, y\}$.
Compute private comparison $[\![b]\!] = [\![k]\!] \geq [\![m]\!]$, where $b = 1$ if $k \geq m$.
Publish $[\![b]\!]$ as b.
return $[\![k]\!], [\![m]\!]$ *if* $b = 1$ *else* $[\![m]\!], [\![k]\!]$

From the viewpoint of building secure computation algorithms, it is easier to think of secure functionalities for individual protocols, like addition, multiplication, comparison, equality checks or bit decomposition. Essentially, if we had such small secure protocols then any algorithm described as using them could immediately be implemented in any concrete instantiations of these protocols while the composition theorem guarantees the security of the algorithm. Therefore, we have a conflict of interest between the frameworks that are specified as secure computational units versus the algorithm development that benefits from considering primitive operations of the computational unit individually. Either the security proofs of the concrete algorithms are very generic and refer more to intuition (e.g., that only published values should be analysed) or the algorithm is proven secure with respect to some fixed MPC framework, e.g., [18–20]. In the first case, we lose the rigour and good security definitions given by detailed security proofs. In the second, we are not exploring the full setting where this algorithm could be applicable and the proofs should be done again when implemented with different basic primitives and protection schemes.

Studying the separate arithmetic protocols as individual secure components is fairly straightforward for the cases of passive security that operate without private setup pa-

rameters. For example, earlier protocol development [21–23] focused their proofs on a fixed representation of the secure values and simply stated that the protocols are more generally applicable in practice. However, for frameworks using private setup parameters like shared keys in their operation, the monolithic functionality is a natural and nicer choice. Essentially, the monolithic functionality allows hiding the setup inside the protocol and using common flavours of composable security. If we would like to consider monolithic functionalities by their components, then we would need to take the joint state of the components into account. This could be achieved, for example, by using the joint-state UC framework [24] for the security proofs.

If we use a secure computation functionality as a starting point for defining a new algorithm, then we can base the algorithm on the ideal functionality of the framework. Therefore, the proof of the algorithm is in a hybrid model assuming interactions with the specification of the underlying computation functionality. The resulting hybrid model is usually still more complex than desired. A malicious adversary could possibly change the scheduling of subprotocols, create unplanned subprotocol instances, alter intermediate values or cause significant local computations. Most proofs first analyse the security in the abstract setting where shares are treated as non-malleable secure storage and focus on analysing only the values that are explicitly revealed.

We rigorously formalise the abstract execution model and study under which conditions the abstract and the full hybrid model are equivalent. We first establish the foundations of specifying secure computation environments and the security of both their computation protocols and secure storage in Section 2. We call the combination of the storage and the protocols the secure protection domain. Second, we study how to extend the protection domain with a new protocol. Third, we show how such security proofs can be done in an abstract setting when making some natural assumptions about the protection domain and the protocol. In doing this, we formalise the intuition that in most algorithms only the values that are public or malleable need proper discussion in a security proof. The abstract model and all relevant conditions are derived in Section 3. We specify the abstract model in a sequence of steps that each simplify some aspect of the hybrid model. In Section 4, we summarise the abstract execution model as well as the conditions under which it can be used. In addition, we explore why these conditions are satisfied by most programmable secure computation frameworks and primitive protocols. For protection domains, we specify the properties that have to be met in order for the storage to be secure and flexible enough to allow secure computation. In addition, we define a canonical form for reasonable functionalities defining the primitive operations for secure computation. Essentially, we assume that all functionalities are such that their output depends on the input values and not on the format of the protection. The storage allows for some homomorphic modifications but does not reveal information about the stored values. We also study the properties of the secure computation protocols that can realise these functionalities. Overall, we note that as long as some parties in the computation remain honest, they should also have control over which computations can be executed with the private values. Therefore, the adversarial actions are quite limited as long as the protocols are able to deal with malformed inputs and have reasonable semantics.

2. Materials and Methods

Modelling MPC protocols as asynchronous distributed system requires many low-level details that cannot be neglected in the definitions and proofs. We define a visual representation for the reactive simulatability framework (RSIM) [25–27] to visualise the main insight and sketch how the arguments can be fleshed out to complete proofs. RSIM is one formalisation for composable security, thus showing security according to their definitions guarantees security in overall contexts where the protocol might be used.

In this section, we describe the RSIM framework and our visual notation for it. Second, we discuss general privacy definitions based on observational equivalence and different models for security and composition. Third, we establish the meta theorem showing what

we have to prove in the following to establish that the proofs in the abstract model are sufficient for the security in the hybrid execution model. Finally, we describe the secure protection domains and the core assumptions that we make regarding them and their execution environments.

2.1. Asynchronous Systems and Visual Notation

This section describes the core of the RSIM model for adaptive adversaries where a system is described by a fixed set of machines, for more details see in [25–27]. An asynchronous distributed system in RSIM consists of machines that we denote as boxes and communication buffers denoted by bullets. All machines communicate with outside world through ports. We denote input ports as white (□) and output ports as grey (■).

Standard buffers have three ports: input, output and clocking. In our notation, these ports are never drawn, as they are always used to connect ports of the machines. Instead, we denote a buffer as an arrow with a bullet (–●→). We extend this model by adding buffers which leak information to the machine that clocks it. These can be used to ensure that the adversary learns some meta information about the messages, such as the subprotocol instance that receives the message. For visual clarity, we omit these details and use an arrow with a dotted bullet (–●→) for the leaky buffer. We use a dedicated notation for sender-clocked (■–●→□) and receiver-clocked (□–●→■) buffers and omit port squares if they are deducible from context. By default all buffers are clocked by the adversary. The notation is illustrated in Figure 1. A message written to the input port of a buffer is appended to an internal queue of messages $q_1, \ldots, q_n$. A leaky buffer also has a corresponding queue of leaks $\ell_1, \ldots, \ell_n$ that is kept in sync. Leaks can be fetched using a dedicated port, thus the clocking machine must have at least one input port to receive the leakage. The full construction of it can be found in Appendix A. An input $i \in \mathcal{Z}$ to standard clocking port causes q_i to be removed from the queue and written to the output port. An empty output ϵ is written to the output if the input is out of range.

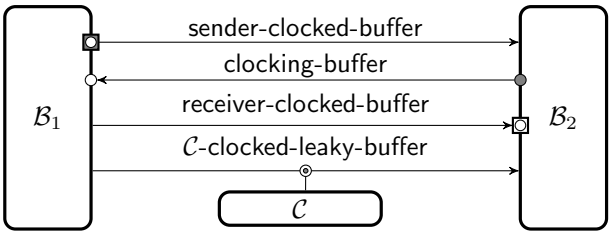


Figure 1. Notation for machines $\mathcal{B}_1$ and $\mathcal{B}_2$ communicating through various buffers with $\mathcal{C}$ clocking the leaky buffer.

Input–output behaviour of a machine is determined by a state update function $\delta : \mathcal{S} \times \mathcal{I} \to \mathcal{S} \times \mathcal{O}$, where $\mathcal{S}$ is the state space and $\mathcal{I}$ is the product of the domains of all input ports and $\mathcal{O}$ is the product of the domains of all output ports including clocking ports. All domains must contain an empty output ϵ. A machine can clock at most one buffer and thus only one clocking output can be non-empty. Execution rules also assure that one and only one input is non-empty when the machine is invoked except the main scheduler that can be invoked with empty inputs. As a result, a machine can clock only a single sender-clocked buffer and leaks cannot reach the clocker without explicit polling.

One machine is declared as the master scheduler that manages all undefined execution timings. In our setting, this machine is always either the adversary or the simulator. At the start of computations, the master scheduler is invoked. The scheduler will write to its output ports and clocks one buffer to start the chain of state transformations. When a machine writes a message to an output port, it is absorbed by the buffer and control goes back to the machine. When a message is written to a clocking port, the corresponding buffer releases a specified message and the control goes to the receiver. When a machine stops execution without clocking anything, the control goes to the master scheduler. The execution stops when the master scheduler reaches an end state and becomes inactive.

A collection $\mathcal{C}$ is a finite set of machines and buffers. It is closed if all its buffers are connected to ports and vice versa. A free connector is a connector that has one end attached to a buffer in a collection while the other end is not attached to any machine in the collection. Similarly, a free port is a port that belongs to a machine in a collection and is not connected to any buffer. An extended collection does not have free ports and a reduced collection does not have free connectors.

Collections $\mathcal{C}_1$ and $\mathcal{C}_2$ have matching interfaces if collections can be merged by joining free port and connector pairs while respecting restrictions posed by destination labels as well as ensuring there are no two ports expecting the same connection. Let the shorthand $\mathcal{C}_1\langle\mathcal{C}_2\rangle$ denote the resulting collection. Notation emphasises that $\mathcal{C}_2$ is a distributed subroutine that matches structural restrictions posed by $\mathcal{C}_1\langle\cdot\rangle$ calling it out. We also use a shorthand $\mathcal{C}_1\langle\mathcal{C}_2, \mathcal{C}_3\rangle$ for $\mathcal{C}_1\langle\mathcal{C}_2\langle\mathcal{C}_3\rangle\rangle$ to emphasise that $\mathcal{C}_1\langle\cdot\rangle$ is the outer environment although the concept is inherently symmetric. In this setting, the interface of $\mathcal{C}_2$ can be partitioned into two sets according to the target collection. We refer to these as sub-interfaces.

We visualise the interface of an extended collection as a dashed border surrounding its machines and buffers. Free connectors must reach a right port type on a border. For clarity, we label these interface ports by the names of their host machine, e.g., which buffers must be connected to the adversary A or environment Env.

2.2. Security through Observational Equivalence

Collections $\mathcal{C}_1$ and $\mathcal{C}_2$ have identical interfaces if there exists a one-to-one mapping between interface elements that respects port types and destination labels. A distinguisher $\mathcal{D}\langle\cdot\rangle$ is a reduced collection that has a matching interface and has a dedicated machine $\mathcal{D}_*$ with two end states 0 and 1. Let $\mathcal{D}\langle\mathcal{C}_i\rangle$ denote the end state of $\mathcal{D}_*$ when the collection stops. Then, the strongest equivalence form known as perfect observational equivalence $\mathcal{C}_1 \equiv \mathcal{C}_2$, which means that $\Pr[\mathcal{D}\langle\mathcal{C}_1\rangle = 1] = \Pr[\mathcal{D}\langle\mathcal{C}_2\rangle = 1]$ for any valid distinguisher $\mathcal{D}\langle\cdot\rangle$. Perfect observational equivalence indicates that $\mathcal{C}_1$ and $\mathcal{C}_2$ realise the same functionality modulo implementation details that are encapsulated by the collection border. Perfect observational equivalence is unattainable for cryptographic constructions as the security inherently emerges from the asymmetry between honest and corrupted parties.

Let Π be a collection that models a protocol. Then, the interface naturally splits into two sub-interfaces. A service interface specifies how to call out the protocol. An adversarial interface exposes protocol weaknesses to the adversary. The set of adversaries $\mathbb{A}$ is compatible with Π and Env if $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle$ is a well-defined and closed collection for any $\mathsf{A} \in \mathbb{A}$. Note that the definition allows collections $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle$ where Env and A are communicating. Similarly we can define a set of compatible environments $\mathbb{E}$.

Definition 1 (Security). *Let Π_1 and Π_2 be collections with an identical service interface and let $\mathbb{E}$ be the set of compatible environments. Let $\mathbb{A}_1, \mathbb{A}_2$ be the set of compatible adversaries. Then, Π_1 is as secure as Π_2 if there exists a function $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that $\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$ for all $\mathsf{A}_1 \in \mathbb{A}_1, \mathsf{Env} \in \mathbb{E}$.*

Let $\Pi_1 \geq \Pi_2$ denote that Π_1 is as secure as Π_2. The notation is justified as the relation is reflexive and transitive for appropriate sets of adversaries. The corresponding equivalence relation $\Pi_1 \equiv \Pi_2 \Leftrightarrow \Pi_1 \geq \Pi_2 \wedge \Pi_2 \geq \Pi_1$ captures protocols with similar security properties. Maximal elements over the relation identify maximally secure protocols, also known as ideal implementations.

This definition allows us to specify a wide spectrum of security definitions [28–31]. We can consider only nonuniform polynomial adversaries or different corruption models, for example, choose between static vs adaptive adversary, or semi-honest vs. active security [32,33]. The protocol Π_2 determines the set of unavoidable attacks. By tweaking the implementation of Π_2, we can model fairness [34,35], selective failure (abort) [36,37] and security against covert adversaries [38]. The exact definition of plausible environments determines how and where the protocol can be used securely. Restrictions on the correspondence ρ define various flavours of black-box [39] and white-box security [40,41] or

specify tightness requirements like polynomial and superpolynomial simulation [42,43]. Restrictions to $\mathbb{A}_1$ and $\mathbb{A}_2$ usually fix the model of corruption while constraints on $\mathbb{E}$ place restrictions on the protocol scheduling.

Theorem 1 (Secure two-system composition)**.** *Assume that we have three collections* Π_e, Π_1, Π_2 *such that collections* $\Pi_e\langle\Pi_1\rangle$ *and* $\Pi_e\langle\Pi_2\rangle$ *are well-defined and have an identical service interface. Let* $\mathbb{E}$ *be the subset of compatible environments and let* $\psi : \mathbb{E} \to \mathbb{E}^*$ *be a natural construction* $\psi(\mathsf{Env}) = \mathsf{Env}_o\langle\Pi_e\rangle$*. Then, the construction* $\phi : \mathbb{A}_1 \to \mathbb{A}_2$ *proves that* $\Pi_1 \geq \Pi_2$ *for the set of environments* $\mathbb{E}^*$ *is also a proof for* $\Pi_e\langle\Pi_1\rangle \geq \Pi_e\langle\Pi_2\rangle$ *for the set of environments* $\mathbb{E}$*.*

The theorem is particularly useful when the set of plausible environments and adversaries is closed, i.e., $\mathbb{E} = \mathbb{E}^*$ and $\mathbb{A}_1 = \mathbb{A}_2$. As security is commonly defined against nonuniform polynomial-time adversaries the second constraint is trivially satisfied. The first constraint is satisfied when environments consist of all sequential compositions of poly-time subprotocols. The resulting sequential composition theorems [29,30,44] play a central role in cryptography. Alternatively, we can consider the set of all concurrent compositions of poly-time subprotocols. The resulting security notion is known as universal composability (UC) and has many flavours [14,25,45–50] which differ in minor details. Most formalisations assume that machines and connections between them remain unaltered during the execution while Canetti's second formalisation of universal composability [51] allows dynamic reconfiguration of the environment. We consider an extension of the RSIM model [25,27] which has leaky buffers for proper modelling of secure communication channels. The resulting adaptive-adversary RSIM is very close to the simplified version of UC (SUC) that was defined to characterise MPC protocols [52]. Our formalisation of MPC in RSIM gives us more flexibility to split protocols into components to modularise the proofs and transformations.

2.3. Soundness and Completeness Theorems

Our main contribution is a description of an abstract execution model which hides all irrelevant technical details while the security proof in this model remains sound and complete. That is, a proof in the abstract model exists if and only if it exists in the original execution model. In other words, the abstract model is both sound and complete and, therefore, a suitable replacement for the hybrid execution model. Here, soundness means that a proof in the abstract setting means that there is also a proof in the original execution model. Completeness, on the other hand, specifies that if there is a proof in the original model, then there is also a proof in the abstract model.

Let Π_1 and Π_2 be the protocols of interest, and let Π_1^* and Π_2^* be their counterparts in the abstract execution model. Let $\mathbb{E}$ and $\mathbb{E}^*$ denote the set of environments for original and abstract execution models. Let $\mathbb{A}_1, \mathbb{A}_2$ and $\mathbb{A}_1^*, \mathbb{A}_2^*$ denote the set of plausible adversaries. To show that security proofs in the abstract model are sound and complete, we define three explicit constructions and their semi-inverses

$$\begin{array}{lll} \psi : \mathbb{E} \to \mathbb{E}^* & \phi_1 : \mathbb{A}_1 \to \mathbb{A}_1^* & \phi_2 : \mathbb{A}_2 \to \mathbb{A}_2^* \\ \psi^* : \mathbb{E}^* \to \mathbb{E} & \phi_1^* : \mathbb{A}_1^* \to \mathbb{A}_1 & \phi_2^* : \mathbb{A}_2^* \to \mathbb{A}_2 \end{array} \tag{1}$$

which satisfy the following three pairs of equivalence relations

$$\forall \mathsf{Env} \in \mathbb{E} : \forall \mathsf{A}_1 \in \mathbb{A}_1 : \quad \mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \psi(\mathsf{Env})\langle\Pi_1^*, \phi_1(\mathsf{A}_1)\rangle$$
$$\forall \mathsf{Env}^* \in \mathbb{E}^* : \forall \mathsf{A}_1^* \in \mathbb{A}_1^* : \quad \mathsf{Env}^*\langle\Pi_1^*, \mathsf{A}_1^*\rangle \equiv \psi^*(\mathsf{Env}^*)\langle\Pi_1, \phi_1^*(\mathsf{A}_1^*)\rangle \tag{2}$$

$$\forall \mathsf{Env} \in \mathbb{E} : \forall \mathsf{A}_2 \in \mathbb{A}_2 : \quad \mathsf{Env}\langle\Pi_2, \mathsf{A}_2\rangle \equiv \psi(\mathsf{Env})\langle\Pi_2^*, \phi_2(\mathsf{A}_2)\rangle$$
$$\forall \mathsf{Env}^* \in \mathbb{E}^* : \forall \mathsf{A}_2^* \in \mathbb{A}_2^* : \quad \mathsf{Env}^*\langle\Pi_2^*, \mathsf{A}_2^*\rangle \equiv \psi^*(\mathsf{Env}^*)\langle\Pi_2, \phi_2^*(\mathsf{A}_2^*)\rangle \tag{3}$$

$$\forall \mathsf{Env} \in \mathbb{E} : \forall \mathsf{A}_2 \in \mathbb{A}_2 : \quad \mathsf{Env}\langle\Pi_2, \mathsf{A}_2\rangle \equiv \psi^*(\psi(\mathsf{Env}))\langle\Pi_2, \mathsf{A}_2\rangle$$
$$\forall \mathsf{Env}^* \in \mathbb{E}^* : \forall \mathsf{A}_2^* \in \mathbb{A}_2^* : \quad \mathsf{Env}^*\langle\Pi_2^*, \mathsf{A}_2^*\rangle \equiv \psi(\psi^*(\mathsf{Env}^*))\langle\Pi_2, \mathsf{A}_2\rangle \ . \tag{4}$$

Note that constructions (1) together with equivalence relations (2)–(4) define a commutative square in Figure 2 with the equivalence guarantees for individual elements where for brevity pairs $\mathsf{Env}^*, \mathsf{Env}$, $\mathsf{A}_1, \mathsf{A}_1^*$ and $\mathsf{A}_2, \mathsf{A}_2^*$ are defined through up or down arrows depending on the direction of traversal. As a result, the existence of ρ implies the existence of ρ^*, and vice versa.

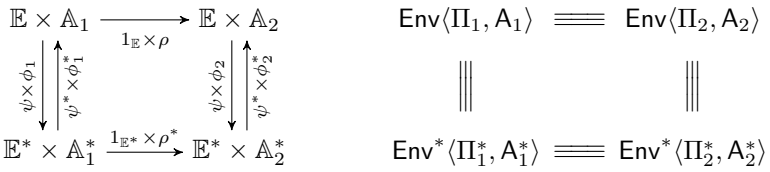


Figure 2. Equivalence guarantees and their relations to ρ and ρ^*.

Theorem 2. *Let $\psi : \mathbb{E} \to \mathbb{E}^*$, $\phi_1 : \mathbb{A}_1 \to \mathbb{A}_1^*$ and $\phi_2 : \mathbb{A}_2 \to \mathbb{A}_2^*$ be constructions with semi-inverses $\psi^*, \phi_1^*, \phi_2^*$ that satisfy the equivalence relations (2)–(4). Then, $\Pi_1 \geq \Pi_2$ for environments $\mathbb{E}$ if and only if $\Pi_1^* \geq \Pi_2^*$ for environments $\mathbb{E}^*$.*

Proof. For the proof, we simply trace the equivalence square depicted in Figure 2.
SOUNDNESS. Assume that there exists $\rho^* : \mathbb{A}_1^* \to \mathbb{A}_2^*$ such that

$$\mathsf{Env}^*\langle \Pi_1^*, \mathsf{A}_1^* \rangle \equiv \mathsf{Env}^*\langle \Pi_2^*, \mathsf{A}_2^* \rangle \tag{5}$$

The equivalence relations (2)–(5) assure that

$$\begin{aligned}
\mathsf{Env}\langle \Pi_1, \mathsf{A}_1 \rangle &\equiv \psi(\mathsf{Env})\langle \Pi_1^*, \phi_1(\mathsf{A}_1) \rangle \\
\psi(\mathsf{Env})\langle \Pi_1^*, \phi_1(\mathsf{A}_1) \rangle &\equiv \psi(\mathsf{Env})\langle \Pi_2^*, \rho^*(\phi_1(\mathsf{A}_1)) \rangle \\
\psi(\mathsf{Env})\langle \Pi_2^*, \rho^*(\phi_1(\mathsf{A}_1)) \rangle &\equiv \psi^*(\psi(\mathsf{Env}))\langle \Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1))) \rangle \\
\psi^*(\psi(\mathsf{Env}))\langle \Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1))) \rangle &\equiv \mathsf{Env}\langle \Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1))) \rangle \ .
\end{aligned}$$

The claim follows as we can define $\rho = \phi_2^* \circ \rho^* \circ \phi_1$ and the transitivity of equivalence relation proves the equivalence $\mathsf{Env}\langle \Pi_1, \mathsf{A}_1 \rangle \equiv \mathsf{Env}\langle \Pi_2, \rho(\mathsf{A}_1) \rangle$.
COMPLETENESS Assume that there exists $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\mathsf{Env}\langle \Pi_1, \mathsf{A}_1 \rangle \equiv \mathsf{Env}\langle \Pi_2, \mathsf{A}_2 \rangle \tag{6}$$

The equivalence relations (2)–(4) and (6) assure that

$$\begin{aligned}
\mathsf{Env}^*\langle \Pi_1^*, \mathsf{A}_1^* \rangle &\equiv \psi^*(\mathsf{Env}^*)\langle \Pi_1, \phi_1^*(\mathsf{A}_1^*) \rangle \\
\psi^*(\mathsf{Env}^*)\langle \Pi_1, \phi_1^*(\mathsf{A}_1^*) \rangle &\equiv \psi^*(\mathsf{Env}^*)\langle \Pi_2, \rho(\phi_1^*(\mathsf{A}_1^*)) \rangle \\
\psi^*(\mathsf{Env}^*)\langle \Pi_2, \rho(\phi_1^*(\mathsf{A}_1^*)) \rangle &\equiv \psi(\psi^*(\mathsf{Env}^*))\langle \Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*))) \rangle \\
\psi(\psi^*(\mathsf{Env}^*))\langle \Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*))) \rangle &\equiv \mathsf{Env}^*\langle \Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*))) \rangle \ .
\end{aligned}$$

The claim follows as we can define $\rho^* = \phi_2 \circ \rho \circ \phi_1^*$ and the transitivity of equivalence relation proves $\mathsf{Env}^*\langle \Pi_1^*, \mathsf{A}_1^* \rangle \equiv \mathsf{Env}^*\langle \Pi_2^*, \rho^*(\mathsf{A}_1^*) \rangle$. □

In many cases, the security definitions limit the resource consumption of the parties, e.g., the adversaries are polynomial, in such cases ψ, ϕ, ρ have to keep these restrictions. We apply this theorem to show that we can hide the vast majority of technical details when analysing the security of a compound protocol. We split the construction into four major blocks. In Section 3.1, we show that certain attack techniques do not help the adversary when the protocol Π satisfies natural requirements to message scheduling. Thus, we can consider only a subset of adversaries, i.e., we partition $\mathbb{A}_1$ and $\mathbb{A}_2$ and choose a canonical representative for each class. As a result, the environment remains the same during the abstraction and $\phi_1^* : \mathbb{A}_1^* \to \mathbb{A}_1$, $\phi_2^* : \mathbb{A}_2^* \to \mathbb{A}_2$ are also identity functions.

In Section 3.2, we separate state from protocol participants and replace message passing with a shared memory. Again the environment remains the same but now $\mathbb{A}_1^* \not\subseteq \mathbb{A}_1$ and $\mathbb{A}_2^* \not\subseteq \mathbb{A}_2$. As a result, we need to explicitly define ϕ_1^* and ϕ_2^*. In Section 3.3, we expose the internals of ideal functionalities to further simplify the memory model and remove the share representation. We again explicitly define ϕ_1^* and ϕ_2^*. In Section 3.4, we define the abstract model by simplifying the environment to a simple representative class of environments. Fortunately, we can define ψ and ψ^* so that observational equivalence guarantees

$$\begin{aligned} \forall \mathsf{Env} \in \mathbb{E} : \quad & \psi^*(\psi(\mathsf{Env})) \equiv \mathsf{Env} \\ \forall \mathsf{Env}^* \in \mathbb{E}^* : \quad & \psi(\psi^*(\mathsf{Env}^*)) \equiv \mathsf{Env}^* \end{aligned}$$

hold and the last pair of equivalence relations (4) follows directly.

2.4. Programmable Multiparty Computation

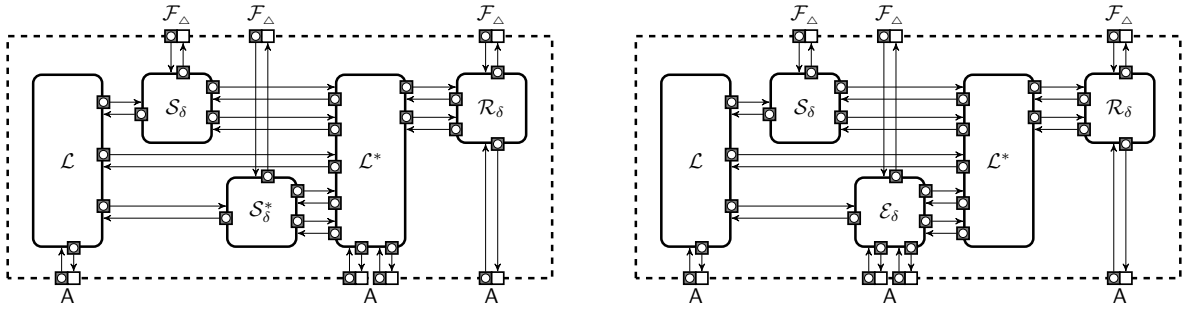
Most platforms for multiparty computations, see, e.g., in [1,2,4], consist of a secure storage and a system of primitive protocols operating on top of the storage. As a result, one can safely combine primitive instructions to implement any algorithm, thus we call such frameworks programmable. In this section, we formalise building blocks and show how one can extend existing secure computation instruction set with new primitives. Our work falls into a long list of MPC formalisations [14,25,29,52–54] where we are focused on specifying programmable secure computation. First, we discuss the storage properties, then we formalise two flavours of computations of the secure computation engine and define protection domains as our abstraction of a secure computation framework. For protection domains, we discuss their security and the natural conditions for environments and adversaries that we expect in our following security analysis.

2.4.1. Security of Distributed Storage Domains

A modular design of multiparty computation protocols requires the ability to store intermediate values. A secure storage can be built on top of different primitives, such as secret sharing, encryption, commitments, trusted hardware or a combination of different schemes. We develop the formalism for secret sharing, however, the abstract description for storing and retrieving values is universal. A secure storage domain δ is defined by two algorithms $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ which can use parameters from shared setup $\mathcal{F}_\triangle$. A machine $\mathcal{S}_\delta$ distributes an input $x \in \mathcal{X}_\delta$ into shares. A machine $\mathcal{R}_\delta$ converts shares back to the original value or returns a special failure symbol $\perp$. Throughout the paper, we explicitly assume that the behaviour of $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ does not depend on previous queries.

An adversarial structure $\mathcal{A}_\delta$ defines which subsets of parties can be corrupted without losing security properties. We define privacy and integrity properties for secure storage through observational equivalence. Our definitions are generalisations of privacy and recoverability of secret sharing schemes [55] and are tailored toward the concrete application of secret sharing as a storage domain.

Intuitively, a storage is hiding if no information about the stored value leaks from shares captured by the adversary. However, there are three subtle issues: First, the outcomes of $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ could leak information about private parameters or other shared values. Second, in security proofs we often want to simulate shares for some values and use the remaining shares without changes. Third, we need to specify what happens when the adversary corrupts more parties than expected. Formally, we define the hiding property through two collections $\mathfrak{B}_0$ and $\mathfrak{B}_1$ that have identical layout, see Figure 3a. Machines $\mathcal{L}$ and $\mathcal{L}^*$ are for storing values and the corresponding shares. The state of $\mathcal{L}$ is a one-dimensional array $\mathfrak{s}$. The state of $\mathcal{L}^*$ consists of a two-dimensional array $\mathfrak{s}_*$ for shares and a one-dimensional array $\mathfrak{b}$ that specifies how the shares will be generated.



(**a**) Hiding property. (**b**) Modification awareness and limited control.

Figure 3. Configurations defining security properties of a storage domain.

An adversary A can adaptively specify the values of $\mathfrak{s}[\ell]$ and $\mathfrak{b}[\ell]$, but each location can only be set once. The adversary A can also read and write shares $\mathfrak{s}_*[\ell_k, i_k]$ of corrupted parties $\mathcal{P}_{i_k}$. A static adversary must send the list of corrupted parties to $\mathcal{L}^*$ before any value is shared while an adaptive adversary can issue corruption calls at any moment. When $\mathfrak{s}_*[\ell, i]$ is queried and the location is uninitialised, $\mathcal{L}^*$ initiates an update cycle. The machine $\mathcal{L}^*$ always asks $\mathcal{L}$ to share the value $\mathfrak{s}[\ell]$ using $\mathcal{S}_\delta$ in collection $\mathfrak{B}_0$. In collection $\mathfrak{B}_1$, the value $\mathfrak{s}[\ell]$ is shared only if $\mathfrak{b}[\ell] = 0$. If $\mathfrak{b}[\ell] = 1$, then $\mathcal{L}^*$ asks a share simulator $\mathcal{S}^*_\delta$ to create the share $\mathfrak{s}_*[\ell, i]$. A share simulator $\mathcal{S}^*_\delta$ is an efficient and potentially stateful machine, which can query values $\mathfrak{s}[\ell]$ from $\mathcal{L}$ only after the set of corrupted parties does not belong into $\mathcal{A}_\delta$. Finally, A can place a reconstruction order for $\mathfrak{s}_*[\ell]$. The machine $\mathcal{L}^*$ sends $\mathfrak{s}_*[\ell]$ to $\mathcal{R}_\delta$ if $\mathfrak{b}[\ell] = 0$. Otherwise, $\mathcal{L}^*$ first asks $\mathcal{L}$ to share the value $\mathfrak{s}[\ell]$ and then forwards the shares to $\mathcal{R}_\delta$. In both cases $\mathcal{R}_\delta$ sends the output back to A.

Definition 2 (Hiding storage)**.** *A storage domain δ is perfectly hiding if no adversary can distinguish configurations $\mathfrak{B}_0$ and $\mathfrak{B}_1$. A storage domain δ is hiding for $\mathbb{A}$ if the advantage is negligible for any adversary from $\mathbb{A}$.*

Many secret sharing schemes do not use private setup parameters. As a result, different sharings are independent from each other and it is sufficient to prove simulatability of a single sharing. In case of adaptive corruption, the secret sharing scheme must be efficiently patchable [56] as the simulator must progressively disclose shares of an unknown value. The existence of trusted setup $\mathcal{F}_\triangle$ allows to achieve integrity even for honest minority. However, now different shares are correlated with each other due to shared setup parameters and we cannot reduce hiding to simpler security notions. In this case, we assume that $\mathcal{S}^*_\delta$ uses the setup parameters of the corrupted parties.

Note that the hiding property does not guarantee privacy throughout the entire period of computations. Instead, each storage domain can define $\mathcal{A}_\delta$ to specify which parties can be corrupted while still maintaining privacy. For instance, the adversary who corrupts $\mathcal{P}_i$ learns its local state which is a separate trivial storage domain. Values in the public domain become visible as soon as the adversary corrupts some participant.

As the adversary can always change shares under its control, security of a compound protocol relies on the integrity of stored values. Robust secret sharing guarantees that values cannot be altered while verifiable secret sharing allows to detect corrupted values. Modification awareness for a storage domain δ is defined through an efficient extractor machine $\mathcal{E}_\delta$ and an efficient operation $\ominus_\delta$. Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ be the original secret sharing where x_i is the share of $\mathcal{P}_i$, and let $\hat{\boldsymbol{x}} = (\hat{x}_1, \ldots, \hat{x}_n)$ be its adversarial modification and A the set of corrupted parties. Then, the extractor gets $(x_i)_{i\in A}, (\hat{x}_i)_{i\in A}$ together with the setup parameters of A as an input and has to output a difference Δ such that $\mathcal{R}_\delta(\boldsymbol{x}) \ominus_\delta \Delta = \mathcal{R}_\delta(\boldsymbol{x}')$. We denote reconstruction failures by $\perp$ and we expect that the modification operator is such that $a \ominus_\delta \perp = \perp$ and $\perp \ominus_\delta a = \perp$ for any a in the value domain. Modification function generalises the observation that in many MPC protocols adversarial modifications result in additive changes to the value [57].

Intuitively, a storage domain is modification aware if there exists a good extractor machine $\mathcal{E}_\delta$ which cannot be fooled by an adversary. The success of an adversary is defined through a collection $\mathfrak{B}_3$ that also contains machines $\mathcal{L}$ and $\mathcal{L}^*$, see Figure 3b. A two-dimensional array $\mathfrak{s}_*$ forms the entire state of $\mathcal{L}^*$. As before, an adversary A can adaptively specify the values of $\mathfrak{s}[\ell]$ but each value can be set only once. The adversary A has the power to corrupt parties and read and modify the shares of corrupted parties by interacting with $\mathcal{E}_\delta$. The extractor $\mathcal{E}_\delta$ just forwards communication between A and $\mathcal{L}$. Queries to uninitialised locations $\mathfrak{s}_*[\ell]$ lead to the same update cycle as in $\mathfrak{B}_0$, i.e., $\mathcal{S}_\delta$ generates shares from $\mathfrak{s}[\ell]$. During a share update query $\mathcal{E}_\delta$ additionally computes the difference Δ and sends a pair ℓ, Δ to $\mathcal{L}$. As a response, $\mathcal{L}$ updates the value $\mathfrak{s}[\ell] = \mathfrak{s}[\ell] \ominus_\delta \Delta$ and gives control back to $\mathcal{E}_\delta$. Each sharing in $\mathcal{L}^*$ can be updated by A at most once. The limit on modifications attempts eliminates trivial attacks where the adversary first invalidates its shares and then changes them back to original values. For many storage schemes this causes $\mathcal{E}_\delta$ to fail as $\mathcal{E}_\delta$ is stateless and has no knowledge of the previous modification or share values, thus it has to assume that $\perp \ominus_\delta \Delta = \perp$ for any Δ. The adversary A can also place reconstruction orders for $\mathfrak{s}_*[\ell]$. Given such an order $\mathcal{L}^*$ sends $\mathfrak{s}_*[\ell]$ to $\mathcal{R}_\delta$ who sends the output back to A. The adversary A wins the game if the outcome of $\mathcal{R}_\delta$ differs from $\mathfrak{s}[\ell]$ and the set of corrupted parties is in $\mathcal{A}_\delta$.

Definition 3 (Modification awareness)**.** *A storage domain δ is modification aware against a class of adversaries $\mathbb{A}$ if the advantage against the modification game is negligible for any* $\mathsf{A} \in \mathbb{A}$.

For robust secret sharing, the extractor $\mathcal{E}_\delta$ always outputs the neutral element of the modification operator because the adversary cannot affect the shared value. Local storage is robust by definition as the adversary cannot alter local state before corrupting the party and after corruption the definition poses no restrictions to modifications. Extractor $\mathcal{E}_\delta$ for verifiable secret sharing can output $\perp$ or the neutral element because the adversary can either invalidate the shared value or create a different sharing of the same value. Therefore, for most cases the potential changes to the value are quite limited.

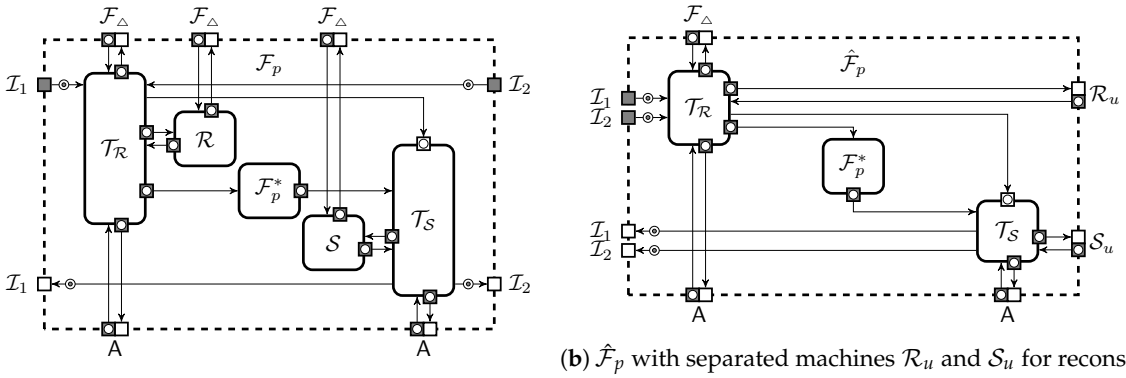
On the other hand, modification awareness does not guarantee that the adversary can efficiently find a share modification for any potential change Δ of the value. A two-way extractor $\mathcal{E}_\delta$ can handle such requests. The success of an adversary against the two-way extractor $\mathcal{E}_\delta$ is defined through a collection $\mathfrak{B}_4$ that has the same layout as $\mathfrak{B}_3$ in Figure 3b. An adversary A can adaptively initialise and update the values of $\mathfrak{s}[\ell]$. For the update, A has to provide Δ to $\mathcal{L}$. As a response $\mathcal{L}$ sets $\mathfrak{s}[\ell] = \mathfrak{s}[\ell] \ominus_\delta \Delta$, sends ℓ, Δ to $\mathcal{E}_\delta$. Given ℓ, Δ from $\mathcal{L}$, the extractor $\mathcal{E}_\delta$ first fetches $(x_i)_{i \in A}$ from $\mathcal{L}^*$ and computes new shares $(\hat{x}_i)_{i \in A}$ for the corrupted parties such that $\mathcal{R}_\delta(\boldsymbol{x}) \ominus_\delta \Delta = \mathcal{R}_\delta(\hat{\boldsymbol{x}})$. Finally, $\mathcal{E}_\delta$ sends $(\hat{x}_i)_{i \in A}$ to $\mathcal{L}^*$ and returns the control to $\mathcal{L}$. $\mathcal{L}$ returns control to A. The rest of the collection specification is identical to $\mathfrak{B}_3$. As before, A must issue a reconstruction order for a location $\mathfrak{s}_*[\ell]$ at the end of the game. The adversary breaks the two-way extractor if the outcome of $\mathcal{R}_\delta$ differs from $\mathfrak{s}[\ell]$ and the set of corrupted parties A is in $\mathcal{A}_\delta$.

Definition 4 (Limited control)**.** *A class of adversaries has a limited control over a storage domain δ if the advantage in the collection $\mathfrak{B}_4$ is negligible for any adversary from the class.*

2.4.2. Canonical Description of Ideal Functionalities

An idealised computation $\mathcal{F}_p$ can be formalised in many different ways. We consider a decomposable functionality working on the data representation of the given protection domain and using the setup from the protection domain. In general, inputs and outputs of an ideal functionality $\mathcal{F}_p$ may belong to several storage domains, e.g., some of them may be secret shared while the others are local variables. The entire computational process can be split into rounds where each round consists of three phases: reconstruction, computation and sharing as in Figure 4. Machines $\mathcal{S}$ and $\mathcal{R}$ are functionalities for sharing and reconstruction which internally execute functionalities $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ of individual storage domains and $\mathcal{F}_\triangle$ is a combined setup procedure. Machine $\mathcal{T}_\mathcal{R}$ gathers inputs and interacts

with $\mathcal{R}$ to reconstruct the values. These values are passed to $\mathcal{F}_p^*$ that evaluates a stateful function and sends outputs to $\mathcal{T}_\mathcal{S}$ together with a storage domain for each output. $\mathcal{T}_\mathcal{S}$ interacts with $\mathcal{S}$ to share outputs. If the reconstruction fails then $\mathcal{F}_p^*$ also outputs $\perp$ and the storage domain has to have a way to generate shares of $\perp$. Most notably, there must be a canonical way to create shares of $\perp$ for verifiable secret sharing. Note that it is always possible to define functionalities that ignore some input and where such condition may not be necessary. However, if the input is used in the computation of the functionality then a malformed input can only result in a malformed output, otherwise the functionality leaks information or introduces selective failures. The adversary A can interact with $\mathcal{F}_p$ through $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$. Machines $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$ can coordinate their actions through a receiver-clocked buffer between $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$. Note that $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{T}_\mathcal{R}$ all receive setup parameters, however there is an important distinction that we expect only $\mathcal{S}$ and $\mathcal{R}$ to get the private parameters of different parties and $\mathcal{T}_\mathcal{R}$ only learns any public parameters. The latter is necessary to correctly define $\mathcal{F}_p^*$ because public parameters might, for example, specify the modulus for all computations.



(**a**) $\mathcal{F}_p$.

(**b**) $\hat{\mathcal{F}}_p$ with separated machines $\mathcal{R}_u$ and $\mathcal{S}_u$ for reconstruction and sharing.

Figure 4. Internal structure of a two-party functionalities.

The corruption mode for $\mathcal{F}_p$ is defined through a communication between $\mathcal{T}_\mathcal{R}$, $\mathcal{T}_\mathcal{S}$ and A. All responses to A must be computable from the inputs and outputs of the protocol instance and the setup parameters received by $\mathcal{F}_p$. For robust protocols, A is disconnected from $\mathcal{F}_p$. For fair protocols, A can send only abort signals to $\mathcal{T}_\mathcal{S}$ but gets no information from $\mathcal{F}_p$. For protocols without fairness the adversary could see corrupted parties outputs before deciding to abort. After abort, all parties get shares of $\perp$ as output. Protocol instances inside $\mathcal{F}_p$ are distinguished by instance tags sent by protocol participants $\mathcal{I}_i$. All protocol instances are run concurrently and independently.

Definition 5 (Canonical ideal functionality)**.** *An ideal functionality $\mathcal{F}_p$ has a standard corruption mode if all outputs are generated by $\mathcal{S}$ and the adversary cannot learn anything about the shares of honest parties other than revealed by the published values. Functionality $\mathcal{F}_p$ is in canonical form if it is a collection of $\mathcal{T}_\mathcal{R}$, $\mathcal{R}$, $\mathcal{F}_p^*$, $\mathcal{T}_\mathcal{S}$ and $\mathcal{S}$ with the internal structure specified above, has a standard corruption mode, and always outputs $\perp$ if any input is $\perp$.*

2.4.3. Canonical Description of Local Functionalities

Ideal functionalities define operations computed together with other parties. Each party may also perform local operations with their shares. In principle, all kinds of local operations are possible. However, usually the storage domain defines a set of meaningful operations where the local operations are also meaningful operations on the shared values. For example, local linear operations are possible for linear sharing schemes. By definition, the output share of the local functionality depends only on the input shares of this party and, therefore, local functionalities cannot be described as canonical ideal functionalities.

Definition 6 (Meaningful local operation)**.** *A local operation $\mathcal{G}_q$ implements a function g_q if for any input $(y_1, \ldots, y_t) = g_q(x_1, \ldots, x_s)$ where $x_1, \ldots, x_s$ are the values reconstructed from the input shares of $\mathcal{G}_q$ and $y_1, \ldots, y_t$ are the values reconstructed from the output shares of $\mathcal{G}_q$.*

In the following, we represent local operations as $\mathcal{G}_1, \ldots, \mathcal{G}_g$ where each $\mathcal{G}_q$ is a collection of machines $\{\mathcal{G}_{q,j}\}_{j \in \mathcal{J}_q}$ implementing local operations carried out by parties in $\mathcal{J}_q$. We assume that each local operation consist of a single round. More complex local computations can be implemented as a series of local computations.

2.4.4. Security of Protection Domains

A protection domain consists of storage domains and computation protocols $\Pi_1, \ldots, \Pi_k$. For instance, a secure computation engine in the Arithmetic Black-Box model [15] is a specific protection domain with no explicit access to the stored values. Protection domain is also a refinement of a standard MPC deployment model [11] which divides participants into input, result and computing parties.

Let $\mathcal{F}_1, \ldots, \mathcal{F}_k$ be the canonical ideal functionalities that the protection domain should implement. We will simplify the ideal functionalities by joining their sharing and reconstruction components. More formally, let $\mathcal{R}_u$ be a machine that has k port pairs for reconstruction. A query to p-th pair is sent internally to $\mathcal{R}$ that is part of the ideal functionality $\mathcal{F}_p$ and the reply is routed back to the corresponding output port. Let $\mathcal{S}_u$ be analogous extension of the machine $\mathcal{S}$. Note that $\mathcal{R}_u$ and $\mathcal{S}_u$ have only a single port pair for $\mathcal{F}_\triangle$. Let $\hat{\mathcal{F}}_p$ be a collection we obtain by removing components $\mathcal{R}$ and $\mathcal{S}$ from $\mathcal{F}_p$ as depicted in Figure 4b. Then, the ideal functionality for the protection domain is defined through a collection $\hat{\mathcal{F}}_1, \ldots, \hat{\mathcal{F}}_k, \mathcal{R}_u, \mathcal{S}_u$ where $\hat{\mathcal{F}}_p$ only use public parameters. Let the corresponding extended collection be denoted as $\mathcal{F}_{pd}$. The collection $\mathcal{F}_1, \ldots, \mathcal{F}_p$ is observationally equivalent to $\mathcal{F}_{pd}$ provided that the trusted setup $\mathcal{F}_\triangle$ sends the same input parameters for all $\mathcal{F}_1, \ldots, \mathcal{F}_k$.

Definition 7. *A protection domain has modular representation if collections $\mathcal{F}_\triangle\langle\mathcal{F}_{pd}\rangle$ and $\mathcal{F}_\triangle\langle\mathcal{F}_1, \ldots, \mathcal{F}_k\rangle$ are observationally equivalent.*

As a next step we need to specify the class of reasonable environments. A typical compound protocol in an MPC platform take shares as input and produces shares as output. The latter causes subtle issues. Note that a universally composable protocol must remain secure even if the adversary knows all inputs, while a verifiable secret sharing could be secure only if the adversary knows only a limited set of inputs. Consequently, universal composability is unachievable as the adversary can alter shared values without detection. However, these protocols do not run in a generic environment, and therefore we should also study their security in a more restricted setting where we separate the outside environment and the other computations that happen in the protection domain. This allows us to make reasonable assumptions about the visibility of the output shares. In the most general case, the best we can achieve if we have any shared setup, is joint-state universal composability [24] while joint-state sequential composition is the absolute minimum.

We define the set of plausible environments through a cartesian product $\{\mathcal{F}_\triangle\} \times \mathbb{E}_\mathsf{o} \times \mathbb{P}$ where $\mathcal{F}_\triangle$ is the trusted setup and $\mathbb{P}$ specifies all plausible compound protocols Π_e and $\mathbb{E}_\mathsf{o}$ environments Env_o in which Π_e might be executed. The inner environment $\Pi_\mathsf{e}\langle\cdot\rangle$ specifies computations done in the MPC framework while $\mathsf{Env}_\mathsf{o}\langle\cdot\rangle$ is an outer environment representing the rest of the world in which the compound protocol should preserve security.

Definition 8. *A list of protocols $\Pi_1, \ldots, \Pi_k$ with a shared setup $\mathcal{F}_\triangle$ is secure protection domain if $\mathcal{F}_\triangle\langle\Pi_\mathsf{e}\langle\Pi_1, \ldots, \Pi_k\rangle\rangle \geq \mathcal{F}_\triangle\langle\Pi_\mathsf{e}\langle\mathcal{F}_{pd}\rangle\rangle$ for any $\Pi_\mathsf{e} \in \mathbb{P}$ and $\mathsf{Env}_\mathsf{o} \in \mathbb{E}_\mathsf{o}$ provided that $\mathsf{Env}_\mathsf{o}\langle\mathcal{F}_\triangle\langle\Pi_\mathsf{e}\langle\Pi_1, \ldots, \Pi_k\rangle\rangle, \mathsf{A}\rangle$ is a well-defined closed collection.*

The signature of a protection domain $(\mathbb{E}, \mathbb{P}, \mathcal{F}_1, \ldots, \mathcal{F}_k)$ determines what can be computed with the protection domain and what restrictions must be met to preserve security.

The actual security guarantees are specified in terms of plausible adversaries $\mathbb{A}_1$ and $\mathbb{A}_2$ which depend on the environment. Each protection domain also specifies an adversary structure $\mathcal{A}_\delta$ which lists all sets of parties that the adversary can corrupt if we want to keep security guarantees. The adversary structure is limited by the secure storage domains of the protection domain and the security properties of the individual protocols in the protection domain. The corruption mode specifies the type of tolerated adversaries ranging from static semi-honest to adaptive malicious adversaries.

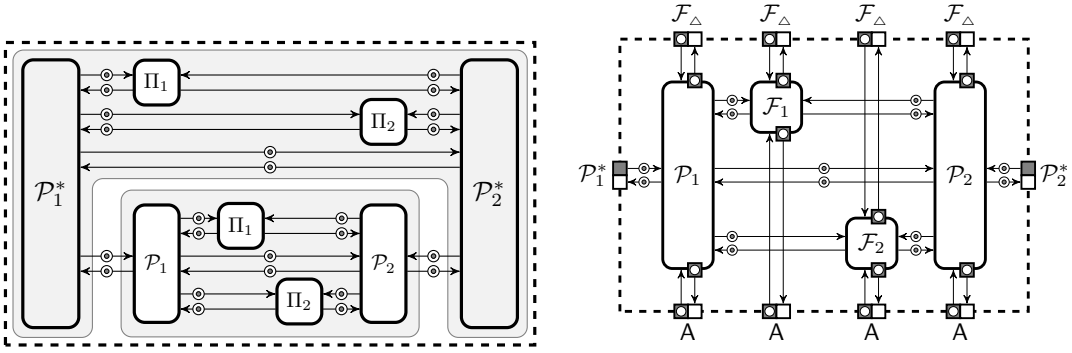
2.4.5. Secure Extension of Protection Domains

To modularise proofs, a protection domain is often defined through a minimal set of protocols $\Pi_1, \ldots, \Pi_k$ that implement ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$. After that each new primitive $\mathcal{F}_0$ is added by defining a protocol $\Pi\langle\Pi_1, \ldots, \Pi_k\rangle$ and proving its security. To establish basic security, we need to prove $\mathcal{F}_\triangle\langle\Pi\langle\Pi_1, \ldots, \Pi_k\rangle\rangle \geq \mathcal{F}_\triangle\langle\mathcal{F}_0\rangle$. Such proofs usually follow the two phase strategy where one proves

$$\mathcal{F}_\triangle\langle\Pi\langle\Pi_1, \ldots, \Pi_k\rangle\rangle \geq \mathcal{F}_\triangle\langle\Pi\langle\mathcal{F}_{pd}\rangle\rangle \geq \mathcal{F}_\triangle\langle\mathcal{F}_0\rangle \ .$$

The first step follows directly from the security definition of a protection domain. Thus, the main bulk of the proof must be carried out in the hybrid execution model where real protocol implementations are replaced with $\mathcal{F}_1, \ldots, \mathcal{F}_k$.

To show validity of the extension, we must analyse the extended protection domain $\Pi\langle\Pi_1, \ldots, \Pi_k\rangle, \Pi_1, \ldots, \Pi_k$ when using the ideal implementation $\mathcal{F}_{pd}$. For that we have to analyse compound protocols $\Pi_\mathsf{e}\langle\Pi\langle\Pi_1, \ldots, \Pi_k\rangle, \Pi_1, \ldots, \Pi_k\rangle$. It is easy to restructure these compound protocols into observationally equivalent collections $\Pi_{**}\langle\Pi_1, \ldots, \Pi_k, \Pi_1, \ldots, \Pi_k\rangle$. For example, Π_{**} can be obtained by joining $\mathcal{P}_i$ and $\mathcal{P}_i^*$ on Figure 5. Formally, the list of protocols $\Pi_1, \ldots, \Pi_k, \Pi_1, \ldots, \Pi_k$ may not be a secure protection domain as each protocol occurs twice. It is easy to see that a secure protection domain is securely extendable provided that each protocol instance is independent of the other instances of the same protocol. The next theorem describes under which conditions the second proof stage can be generalised.



(**a**) $\Pi_\mathsf{e}\langle\Pi\langle\Pi_1, \Pi_2\rangle, \Pi_1, \Pi_2\rangle$. (**b**) $\Pi\langle\mathcal{F}_1, \mathcal{F}_2\rangle$.

Figure 5. Protection domain extension with protocols and ideal functionalities where Π_e is carried out by $\mathcal{P}_i^*$ and Π is carried out by $\mathcal{P}_i$.

Definition 9. *Protocols $\Pi_1, \ldots, \Pi_k$ with shared setup $\mathcal{F}_\triangle$ are a securely extendable protection domain if the list of protocols $\Pi_1, \Pi_1, \ldots, \Pi_k, \Pi_k$ with $\mathcal{F}_\triangle$ is also a secure protection domain.*

Theorem 3. *Let $\Pi_1, \ldots, \Pi_k$ be a securely extendable protection domain with a shared setup $\mathcal{F}_\triangle$. Let $\Pi\langle\Pi_1, \ldots, \Pi_k\rangle$ be as secure as an ideal functionality $\mathcal{F}_0$. Then, $\Pi\langle\Pi_1, \ldots, \Pi_k\rangle, \Pi_1, \ldots, \Pi_k$ is a secure protection domain for compound protocols $\mathbb{P}$ provided that $\Pi\langle\mathcal{F}_1, \ldots, \mathcal{F}_k\rangle$ is a secure protection domain for the set of compound protocols $\mathbb{P}_* = \{\Pi_\mathsf{e}\langle\mathcal{F}_1, \ldots, \mathcal{F}_k\rangle\langle\cdot\rangle : \Pi_\mathsf{e} \in \mathbb{P}\}$.*

Proof. Let us fix a target signature $(\mathbb{E}, \mathbb{P}, \mathcal{F}_0, \ldots, \mathcal{F}_k)$ for the extended domain, and let $\Pi_\mathsf{e}\langle\cdot\rangle \in \mathbb{P}$ be a compound protocol. Let $\mathbb{A}_1$ be the set of adversaries against

$\Pi_{\mathsf{e}}\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle,\Pi_1,\ldots,\Pi_k\rangle$. As we can restructure the compound protocol into observationally equivalent form $\Pi_{**}\langle\Pi_1,\ldots,\Pi_k,\Pi_1,\ldots,\Pi_k\rangle$, we can replace all protocols with ideal implementations and we need to show

$$\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle,\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_0,\ldots,\mathcal{F}_k\rangle\rangle$$

for environments $\mathbb{E}$ and adversaries $\mathbb{A}_2$. By pushing $\mathcal{F}_1,\ldots,\mathcal{F}_k$ into Π_{e} we obtain a compound protocol $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle \in \mathbb{P}^*$ that interfaces only with $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$, $\mathsf{A} \in \mathbb{A}_2$ and $\mathsf{Env}_{\mathsf{o}} \in \mathbb{E}_{\mathsf{o}}$. Indeed, the construction just removes interface boundaries between Π_{e} and $\mathcal{F}_i$, and nothing else changes. By the assumptions $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle$ is a valid compound protocol for $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ and thus over the environments $\mathbb{E}$

$$\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\mathcal{F}_0\rangle\rangle \ .$$

The latter completes the proof as we can push $\mathcal{F}_1,\ldots,\mathcal{F}_k$ out of the compound protocol to obtain $\Pi_{\mathsf{e}}\langle\mathcal{F}_0,\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$. □

In other words, it is sufficient to analyse the security of $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ against adversaries $\mathbb{A}_2$ implicitly defined by the definition of securely extendable protection domains with respect to compound protocols $\mathbb{P}_*$ and environments $\mathbb{E}$.

2.4.6. Restrictions to Environments and Adversaries

It is usually impossible to prove $\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\Pi_1,\ldots,\Pi_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\mathcal{F}\rangle$ for canonical $\mathcal{F}$ if Π_{e} leaks the joint state to the environment $\mathsf{Env}_{\mathsf{o}}$. In particular, no shares can pass the service interface between $\mathsf{Env}_{\mathsf{o}}$ and Π_{e} nor can Π_{e} send outputs that depend on the private setup parameters. We can force this constraint structurally by including dedicated protocols into the protection domain which allow parties to securely share and reconstruct values, i.e., they are are equivalent to securely applying $\mathcal{S}_u$ and $\mathcal{R}_u$ to inputs and outputs.

Let us consider the inner environment Π_{e} that shares the state with Π. Note that the inner environment Π_{e} can carry out the same actions as the protocol Π, thus we prefer this notation to clearly distinguish it from Env that represents also the rest of the world and all actions possible there. In the following, Env compatible with Π means $\mathsf{Env}_{\mathsf{o}}\langle\Pi_{\mathsf{e}}\rangle$ that is the full environment against the protocol Π. Due to nesting, the same physical entity is represented by different machines in different collections, such as $\mathcal{P}_i^*$ and $\mathcal{P}_i$ in Figure 5. In principle, a protocol participant can communicate with many machines from the inner environment Π_{e}. However, simple physical considerations suggest that each protocol node $\mathcal{P}_i$ should have only one parent node $\mathcal{P}_i^*$ that provides inputs to $\mathcal{P}_i$ and receives its outputs. By duplicating protocols we can always reach a configuration where $\mathcal{P}_i$ communicates only with a single parent node $\mathcal{P}_i^*$. We consider only such inner environments $\mathbb{P}$ that satisfy this restriction. In addition, we assume that the state of $\mathcal{P}_i$ is such that it allows to restore all computations that have occurred before it was corrupted (e.g., inputs and random choices are stored).

Definition 10 (Generic adversary)**.** *We call the class of adversaries against a protocol* Π *and the inner environment* Π_{e} *generic if the only restrictions on the adversary are the port compatibility with the protocol* Π*, the inner environment* Π_{e} *and the environment* $\mathsf{Env}_{\mathsf{o}}$*.*

A real-world adversary corrupts physical hardware or administrators, thus it is natural to assume that it will corrupt all machines hosted by it. Therefore, we assume that either all machines representing a party are corrupted or none are. Sometimes many logical protocol participants are represented by one physical party and in such cases it is also reasonable to assume that they are also all corrupted together.

Definition 11 (Coherent adversary)**.** *A coherent adversary always corrupts* $\mathcal{P}_i$ *and* $\mathcal{P}_i^*$ *simultaneously, i.e.,* A *sends a corruption call to* $\mathcal{P}_i$ *immediately after* $\mathcal{P}_i^*$ *responds to a corruption call, or vice versa.*

Lemma 1. *Any generic adversary can be extended to a coherent adversary provided that adversary structures for* Π_{e} *and* Π *are compatible.*

3. Results

In this section, we show the required transformations from the hybrid protocol to the abstract execution model. We show that any hybrid protocol satisfying some conditions can be translated to the abstract setting and vice versa. Therefore, we fulfil the requirements of Theorem 2 by defining the ψ, ϕ_1, ϕ_2 and their semi-inverses and showing that the protocol designer only has to define ρ in the abstract execution model. We consider the protocol Π as a subprotocol of Π_{e} representing the rest of the computations in the protection domain and the outer environment Env using and controlling the protection domain. The combination of Π_{e} and Env forms the class $\mathbb{E}_{\Pi}$ of environments against this protocol Π.

We define the transformation to the abstract model in small steps in order to make it clear where the different conditions come from and to make it easier to argue the correctness of these transformations. In Section 3.1, we show that it is sufficient to limit adversarial capabilities. In Section 3.2, we modify the protocol description and adversary to use a shared memory and limit adversarial actions to only modifications of real protocol messages. In Section 3.3, we show how to remove share representation and only give the adversary the access allowed by the limited control property of the storage domain. Finally, in Section 3.4 we arrive at the abstract execution model.

3.1. Minimal Requirements to Message Scheduling

In the following, we show that under certain natural restrictions about the protocol Π the adversaries ability to influence the execution is rather limited. All attacks can be accomplished by only modifying the state of corrupted parties while keeping them running. This is the first substantial step towards abstract model, as these attacks preserve the structure of computations. In term of the soundness theorem we define a universal construction ϕ_{lazy} that achieves

$$\forall \Pi \in \mathbb{P} : \ \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \ \forall \mathsf{A} \in \mathbb{A}_{\Pi,\mathsf{Env}} : \quad \mathsf{Env}\langle \Pi, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \Pi, \phi_{lazy}(\mathsf{A}) \rangle \tag{7}$$

where $\mathbb{P}$ is the set of protocols that use $\mathcal{F}_0, \ldots, \mathcal{F}_k$ and $\mathbb{E}_{\Pi}$ is the set of environments where the protocol Π is intended to run, meaning they contain the outer environment Env and the inner environment Π_{e}. The main result in shown in Theorem 4.

3.1.1. Basics of Protocol Execution

The formal description of a participant of Π is quite complicated, as it must be able to execute several instances of the protocol in parallel and correctly handle corruption queries from the adversary A. To simplify matters, we represent the participant $\mathcal{P}_i$ as a collection of two machines $\mathcal{I}_i$ and $\mathcal{Z}_i$, where $\mathcal{I}_i$ interprets the original protocol without modifications and $\mathcal{Z}_i$ models the effects of corruption by switching communication between the adversary A and other machines.

The corresponding collection is depicted in Figure 6 together with the numbering of port pairs and machine names connecting to them. The zeroth port pair between $\mathcal{I}_i$ and $\mathcal{Z}_i$ is for communicating with the adversary. Next k port pairs between $\mathcal{I}_i$ and $\mathcal{Z}_i$ are for calling out subprotocols. The last port pair between $\mathcal{Z}_i$ and $\mathcal{I}_i$ is for communicating with the inner environment Π_{e}. All these ports are directly matched with port-pairs between $\mathcal{Z}_i$ and the corresponding external machines. Note that all our buffers come in pairs, thus we also use the shorthand b_p^+, b_p^- to specify the pair, where b_p^+ is outgoing from $\mathcal{P}_i$ and b_p^- is incoming to the party. This notation can be enhanced with additional indices if it is important to consider many parties or ideal functionalities at once.

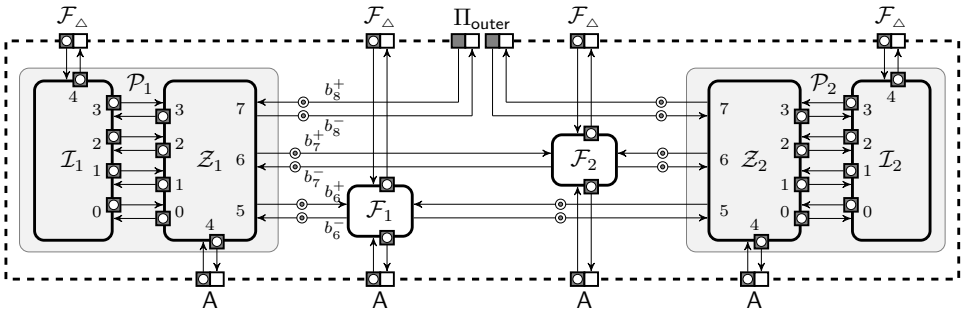


Figure 6. Internal structure of a two-party protocol Π calling out two protocols $\mathcal{F}_1$ and $\mathcal{F}_2$.

When $\mathcal{Z}_i$ is in the honest state it mediates communication between the matching ports. Every time $\mathcal{Z}_i$ receives a message, it writes the respective message and also clocks the messages to $\mathcal{I}_i$. When receiving a message from $\mathcal{I}_i$ it also gives control back to $\mathcal{I}_i$ after writing the message to the output port. When $\mathcal{Z}_i$ is corrupted, then A can order $\mathcal{Z}_i$ to write a message to any of its output ports and all messages arriving to the input ports of $\mathcal{Z}_i$ are forwarded to A. Additionally, A can issue a special REVEAL message to $\mathcal{I}_i$ to receive the internal state of $\mathcal{I}_i$. Each message sent between $\mathcal{P}_i$ and $\mathcal{F}_p$ is a triple (t_1, t_2, m) where t_1 specifies the instance of the protocol Π and t_2 the instance of the sub-protocol called by $\mathcal{I}_i$. Similarly, a message sent between $\mathcal{P}_i$ and Π_e is triple (t_1, t_2, m) where t_2 specifies the instance of the protocol Π and t_1 the instance of Π_e calling it.

3.1.2. Tight Message Scheduling

The structure of subprotocols $\Pi_1, \ldots, \Pi_k$ determines what the adversary can do with ingoing and outgoing messages. For example, consider a protocol where $\mathcal{P}_i$ submits several inputs $x_1, x_2, \ldots, x_\ell$ without any reply from $\mathcal{F}_p$. Then, the adversary can trivially reorder inputs by delaying messages. Although sequence numbers can be added to fix the intended order of messages, we still cannot guarantee the arrival of x_i. Only a reply to $\mathcal{P}_i$ after the ℓ-th input stops the flow of inputs which the adversary can reorder. Therefore, sending $x_1, \ldots, x_\ell$ one-by-one has no theoretical benefit over a single message $(x_1, \ldots, x_\ell)$. In practice one could still stream these as individual messages but it does not affect the theoretical communication model. The same argument invalidates the utility of piecewise release of outputs. As a result, neither $\mathcal{P}_i$ nor $\mathcal{F}_p$ should send a new message before they get a reply from their respondent. A reply in a protocol fixes time-point in a protocol after which an input or an output is committed and can not be changed.

A protocol might include a party without consent by sending outputs $\mathcal{P}_i$ before it has sent inputs. In practical protocol constructions, we always know when $\mathcal{P}_i$ is going to participate in a protocol, and thus we can always assume that all parties provide an input before receiving any outputs. The following definition summarises minimal requirements for protocol constructions to be secure against network delays. Communication patterns of such protocols may still depend on inputs or outputs. For example, a party can submit an unbounded number of inputs that depend on previous replies.

Definition 12 (Tight message scheduling)**.** *An ideal functionality $\mathcal{F}_p$ has a tight message scheduling if $\mathcal{P}_i$ and $\mathcal{F}_p$ cannot send two consecutive messages to their recipients. Additionally, $\mathcal{P}_i$ must send the first message before receiving anything from $\mathcal{F}_p$ and both $\mathcal{F}_p$ and $\mathcal{P}_i$ must know when the other stops sending messages for a given protocol instance.*

3.1.3. Robustness against Malformed Inputs

As a corrupted party can arbitrarily deviate from a protocol specification, we must relate its messages with the state progression in the honest protocol run. For that we show that a corrupted party can always run the interpreter honestly and deliver all messages from ideal functionalities to the interpreter instantly.

Definition 13 (Semi-simplistic adversary)**.** *An adversary is semi-simplistic if it fulfils the following conditions for corrupted $\mathcal{P}_i$.*

(a) The adversary clocks any outgoing buffer b_p^+ and any incoming buffer b_p^- connected to an honest party only when all incoming buffers b_p^- connected to corrupted parties are empty.
(b) Upon receiving a message from $\mathcal{Z}_i$ that comes from $\Pi_e, \mathcal{F}_1, \ldots, \mathcal{F}_k$, the adversary immediately orders $\mathcal{Z}_i$ to forward it to $\mathcal{I}_i$ without changes.
(c) The adversary can send arbitrary messages to $\Pi_e, \mathcal{F}_1, \ldots, \mathcal{F}_k$ on behalf of $\mathcal{P}_i$.
(d) The adversary can fetch the state of $\mathcal{I}_i$.
(e) The adversary gives no other orders to $\mathcal{Z}_i$.

Conditions (a)–(b) formalise instant message delivery which preserves the order of messages: if $\mathcal{P}_i$ receives m_1 before m_2 then $\mathcal{I}_i$ must receive m_1 before m_2. Furthermore, corrupted parties receive messages earlier than honest parties. Conditions (c)–(e) guarantee that the adversary can not directly manipulate the state of $\mathcal{I}_i$, thus $\mathcal{I}_i$ is running honestly.

Lemma 2. *For any adversary, A, there exists an equivalent semi-simplistic adversary A^*. The overhead in computational complexity can be arbitrary.*

Proof. ADVERSARIAL BEHAVIOUR. When a party, $\mathcal{P}_i$, is not corrupted A^* just does whatever A does. Whenever A corrupts $\mathcal{P}_i$, the new adversary A^* also corrupts $\mathcal{P}_i$ and sends the REVEAL message to get the internal state of $\mathcal{I}_i$. After that, A^* can internally simulate the interpreter by initialising it with the state. Let $\mathcal{I}_i^*$ denote the corresponding virtual interpreter. Whenever A^* gets an incoming message destined to the interpreter, it forwards it to $\mathcal{I}_i$ without changes. If $\mathcal{I}_i$ sends back a reply, A^* deletes it. If $\mathcal{I}_i$ does not send a reply, then control still goes to A^* when $\mathcal{I}_i$ stops. After that, A^* passes the original message to A. Whenever A wants to send a message to $\mathcal{I}_i$, A^* sends it to $\mathcal{I}_i^*$. If $\mathcal{I}_i^*$ sends back a reply, A^* forwards it to A. If A wants to send a message to other parties A^* forwards it to $\mathcal{Z}_i$. As a result, all incoming messages reach $\mathcal{I}_i$ without changes and right after being received by $\mathcal{Z}_i$ while the A^* sends out exactly the same messages as A.

MODIFIED CLOCKING. To guarantee that A^* can always empty a buffer b_p^-, we must do another modification. First, the adversary A^* can keep the buffer b_p^- empty by clocking it immediately when $\mathcal{F}_p$ writes to it for corrupted $\mathcal{P}_i$. This fulfils the conditions (a)–(b). The timing of $\mathcal{I}_i$ might change but we only need to preserve the behaviour of A. For that, A^* stores the messages and internally simulates buffers b_p^- to A.

COMPLEXITY. The overhead in the computational complexity consists of copying and running the interpreter. The state of the $\mathcal{I}_i$ must be copied and its further actions as $\mathcal{I}_i$ must be simulated. Simulated interpretation comes with at most a polynomial slowdown as the state is of polynomial size. As incoming messages of $\mathcal{I}_i$ are potentially altered by the actions of A, the interpreter $\mathcal{I}_i$ is not guaranteed to terminate. Therefore, we can give no overhead bound in this construction. Nevertheless, A^* is semi-simplistic. □

Adversarial actions may lead to unexpected inputs, thus the interpreter $\mathcal{I}_i$ is not guaranteed to terminate. Overall, there are three possibilities to overload the interpreter. First, they may get unexpected messages from ideal functionalities or the environment. The interpreter should be able to ignore such messages. Second, the adversary might be able to trick the environment or an ideal functionality to send overly long inputs to the interpreter. These attacks are harmless as long as the interpreter knows the maximal input length and ignores the rest. Third, the adversary might trick the interpreter to do expensive local computations. This is a serious concern unless the amount of local computations is bounded.

Definition 14 (Robustness against malformed inputs)**.** *A protocol is robust against malformed inputs if the running time of the interpreter is polynomial for all semi-simplistic adversaries.*

Corollary 1. *If a protocol is robust against malformed inputs, then semi-simplistic and generic adversaries are equivalent to each other.*

Proof. The robustness guarantees that the construction introduced in Lemma 2 has a polynomial overhead. Each time A^* invokes $\mathcal{I}_i$, it is guaranteed to stop and pass the control back to A^*. As the number of times A^* invokes $\mathcal{I}_i$ is bounded by the running time of A, the total running time of $\mathcal{I}_i$ can be only polynomial times bigger than the running time of A. In addition, any semi-simplistic adversary is also a generic adversary. □

3.1.4. Security against Rushed Execution

A semi-simplistic adversary may create messages that are dropped by recipients as they are not ready to process them. Let a tuple $(i, p, t_1, t_2 \rightarrow)$ denote the event where $\mathcal{I}_i$ writes a message (t_1, t_2, m) to the output port p, and let $(i, p, t_1, t_2, \leftarrow)$ denote the event where the recipient $\mathcal{F}_p$ writes a reply (t_1, t_2, m) to its output port. Note that for semi-simplistic adversaries the interpreters are always running honestly.

Definition 15 (Input and output signature)**.** *Let the input signature for a particular round of computations in canonical ideal functionality (Definition 5) $\mathcal{F}_p$ be the set of parties that must provide inputs before $\mathcal{T}_\mathcal{R}$ forwards recovered inputs to $\mathcal{F}_p^*$, and let the output signature be the set of parties that receive output shares from $\mathcal{T}_\mathcal{S}$.*

Definition 16. *We say that a round of computation is rushed if the ideal functionality $\mathcal{F}_p$ executes the computation before some interpreter $\mathcal{I}_i$ in the input signature has computed its input to this round. A protocol is secure against rushed execution for the set of environments $\mathbb{E}$ if no semi-simplistic adversary from the class of adversaries $\mathbb{A}$ can rush a round of computation.*

The explicit limitations on the set of adversaries is necessary, as there may be a gap between protocols that unbounded adversaries can rush vs. polynomial time adversaries. Usually, one considers only polynomial-time adversaries. Security against rushing allows us to avoid situations where semi-simplistic adversaries desynchronise a protocol by sending out messages (t_1, t_2, m) way earlier than the event $(i, p, t_1, t_2, \rightarrow)$ takes place.

Definition 17 (Lazy adversary)**.** *A semi-simplistic adversary is lazy if it always waits for $(i, p, t_1, t_2, \rightarrow)$ signal from $\mathcal{I}_i$ to clock a message (t_1, t_2, m) out of the buffer b_p^+ and it always clocks at most one message with right tags per signal.*

Lemma 3. *Assume that ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ and protocol Π have tight scheduling. If a protocol Π is secure against rushed execution, then semi-simplistic adversary can be converted to equivalent lazy semi-simplistic adversary.*

Proof. Assume that all ideal functionalities have unlimited buffering. Let A^* be a modified adversary that internally runs the original adversary A and monitors what messages are written to outgoing buffers and when they are clocked. This allows A^* to catch all events where A tries to clock a message (t_1, t_2, m) out of a buffer b_p^+ before the interpreter $\mathcal{I}_i$ has produced the event $(i, p, t_1, t_2, \rightarrow)$. For these events, A^* catches the clocking signal and forwards it as soon as the event $(i, p, t_1, t_2, \rightarrow)$ occurs. Note that thanks to tight scheduling the tags t_1, t_2 and the port uniquely determine the buffer and the message of the protocol and it is clear which message can be clocked.

Such delays have no effect on execution. If nobody submits its input after the event $(i, p, t_1, t_2, \rightarrow)$, then we have a prohibited a rushing event. As we have security against rushing, $\mathcal{T}_\mathcal{R}$ still waits for inputs when A^* clocks (t_1, t_2, m). If A^* never clocks a message, then the event $(i, p, t_1, t_2, \rightarrow)$ never occurs, and thus security against rushing guarantees that the corresponding round of computation is never completed or this party was not in the input signature. Consequently, the overall execution does not change.

In the general case, $\mathcal{F}_p$ might keep (t_1, t_2, m) after a delay while its dropped in the original run. The reverse is not possible as no messages can take the place of delayed (t_1, t_2, m) because tightness ensures $\mathcal{P}_i$ has only one outstanding message for instance t_2 of $\mathcal{F}_p$. As A^* knows all messages sent and received by $\mathcal{F}_p$, tight message scheduling guarantees that A^* knows which rounds of computations are completed or pending. Therefore, A^* can efficiently compute whether a message will be dropped or not. Thus, we can always convert A to the lazy adversary that never clocks a message that is dropped.

Note that the the communication between Π and Π_e is similar, except that Π is in the role of the $\mathcal{F}_p$ (with tight scheduling), and we can only convert the adversary to lazy clocking if Π_e is secure against rushing. Otherwise, for a coherent adversary, we know that adversary can only create messages to the buffers between the corrupted party in Π and Π_e. Therefore, delays can affect only corrupted buffers and without lessening of generality we can assume that instead the adversary could perform the follow up actions of the corrupted party in Π_e without really clocking this message. □

Theorem 4. *If a protocol is robust against malformed inputs and is secure against rushed execution, then lazy semi-simplistic and generic adversaries are equivalent to each other.*

Proof. From Corollary 1, we know that generic adversaries are equivalent with semi-simplistic adversaries. Lemma 3 proves that any semi-simplistic adversary can be transformed to a lazy semi-simplistic adversary. In turn, each lazy semi-simplistic adversary is also a semi-simplistic adversary. □

Theorem 5 (Characterisation of rushing)**.** *A semi-simplistic adversary can rush a round of computation π for a functionality with tight scheduling only if one of the following holds for a corrupted $\mathcal{P}_i$ in the input signature.*

(a) The round π is not in the program code of the interpreter $\mathcal{I}_i$.
(b) The interpreter $\mathcal{I}_i$ needs an input from Π_e to submit an input to π.
(c) The interpreter $\mathcal{I}_i$ needs an output from a round of computation π' to submit an input to π and π' is executed concurrently or after π.

Proof. Let a party $\mathcal{P}_i$ be in the input signature of a rushed round of computation π such that its interpreter $\mathcal{I}_i$ computes the input for the round after π is completed and let b_p^+ be the corresponding outgoing buffer. As $\mathcal{T}_\mathcal{R}$ does not proceed without the input from $\mathcal{P}_i$, the input had to be present in b_p^+. For honest parties, only $\mathcal{I}_i$ can create such inputs. Consequently, $\mathcal{P}_i$ must have been corrupted before the input to π was clocked.

There are two options why the interpreter $\mathcal{I}_i$ cannot produce input before this clocking event. First, the program code of $\mathcal{I}_i$ never computes inputs for π, i.e., π is not scheduled. Second, $\mathcal{I}_i$ must be waiting for a message m to arrive through some incoming buffer b_q^- before it can compute the input to π. For semi-simplistic adversaries, the buffer b_q^- is always emptied before clocking of b_p^+. Thus, the input m must be created by another round of computation π' or inputs from Π_e that is still incomplete. □

The possibility of (b) can be eliminated by the design of Π or Π_e. We can include Byzantine agreements in Π to make sure that no honest party starts π before necessary inputs are received from Π_e, or we can restrict Π_e to give inputs in a manner that all parties who execute π receive inputs in one go. Let $\mathcal{F}_0$ be the canonical ideal functionality for Π. Then, we can analyse if the adversary can rush $\mathcal{F}_0$ in $\mathsf{Env}\langle\mathcal{F}_\triangle\langle\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_k\rangle\rangle$ using Theorem 5. If rushing is impossible, then all corrupted parties in round π are guaranteed to get input from Π_e before executing π and we need to exclude only possibilities of (a) and (c).

The majority of all multiparty computation protocols operate with values that are secret shared among all participants. Consequently, no computation round can be rushed if the protocol consists of sequential execution of subprotocols and some party is honest. In particular, if a protocol description is symmetric for all parties and input functionalities are

also symmetric, then dependencies between rounds of computation are the same for all parties and no round is computed without the inputs from honest parties.

3.2. Shared Memory and Simplistic Adversaries

Lazy semi-simplistic adversaries are quite restricted, as they clock messages to $\mathcal{F}_p$ according to protocol specification. Still, they can send out more messages than are finally clocked. We define a shared memory model where such attacks can be carried out by modifying a limited set of memory locations and call this the simplistic adversary. In more formal terms, we define a $\bowtie$-operator that acts on protocols and their components together with a universal construction $\phi_{\bowtie}$ and its semi-inverse $\phi^*_{\bowtie}$ that achieves

$$\forall \Pi \in \mathbb{P} : \ \forall \mathsf{Env} \in \mathbb{E}_\Pi : \ \forall \mathsf{A} \in \mathbb{A}^{lazy}_{\Pi,\mathsf{Env}} : \quad \mathsf{Env}\langle \Pi, \mathsf{A}\rangle \equiv \mathsf{Env}\langle \Pi^{\bowtie}, \phi_{\bowtie}(\mathsf{A})\rangle \tag{8}$$

$$\forall \Pi \in \mathbb{P} : \ \forall \mathsf{Env} \in \mathbb{E}_\Pi : \ \forall \mathsf{A} \in \mathbb{A}^{\bowtie}_{\Pi,\mathsf{Env}} : \quad \mathsf{Env}\langle \Pi^{\bowtie}, \mathsf{A}^{\bowtie}\rangle \equiv \mathsf{Env}\langle \Pi, \phi^*_{\bowtie}(\mathsf{A}^{\bowtie})\rangle \tag{9}$$

where $\mathbb{P}$ is the set of protocols satisfying the above restrictions which use $\mathcal{F}_0, \ldots, \mathcal{F}_k$ in black-box way and $\mathbb{E}_\Pi$ is the set of compatible environments. The first part constructing the memory model and simplistic adversary are shown in Theorems 6 and 7 through a $\diamond$-operator. This is extended to memory alignment in Theorem 8.

3.2.1. Interpreter Specification

The changes to the memory model alter the interpreters as well as the communication patterns. The interpreter $\mathcal{I}_i$ in Π is a universal random access machine with two special communication instructions DMACALL and SEND. All instances of $\mathcal{I}_i$ share a program $\mathfrak{p}$. The internal state of $\mathcal{I}_i$ is a three-dimensional array $\mathfrak{s}[t, \delta, \ell]$ where t specifies a protocol instance, δ a storage domain and ℓ a memory location. A protocol instance t can access only its slice $\mathfrak{s}[t, \cdot, \cdot]$. Initially, the state $\mathfrak{s}$ is empty and there are no active protocol instances. Π_{e} launches a new protocol instance by sending a special triple denoted as $\textsc{Init}(t_1, t_2, \delta, m)$ to $\mathcal{I}_i$. Upon initialisation, $\mathcal{I}_i$ launches a new protocol instance t_2 with the input m of type δ and stores t_1 as the instance of the parent protocol.

An instruction $\textsc{DmaCall}(t, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ initiates a query–response round where the vector $\boldsymbol{\alpha} = ((\delta_1, \ell_1), \ldots, (\delta_u, \ell_u))$ specifies the memory locations to be assembled into a tuple m and $\boldsymbol{\beta} = ((\delta'_1, \ell'_1), \ldots, (\delta'_v, \ell'_v))$ specifies to which locations the elements of a response tuple m' are stored. The respondent and its protocol instance is fixed by the port number p and the instance tag t. The instruction is not complete until the response comes. An instruction $\textsc{Send}(t, p, \boldsymbol{\alpha})$ initiates analogous communication without response.

Tags t_1 and t_2 encode the instance of a caller and a callee in all triplets (t_1, t_2, m). Therefore, the outcome of DMACALL and SEND instructions depends on the port p. As ports $1, \ldots, k$ are meant for subprotocols, outgoing messages must be in the form (t_1, t_2, m) where t_1 is the current protocol instance and t_2 fixes the instance of a subprotocol.

The remaining instructions formalise a type safe memory manipulation and conditional jumps. Conditional jumps in the program can occur only on public or local variables. A special local storage domain is used for storing the state of the instruction interpreter including program counters and memory locations of incomplete DMACALL instructions. This state can be quite complex for interpreter types that concurrently execute several protocol instructions.

Definition 18. *A program* $\mathfrak{p}$ *is well-formed if the following holds.*

(a) *Each memory location* $\mathfrak{s}[t, \delta, \ell]$ *can be assigned only once.*
(b) *No instruction can read a memory location before it is initialised.*
(c) *A new message with tag* (t_1, t_2) *is never written to the output port* p *to* Π_{e} *before reading a message with tag* (t_1, t_2) *from the input port* p *from* Π_{e}.
(d) *For instructions* $\textsc{DmaCall}(t, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ *and* $\textsc{Send}(t, p, \boldsymbol{\alpha})$, *no other program instruction can read memory locations in the vector* $\boldsymbol{\alpha}$ *and write the memory location* $\boldsymbol{\beta}$.

Note that after the location $\boldsymbol{\alpha}$ has been used as an input to some ideal functionality, only $\mathcal{F}$ and A are allowed to read it and only $\mathcal{F}$ writes the return location $\boldsymbol{\beta}$. For conceptual clarity and brevity, we consider only well-formed programs. Well-formedness is largely a property of the concrete implementation of the protocol, hence we may also address it as the protocol with well-formed implementation.

3.2.2. Shared Memory Model for Communication

Replacing message passing with shared memory allows us to merge individual states of interpreters into consolidated memory and limits the adversary to only work with messages sent by $\mathcal{I}_i$. Let $\mathcal{I}_i^\diamond$ be a stateless interpreter, $\mathcal{F}_{\text{io}}^\diamond$ is a collection of parts of interpreters that deal with protocol inputs and outputs, and $\mathcal{M}_i^\diamond$ is a memory machine for storing the internal state of $\mathcal{I}_i$. We introduce $\mathcal{F}_{\text{io}}^\diamond$ to make some follow-up steps of the transformation clearer later on. $\mathcal{F}_{\text{io}}^\diamond$ contains the input–output modules of each interpreter in the protocol. When $\mathcal{F}_{\text{io}}^\diamond$ gives an input to $\mathcal{I}_i$ then it writes it to memory and clocks the notification to $\mathcal{I}_i$. When $\mathcal{F}_{\text{io}}^\diamond$ receives output from $\mathcal{I}_i^\diamond$ then it reads it from memory, writes it to buffer to Π_{e} and gives control back to $\mathcal{I}_i^\diamond$. Let $\mathcal{F}_1^\diamond, \ldots, \mathcal{F}_k^\diamond$ be modified ideal functionalities with access to the shared memory $\mathcal{M}_1^\diamond, \ldots, \mathcal{M}_k^\diamond$ as depicted in Figure 7.

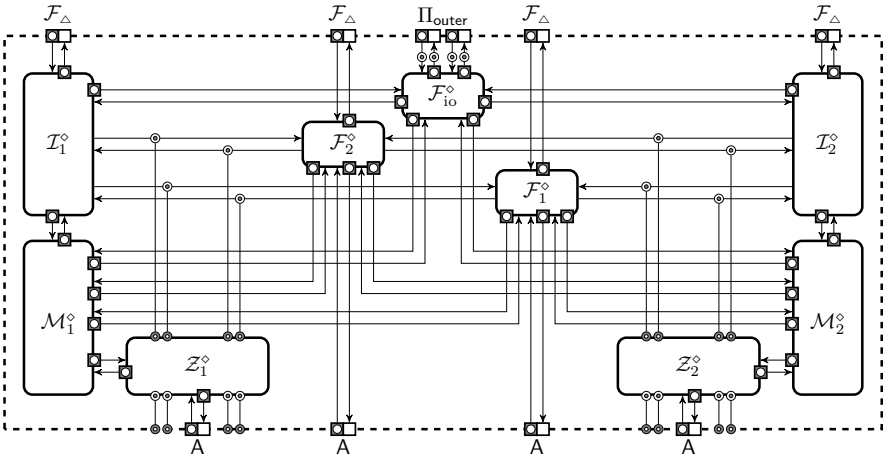


Figure 7. Decomposed interpreters for protocol $\Pi^\diamond$ calling out protocols $\mathcal{F}_1^\diamond$ and $\mathcal{F}_2^\diamond$.

The memory $\mathcal{M}_i^\diamond$ stores the state of $\mathcal{I}_i$ as a three-dimensional array $\mathfrak{s}$. Each buffer pair can be used to access and modify $\mathfrak{s}$. Given an input $\textsc{Fetch}(t, \delta, \ell)$, $\mathcal{M}_i^\diamond$ returns $(t, \mathfrak{s}[t, \delta, \ell])$. Given an input $\textsc{Set}(t, \delta, \ell, m)$, $\mathcal{M}_i^\diamond$ updates the state by setting $\mathfrak{s}[t, \delta, \ell] = m$ and replies $\textsc{Ok}(t)$. For convenience, there are also commands for block reads and writes. The communication between $\mathcal{I}_i^\diamond$ and $\mathcal{F}_p^\diamond$ goes through the exchange of memory locations. The interpreter $\mathcal{I}_i^\diamond$ translates $\textsc{DmaCall}(t_2, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ into a message $(t_1, t_2, \boldsymbol{\alpha}, \boldsymbol{\beta})$, where t_1 is the caller instance, $\boldsymbol{\alpha}$ specifies the locations of message components and $\boldsymbol{\beta}$ locations for the reply. Send instructions are translated into triples $(t_1, t_2, \boldsymbol{\alpha})$. A recipient $\mathcal{F}_p^\diamond$ or $\mathcal{F}_{\text{io}}^\diamond$ queries $\mathcal{M}_i^\diamond$ to assemble the message m and processes the resulting triple (t_1, t_2, m) as in the original setting. When a reply $(m'_1, \ldots, m'_u)$ is generated then all elements m'_j are stored to memory locations $\mathfrak{s}[t_1, \delta'_j, \ell'_j]$ and a special message (t_1, t_2, ϵ) is sent back. Upon receiving $\textsc{Init}(t_1, t_2, \delta, m)$ for $\mathcal{P}_i$ and protection domain δ, $\mathcal{F}_{\text{io}}^\diamond$ writes components of an input m to some default memory locations $\mathfrak{s}[t_\star, \delta_j, \ell_j]$ and sends $\textsc{Init}(t_1, t_2, \delta, \boldsymbol{\alpha})$ to $\mathcal{I}_i^\diamond$.

The machine $\mathcal{Z}_i^\diamond$ simulates the original execution of the protocol Π to A. For that, $\mathcal{Z}_i^\diamond$ must simulate a missing machine $\mathcal{Z}_i$ and buffers b_p^-, b_p^+ between $\mathcal{Z}_i$ and $\mathcal{F}_p$. Let the buffers connecting $\mathcal{I}_i^\diamond$ to $\mathcal{F}_p^\diamond$ be c_p^+ and c_p^- analogously to b_p^-, b_p^+ between $\mathcal{Z}_i$ and $\mathcal{F}_p$. $\mathcal{Z}_i^\diamond$ can communicate with $\mathcal{M}_i^\diamond$ and clock buffers c_p^+ and c_p^- but can not access $\mathcal{M}_i^\diamond$ before adversary A issues a corruption call. $\mathcal{Z}_i^\diamond$ modifies only the memory locations $\boldsymbol{\alpha}$ of incomplete $\textsc{DmaCall}(t, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ and $\textsc{Send}(t, p, \boldsymbol{\alpha})$ calls. Define $\Pi^\diamond$ as an extended collection consisting of machines $\mathcal{I}_1^\diamond, \ldots, \mathcal{I}_n^\diamond, \mathcal{M}_1^\diamond, \ldots, \mathcal{M}_n^\diamond, \mathcal{F}_0^\diamond, \ldots, \mathcal{F}_k^\diamond, \mathcal{F}_{\text{io}}^\diamond$ together with

all buffers attached to the machines. Let the corresponding adversarial construction $\phi^{\diamond}(\mathsf{A})$ be defined as a reduced collection consisting of $\mathsf{A}, \mathcal{Z}_1^{\diamond}, \ldots \mathcal{Z}_n^{\diamond}$.

Theorem 6. *Let* Π *be the protocol with a well-formed implementation and let* $\mathbb{E}_{\Pi}$ *be the set of compatible environments. Then,*

$$\forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \quad \forall \mathsf{A} \in \mathbb{A}^{lazy}_{\Pi,\mathsf{Env}} : \qquad \mathsf{Env}\langle \Pi, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \Pi^{\diamond}, \mathsf{A}^{\diamond} \rangle$$

where and $\mathbb{A}^{lazy}_{\Pi,\mathsf{Env}}$ *is the set of compatible lazy semi-simplistic adversaries.*

Proof sketch. Let us define another extended collection $\hat{\Pi}^{\diamond}$ that consists of machines $\mathcal{I}_1^{\diamond}, \ldots, \mathcal{I}_n^{\diamond}, \mathcal{M}_1^{\diamond}, \ldots, \mathcal{M}_n^{\diamond}, \mathcal{Z}_1^{\diamond}, \ldots \mathcal{Z}_n^{\diamond}, \mathcal{F}_0^{\diamond}, \ldots, \mathcal{F}_k^{\diamond}, \mathcal{F}_{\text{io}}^{\diamond}$ together with all attached buffers. We get a trivial equivalence $\mathsf{Env}\langle \hat{\Pi}^{\diamond}, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \Pi^{\diamond}, \mathsf{A}^{\diamond} \rangle$ for any A and Env. Therefore, it is sufficient to prove that $\mathsf{Env}\langle \Pi, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \hat{\Pi}^{\diamond}, \mathsf{A} \rangle$ for any Env and A that meet the restrictions. Although collections Π and $\hat{\Pi}^{\diamond}$ are quite different there is a natural matching between machines and their internal states. Initially, the internal states of $\mathcal{F}_p$ and $\mathcal{F}_p^{\diamond}$ coincide. The same is true for the interpreters $\mathcal{I}_i$ and $\mathcal{I}_i^{\diamond}$ although their internal state $\mathfrak{s}$ is stored in different locations. Therefore, we can run the standard bisimulation argument and show that the actions of A or Env cannot diverge execution to non-equivalent states.

First, define $\mathcal{Z}_i^{\diamond}$ that achieves the goal by ignoring memory access restrictions. As $\mathcal{M}_i^{\diamond}$ contains the entire state of $\mathcal{I}_i^{\diamond}$, $\mathcal{Z}_i^{\diamond}$ can internally replicate all computations of $\mathcal{I}_i^{\diamond}$ and fetch all messages sent by $\mathcal{F}_p$ from $\mathcal{M}_i^{\diamond}$. Therefore, $\mathcal{Z}_i^{\diamond}$ can perfectly simulate $\mathcal{Z}_i$ and the buffers b_p^+ and b_p^- provided that states of $\mathcal{I}_i$ and $\mathcal{I}_i^{\diamond}$ and $\mathcal{F}_p$ and $\mathcal{F}_p^{\diamond}$ have not diverged yet. However, note that only the messages from and responses to corrupted parties are required to be accessed from the memory, thus the same $\mathcal{Z}_i^{\diamond}$ can easily also satisfy memory access restrictions.

As A is lazy semi-simplistic, we know that $\mathcal{I}_i$ creates a message before $\mathcal{F}_p$ reads the corresponding input. Therefore, $\mathcal{I}_i^{\diamond}$ translates $\textsc{DmaCall}(t_2, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ and $\textsc{Send}(t_2, p, \boldsymbol{\alpha})$ instructions before $\mathcal{F}_p^{\diamond}$ must fetch the corresponding input from the memory. Property (a) of a well-formed program guarantees that $\mathcal{Z}_i^{\diamond}$ can swap the values in the locations $\boldsymbol{\alpha}$ just before it clocks the address tuple to $\mathcal{F}_p^{\diamond}$. Property (d) guarantees that the change does not alter further actions of $\mathcal{I}_i^{\diamond}$. Consequently, we can guarantee that $\mathcal{F}_p$ and $\mathcal{F}_p^{\diamond}$ always get the same inputs and thus the executions of the ideal functionalities in the two worlds give the same results. Therefore, also $\mathcal{I}_i$ and $\mathcal{I}_i^{\diamond}$ get the same results from the ideal functionalities and the simulation done by $\mathcal{Z}_i^{\diamond}$ is perfect. □

To show the full equivalence of the two execution models, we need to define a class of simplistic adversaries $\mathbb{A}^{\diamond}_{\Pi,\mathsf{Env}}$ against $\Pi^{\diamond}$ and Env that are produced by $\phi_{\diamond}$ and then define semi-inverse of $\phi_{\diamond}^*$ with the right properties.

Definition 19 (Simplistic adversary)**.** *An adversary is simplistic if it satisfies the following.*

(a) The adversary clocks any outgoing buffer b_p^+ *and any incoming buffer* b_p^- *to an honest party only when all incoming buffers* b_p^- *to corrupted parties are empty.*

(b) The adversary can modify the state of the corrupted party only in the locations $\boldsymbol{\alpha}$ *of pending* $\textsc{DmaCall}(t, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ *and* $\textsc{Send}(t, p, \boldsymbol{\alpha})$ *instructions. These changes are done before the corresponding tuple is clocked to* $\mathcal{F}_p^{\diamond}$ *and each value is modified at most once.*

Corollary 2. *For any lazy semi-simplistic adversary* A *and for any well-formed implementation of* Π*, the construction* $\phi_{\diamond}(\mathsf{A})$ *is simplistic adversary.*

Proof. The way $\mathcal{Z}_i^{\diamond}$ alters memory just before clocking b_p^+ in the proof of Theorem 6 guarantees that the property (b) of simplistic adversary is satisfied. The clocking rules for $\phi_{\diamond}(\mathsf{A})$ are analogous the semi-simplistic adversary A and are therefore satisfied as $\phi_{\diamond}(\mathsf{A})$ preserves clocking with respect to the matched buffers in the two configuration. □

We specify the semi-inverse through simulator machines $\mathcal{Z}_i^*$ that go between $\mathcal{Z}_i$ and $\mathsf{A}^\diamond$. The simulator translates clocking signals, provides read only access to the state of $\mathcal{I}_i$ and translates memory writes to actual protocol messages. As before, let $\phi_\diamond^*(\mathsf{A}^\diamond)$ be the reduced collection consisting of $\mathsf{A}, \mathcal{Z}_1^*, \dots \mathcal{Z}_n^*$.

Theorem 7. *Let $\Pi^\diamond$ be the protocol with a well-formed implementation and let $\mathbb{E}$ be the set of compatible environments. Then,*

$$\forall \mathsf{Env} \in \mathbb{E}: \quad \forall \mathsf{A}^\diamond \in \mathbb{A}^\diamond_{\Pi,\mathsf{Env}}: \qquad \mathsf{Env}\langle \Pi^\diamond, \mathsf{A}^\diamond \rangle \equiv \mathsf{Env}\langle \Pi, \phi_\diamond^*(\mathsf{A}^\diamond) \rangle$$

where and $\mathbb{A}^\diamond_{\Pi,\mathsf{Env}}$ is the set of simplistic adversaries compatible with the protocol and the environment. The resulting adversary $\phi_\diamond^*(\mathsf{A}^\diamond)$ is lazy and semi-simplistic.

Proof sketch. Let us define another extended collection $\hat{\Pi}$ that consists of machines $\mathcal{I}_1, \dots, \mathcal{I}_n, \mathcal{Z}_1, \dots, \mathcal{Z}_n, \mathcal{Z}_1^*, \dots \mathcal{Z}_n^*, \mathcal{F}_0, \dots, \mathcal{F}_k$ together with all buffers attached to the machines. Then, it is sufficient to prove that the equivalence $\mathsf{Env}\langle \Pi^\diamond, \mathsf{A}^\diamond \rangle \equiv \mathsf{Env}\langle \hat{\Pi}, \mathsf{A}^\diamond \rangle$ holds for any Env and $\mathsf{A}^\diamond$ as $\mathsf{Env}\langle \hat{\Pi}, \mathsf{A}^\diamond \rangle \equiv \mathsf{Env}\langle \Pi, \phi_\diamond^*(\mathsf{A}^\diamond) \rangle$ by construction.

We can use the same natural matching between machines and their internal states as in Theorem 6. Equivalence is obvious when $\mathsf{A}^\diamond$ does not corrupt $\mathcal{M}_i^\diamond$. The machine $\mathcal{Z}_i^*$ just forwards all clocking signals and the equivalence follows from the construction of $\mathcal{I}_i^\diamond$ and $\mathcal{F}_p^\diamond$. To fetch a value $\mathfrak{s}[t, \delta, \ell]$, the machine $\mathcal{Z}_i^*$ sends $\textsc{Reveal}(t, \delta, \ell)$ message to $\mathcal{Z}_i$ and forwards the response. After a first $\textsc{Reveal}$, call $\mathcal{Z}_i$ is corrupted and $\mathcal{Z}_i^*$ will instantly forward the communication from $\mathcal{F}_p$ to $\mathcal{I}_i$.

When $\mathsf{A}^\diamond$ issues memory modification instructions, $\mathcal{Z}_i^*$ can locally store all altered values $\mathfrak{s}[t, \delta_i, \ell_i]$. As the adversary $\mathsf{A}^\diamond$ is simplistic, it alters only memory locations related to $\textsc{DmaCall}(t, p, \boldsymbol{\alpha}, \boldsymbol{\beta})$ and $\textsc{Send}(t, p, \boldsymbol{\alpha})$ instructions. Let tuples (t_1, t_2, m) and $(t_1, t_2, \boldsymbol{\alpha}, \dots)$ denote the outcomes of $\mathcal{I}_i$ and $\mathcal{I}_i^\diamond$ for these instructions. Then, by construction $\mathsf{A}^\diamond$ can clock a tuple $(t_1, t_2, \boldsymbol{\alpha}, \dots)$ only after (t_1, t_2, m) is written to b_p^+. Moreover, $\mathsf{A}^\diamond$ does not change values in the location $\boldsymbol{\alpha}$ after $(t_1, t_2, \boldsymbol{\alpha}, \dots)$ is clocked to $\mathcal{F}_p^\diamond$. By property (d) of a well-formed implementation, these changes alter only the corresponding message m' assembled by $\mathcal{F}_p^\diamond$ and nothing more. Therefore, $\mathcal{Z}_i^*$ must always check whether $\mathsf{A}^\diamond$ has altered any of the locations $\boldsymbol{\alpha}$. If there are no changes, $\mathcal{Z}_i^*$ clocks the message (t_1, t_2, m) to $\mathcal{F}_p$. If there are changes $\mathcal{Z}_i^*$ orders $\mathcal{Z}_i$ to write a message (t_1, t_2, m') to b_p^+. After that $\mathcal{Z}_i^*$ can clock (t_1, t_2, m') and ignore the original message (t_1, t_2, m). As a result, $\mathcal{F}_p$ and $\mathcal{F}_p^\diamond$ get the same message at the same time and equivalence is preserved. As $\mathsf{A}^\diamond$ is simplistic then the clocking and modification rules carry over and the resulting adversary is lazy semi-simplistic adversary. □

3.2.3. Memory Alignment and Protocol Specification

Let a global state $\mathfrak{gs}$ be a four-dimensional array that combines the states of all machines $\mathcal{M}_1^\diamond, \dots, \mathcal{M}_n^\diamond$. More precisely, let $\mathfrak{gs}[t, \delta, \ell, i] = \mathfrak{s}[t, \delta, \ell]$ for parties $\mathcal{P}_i$ in the storage domain δ and $\mathfrak{gs}[t, \delta, \ell, i] = \epsilon$ for parties $\mathcal{P}_i$ outside the domain δ. For brevity, let $\mathfrak{gs}[t, \delta, \ell]$ denote a tuple $(\mathfrak{gs}[t, \delta, \ell, 1], \dots, \mathfrak{gs}[t, \delta, \ell, n])$.

Definition 20 (Memory-alignment)**.** *A well-formed protocol uses an ideal functionality in memory-aligned manner if each individual input is reconstructed from $\mathfrak{gs}[t, \delta, \ell]$ and each output is shared to $\mathfrak{gs}[t, \delta, \ell]$. This restriction must hold regardless of adversarial behaviour.*

Note that memory-aligned usage is not a property of a protocol. It easy to define implementations where memory locations are not aligned in subprotocol calls. However, simple incremental addressing is enough to achieve memory alignment for all ideal functionalities when there are no local operations. Local operations without restrictions can easily break the alignment if some parties carry them out and others do not. However, alignment could be achieved by giving a dedicated memory region to local computation outputs that are not inputs to some ideal functionality.

However, from now we will treat local operations explicitly as external components rather than internal affairs of the interpreter. We represent local operations reading and writing to $\mathcal{M}^{\diamond}$ as fragmented functionalities $\mathcal{G}_1^{\diamond}, \ldots, \mathcal{G}_m^{\diamond}$ where each $\mathcal{G}_q^{\diamond}$ is a collections of machines $\{\mathcal{G}_{q,j}^{\diamond}\}_{j \in \mathcal{J}_q}$ implementing local operations. Formally, we need a new interpreter $\mathcal{I}_i^{\bowtie}$ and memory machine $\mathcal{M}_i^{\bowtie}$ that have additional sender-clocked port pairs for communicating with $\mathcal{G}_1^{\diamond}, \ldots, \mathcal{G}_m^{\diamond}$. To initiate an operation, $\mathcal{I}_i^{\bowtie}$ sends a tuple of locations $(t, \boldsymbol{\alpha}, \boldsymbol{\beta})$ to $\mathcal{G}_q^{\diamond}$ which performs the computation and gives control back $\mathcal{I}_i^{\bowtie}$. As before, $\boldsymbol{\alpha}$ determines the locations of inputs and $\boldsymbol{\beta}$ the locations of outputs in the state of $\mathcal{M}_i^{\bowtie}$.

We add a sender-clocked buffer pair between $\mathcal{G}_{q,i}^{\diamond}$ and $\mathcal{F}_{\triangle}$ as local computations may utilise setup parameters of the respective party $\mathcal{P}_i$. As a result, all local operations can be pushed out from $\mathcal{I}_i^{\diamond}$ to $\mathcal{G}_{q,i}^{\diamond}$ and $\mathcal{I}_i^{\bowtie}$ without changing the overall execution as shown in Figure 8.

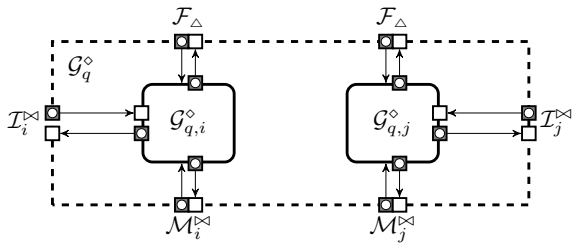


Figure 8. Encapsulation of local computations involving two parties.

Definition 21 (Canonical protocol)**.** *Let $\mathcal{F}_1^{\diamond}, \ldots, \mathcal{F}_k^{\diamond}, \mathcal{F}_{\text{io}}^{\diamond}, \mathcal{G}_1^{\diamond}, \ldots, \mathcal{G}_m^{\diamond}$ denote ideal functionalities used in a well-formed protocol. A protocol specification is in a canonical form if*

(a) all conditional jumps are based on local values,
(b) the remaining local operations are implemented with $\mathcal{G}_1^{\diamond}, \ldots, \mathcal{G}_m^{\diamond}$ and
(c) all ideal functionalities are used in memory-aligned manner.

These conditions must hold regardless of the adversarial behaviour.

Similarly to the previous cases, any well-formed protocol can also be represented in the canonical form. Note that in canonical protocol specification, the interpreter $\mathcal{I}_i^{\bowtie}$ can be isolated from the trusted setup $\mathcal{F}_{\triangle}$ as it runs no computations on its own. A canonical protocol specifications guarantees that $\mathcal{F}_{pd}^{\diamond}$ operates with aligned memory locations. Therefore, the entire collection of memory modules $\mathcal{M}_1^{\bowtie}, \ldots, \mathcal{M}_n^{\bowtie}$ can be replaced by a single memory module $\mathcal{M}^{\bowtie}$ with the same set of port pairs that keeps the internal state $\mathfrak{gs}$ to answer all queries. To reconstruct an input or store an output, $\hat{\mathcal{F}}_p^{\diamond}$ always addresses a block $\mathfrak{gs}[t, \delta, \ell]$.

This allows us to define a collection $\mathcal{F}_{pd}^{\bowtie}$ that meets the specification of $\mathcal{F}_{pd}^{\diamond}$ by replacing $\mathcal{R}_u$ and $\mathcal{S}_u$ with a machines $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ which directly communicate with the shared memory $\mathcal{M}^{\bowtie}$. Given an input (t, δ, ℓ), the machine $\mathcal{R}_u^{\bowtie}$ fetches $\mathfrak{gs}[t, \delta, \ell]$, reconstructs the underlying value x and sends it back to $\mathcal{M}^{\bowtie}$. Given an input (t, δ, ℓ, y) the machine $\mathcal{S}_u^{\bowtie}$ computes shares for y and writes them to $\mathfrak{gs}[t, \delta, \ell]$.

To preserve compatibility, we replace $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{S}}$ with machines $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ that consolidate memory locations instead of messages. The machine $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ collects location tuples $(t_1, t_2, \boldsymbol{\alpha}, \ldots)$ instead of incoming messages. When all tuples for a particular round of computation have arrived, $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ extracts all unique input locations and reconstructs inputs with the help of $\mathcal{R}_u^{\bowtie}$. After that it sends output locations to $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ and proceeds as $\mathcal{T}_{\mathcal{R}}$. Whenever $\mathcal{F}_p^{\bowtie}$ wakes up $\mathcal{T}_{\mathcal{S}}^{\bowtie}$, it first clocks in a message from $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ that contains the output locations and uses $\mathcal{S}_u^{\bowtie}$ to share the outputs. To isolate $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ from the memory, we add a separate machine $\mathcal{T}_{\mathcal{M}}^{\bowtie}$ between $\mathcal{T}_{\mathcal{R}}^{\bowtie}, \mathcal{T}_{\mathcal{S}}^{\bowtie}$ and A. This machine $\mathcal{T}_{\mathcal{M}}^{\bowtie}$ manages all cases when $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ or $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ give any values to A by receiving the locations from $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ or $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ and retrieving the necessary values itself. For clarity, let us define $\hat{\mathcal{F}}^{\bowtie}{}_p$ as a collection of $\mathcal{T}_{\mathcal{R}}^{\bowtie}, \mathcal{F}_p^{\bowtie}, \mathcal{T}_{\mathcal{S}}^{\bowtie}$ as we push $\mathcal{T}_{\mathcal{M}}^{\bowtie}$ into the adversary in the next step.

Thanks to $\mathcal{T}^{\bowtie}_{\mathcal{M}}$, $\hat{\mathcal{F}}^{\bowtie}{}_p$ can still leak inputs, outputs and their locations, or perform selective aborts based on values. The adversary gets a limited access to the memory $\mathcal{M}^{\bowtie}$ through $\mathcal{T}^{\bowtie}_{\mathcal{M}}$. Note that we are working with the canonical ideal functionalities as specified in Section 2.4.2 which limits $\mathcal{T}^{\bowtie}_{\mathcal{M}}$ and subsequent adversaries to only read the shares of corrupted parties. $\mathcal{F}^{\bowtie}_{\text{io}}$ is defined based on $\mathcal{F}^{\diamond}_{\text{io}}$ with the difference that it writes the values to $\mathcal{M}^{\bowtie}$ and sends only the location information to $\mathcal{I}^{\bowtie}$. Define $\Pi^{\bowtie}$ as an extended collection consisting of machines $\mathcal{I}^{\bowtie}_1, \ldots, \mathcal{I}^{\bowtie}_n$, $\mathcal{M}^{\bowtie}$, $\hat{\mathcal{F}}^{\bowtie}{}_1, \ldots, \hat{\mathcal{F}}^{\bowtie}{}_k$,$\mathcal{G}^{\diamond}_{1,1}, \ldots, \mathcal{G}^{\diamond}_{g,n}$, $\mathcal{F}^{\bowtie}_{\text{io}}$ with all attached buffers. Let $\phi_{\bowtie}(\mathsf{A}^{\diamond})$ be the reduced collection consisting of A and all $\mathcal{T}^{\bowtie}_{\mathcal{M}}$. Therefore, the new adversary expects memory locations from the ideal functionality and fetches corrupted parties values from the memory $\mathcal{M}^{\bowtie}$.

Theorem 8. *Let $\Pi^{\diamond}$ be a well-formed protocol specification that is in a canonical form and let $\mathbb{E}$ be the set of compatible environments. Then,*

$$\forall \mathsf{Env} \in \mathbb{E}: \quad \forall \mathsf{A}^{\diamond} \in \mathbb{A}^{\diamond}_{\Pi^{\diamond},\mathsf{Env}}: \qquad \mathsf{Env}\langle \Pi^{\diamond}, \mathsf{A}^{\diamond} \rangle \equiv \mathsf{Env}\langle \Pi^{\bowtie}, \phi_{\bowtie}(\mathsf{A}^{\diamond}) \rangle$$
$$\forall \mathsf{Env} \in \mathbb{E}: \quad \forall \mathsf{A}^{\bowtie} \in \mathbb{A}^{\bowtie}_{\Pi^{\bowtie},\mathsf{Env}}: \qquad \mathsf{Env}\langle \Pi^{\bowtie}, \mathsf{A}^{\bowtie} \rangle \equiv \mathsf{Env}\langle \Pi^{\diamond}, \phi^{*}_{\bowtie}(\mathsf{A}^{\bowtie}) \rangle$$

where and $\mathbb{A}^{\diamond}_{\Pi^{\diamond},\mathsf{Env}}$ is the set of compatible simplistic adversaries for $\Pi^{\diamond}$ and $\mathbb{A}^{\bowtie}_{\Pi^{\bowtie},\mathsf{Env}}$ for $\Pi^{\bowtie}$. The resulting adversaries $\phi_{\bowtie}(\mathsf{A}^{\diamond})$ and $\phi^{*}_{\bowtie}(\mathsf{A}^{\bowtie})$ are simplistic.

Proof. The buffers between $\mathcal{I}^{\diamond}_j$ and $\mathcal{I}^{\bowtie}_j$ and adversary are the same in both worlds, and the adversary accesses the same state of the interpreter in $\mathcal{M}^{\diamond}_i$ and $\mathcal{M}^{\bowtie}$. The differences are the joining $\mathcal{M}^{\diamond}_i$ to $\mathcal{M}^{\bowtie}$, introducing $\mathcal{G}^{\diamond}_{q,i}$ as separate functionalities and decomposing $\mathcal{F}_p$ to $\mathcal{T}^{\bowtie}_{\mathcal{S}}$, $\mathcal{T}^{\bowtie}_{\mathcal{R}}$ and $\mathcal{T}^{\bowtie}_{\mathcal{M}}$ that also interact with $\mathcal{M}^{\bowtie}$.

All memory addresses are written once by a well-formed protocol, thus the outputs of $\mathcal{G}^{\diamond}_{q,i}$ are never overwritten and the adversary can access them from $\mathcal{M}^{\bowtie}$ when needed. In addition, the buffers connecting $\mathcal{G}^{\diamond}_{q,i}$ to $\mathcal{I}^{\bowtie}_i$ and $\mathcal{M}^{\bowtie}$ are sender-clocked and therefore this separation is invisible for A. Therefore, separating the local functionalities is equivalent to the local functionalities inside $\mathcal{I}^{\diamond}$ where $\mathcal{I}^{\diamond}$ writes all outputs to $\mathcal{M}^{\bowtie}$. As all ideal functionalities are canonical, we can replace all instances of $\mathcal{S}_u$ and $\mathcal{R}_u$ with $\mathcal{S}^{\bowtie}_u$ and $\mathcal{R}^{\bowtie}_u$ provided that we rewire the memory to $\mathcal{M}^{\bowtie}$.

Adversary $\phi_{\bowtie}(\mathsf{A}^{\diamond})$ is simplistic as it respects the clocking rules of simplistic $\mathsf{A}^{\diamond}$ and the construction did not change the clocking or the modified memory locations. Similarly, transforming $\mathsf{A}^{\bowtie}$ to equivalent $\mathsf{A}^{\diamond}$ is straightforward and can be accomplished getting the same values as read from $\mathcal{M}^{\bowtie}$ from communication with $\mathcal{F}^{\diamond}$. □

3.3. Reduction to Abstract Memory Model

Simplistic adversaries are extremely restricted. They can alter a fixed set of memory locations and decide when ideal functionalities and interpreters perform their computations. As a protection domain $\mathcal{F}^{\bowtie}_{pd}$ consists of ideal functionalities $\hat{\mathcal{F}}^{\bowtie}{}_1, \ldots, \hat{\mathcal{F}}^{\bowtie}{}_1$ which use two universal machines $\mathcal{R}^{\bowtie}_u$ and $\mathcal{S}^{\bowtie}_u$ for reconstruction and sharing, it is straightforward to define an intermediate memory module $\mathcal{M}_0$ that stores reconstructed values. With additional simplifications we can restate all operations in terms of $\mathcal{M}_0$ and quantify memory modifications in terms of underlying values instead of shares. In more formal terms, we define a $*$-operator that acts on protocols and their components together with a universal $\phi_* : \mathbb{A}^{\bowtie} \to \mathbb{A}^{*}$ and its semireverse $\phi^{*}_{*} : \mathbb{A}^{*} \to \mathbb{A}^{\bowtie}$ that achieves

$$\forall \Pi \in \mathbb{P}: \ \forall \mathsf{Env} \in \mathbb{E}_{\Pi}: \ \forall \mathsf{A} \in \mathbb{A}^{\bowtie}: \quad \mathsf{Env}\langle \Pi^{\bowtie}, \mathsf{A}^{\bowtie} \rangle \equiv \mathsf{Env}\langle \Pi^{*}, \phi_{*}(\mathsf{A}^{\bowtie}) \rangle \tag{10}$$
$$\forall \Pi \in \mathbb{P}: \ \forall \mathsf{Env} \in \mathbb{E}_{\Pi}: \ \forall \mathsf{A} \in \mathbb{A}^{*}: \quad \mathsf{Env}\langle \Pi^{*}, \mathsf{A}^{*} \rangle \equiv \mathsf{Env}\langle \Pi^{\bowtie}, \phi^{*}_{*}(\mathsf{A}^{*}) \rangle \tag{11}$$

where $\mathbb{P}$ is the set of protocols with canonical specification and $\mathbb{E}_{\Pi}$ is the set of compatible environments. The forward transformation is covered step by step through the whole section and the reverse is summarised in Lemma 12.

3.3.1. Introduction of Abstract Memory

We split the memory $\mathcal{M}^{\bowtie}$ of values and shares and introduce a dedicated memory module $\mathcal{M}_0$ for the values. The internal state of $\mathcal{M}_0$ is a three-dimensional array $\mathfrak{s}_0$ is such that an entry $\mathfrak{s}_0[t,\delta,\ell]$ contains the value corresponding to shares in $\mathfrak{gs}[t,\delta,\ell]$ in $\mathcal{M}$ at all critical time-steps. Then, we place machines $\mathcal{R}_u^*$ and $\mathcal{S}_u^*$ between $\mathcal{M}_0$ and $\mathcal{M}$ together with a pair of sender-clocked buffers for synchronisation as depicted in Figure 9. Recall that $\mathcal{T}_{\mathcal{R}}$ only receive public parameters. Local variables and the state of each party are also stored in $\mathcal{M}_0$. For that we replace the pair of sender clocked buffers between each interpreter $\mathcal{I}_i^{\bowtie}$ and $\mathcal{M}^{\bowtie}$ with a corresponding buffer pair between $\mathcal{I}_i^*$ and $\mathcal{M}_0$. $\mathcal{M}_0$ allows the adversary can read and write local variables of corrupted parties.

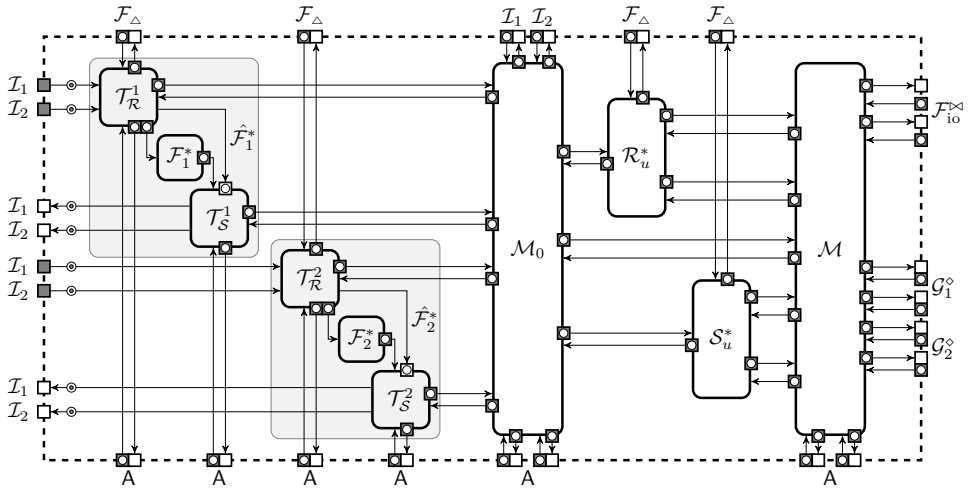


Figure 9. Separating memory to $\mathcal{M}_0$ for values and $\mathcal{M}$ for shares, $\mathfrak{F}_1$.

We synchronise the states of $\mathfrak{s}_0$ and $\mathfrak{gs}$ as follows. When a machine queries a share from a location (t,δ,ℓ) and $\mathfrak{gs}[t,\delta,\ell]$ is empty or invalid $\mathcal{M}$ writes $\textsc{Share}(t,\delta,\ell)$ to the synchronisation buffer. Then, $\mathcal{M}_0$ uses $\mathcal{S}_u^*$ to share $\mathfrak{s}_0[t,\delta,\ell]$ and gives control back to $\mathcal{M}$ who uses $\mathfrak{gs}[t,\delta,\ell]$ to complete the query. Whenever $\mathfrak{gs}[t,\delta,\ell]$ is updated $\mathcal{M}$ writes $\textsc{Update}(t,\delta,\ell)$ to the synchronisation buffer. After that $\mathcal{M}_0$ uses $\mathcal{R}_u^*$ to update $\mathfrak{s}_0[t,\delta,\ell]$ and gives control back to $\mathcal{M}$. Finally, $\mathcal{M}_0$ must write $\textsc{Invalid}(t,\delta,\ell)$ to the synchronisation buffer when the value of $\mathfrak{s}_0[t,\delta,\ell]$ is updated. After this $\mathcal{M}$ marks the location (t,δ,ℓ) as invalid and gives control back to $\mathcal{M}_0$. This mechanism guarantees that $\mathcal{M}_0$ and $\mathcal{M}$ give always coherent replies to other machines.

As a result, we can connect $\hat{\mathcal{F}^{\bowtie}}_p$ directly to $\mathcal{M}_0$ instead of $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ without changing the execution, let this be $\hat{\mathcal{F}}^*$. When $\mathcal{T}_{\mathcal{R}}$ queries (t,δ,ℓ), the machine $\mathcal{M}_0$ replies $\mathfrak{s}_0[t,\delta,\ell]$. When $\mathcal{T}_{\mathcal{S}}$ wants to share a value x to a location (t,δ,ℓ), the machine $\mathcal{M}_0$ sets $\mathfrak{s}_0[t,\delta,\ell] = x$ and marks the location (t,δ,ℓ) as invalid. The latter allows us to replace a sub-collection inside $\Pi^{\bowtie}$ without changing the execution outcome. More precisely, let $\mathfrak{F}_0$ be an extended collection consisting of $\hat{\mathcal{F}^{\bowtie}}_1, \ldots, \hat{\mathcal{F}^{\bowtie}}_p, \mathcal{R}_u^{\bowtie}, \mathcal{S}_u^{\bowtie}, \mathcal{M}^{\bowtie}$ and let $\mathfrak{F}_1$ be an extended collection consisting of $\hat{\mathcal{F}^*}_1, \ldots, \hat{\mathcal{F}^*}_p, \mathcal{M}_0, \mathcal{R}_u^*, \mathcal{S}_u^*, \mathcal{M}$.

Theorem 9. *Collections $\mathfrak{F}_0$ and $\mathfrak{F}_1$ are observationally equivalent for well-formed protocol specification in a canonical form.*

Proof. The substitution does not change communication between the collection and $\mathcal{F}_{\triangle}$. Therefore, it is sufficient to show that machines $\mathcal{M}, \hat{\mathcal{F}}_1^*, \ldots, \hat{\mathcal{F}}_p^*$ run in the same order and provide identical replies to A and Π_{e}. The latter is sufficient as communication with other machines in the collection is sender clocked and invisible to outside observers.

In particular, we need to show that when a machine queries $\mathfrak{gs}[t,\delta,\ell]$ its value is same in both collections. Assume that so far all queries have yielded identical results. In $\mathfrak{F}_0$, the response to $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ is computed from $\mathfrak{gs}[t,\delta,\ell]$, while $\mathcal{M}_0$ responds $\mathfrak{s}_0[t,\delta,\ell]$. By the construc-

tion the value of $\mathfrak{s}_0[t,\delta,\ell]$ is invalidated whenever $\mathfrak{gs}[t,\delta,\ell]$ is updated. Consequently, the reply $\mathfrak{s}_0[t,\delta,\ell]$ contains the reconstruction of $\mathfrak{gs}[t,\delta,\ell]$ as in $\mathfrak{F}_0$.

Assume that the location $\mathfrak{gs}[t,\delta,\ell]$ of a new query is initialised. If $\mathcal{T}^*_{\mathcal{S}}$ was the last machine to update $\mathfrak{gs}[t,\delta,\ell]$ in $\mathfrak{F}_0$ then it was also the last to update $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathfrak{F}_1$. Therefore, $\mathfrak{s}_0[t,\delta,\ell]$ contains the correct value. By the construction, the update marked the location $\mathfrak{gs}[t,\delta,\ell]$ as invalid and thus the reply yields shares generated by $\mathcal{S}_u$. For other machines, the memory cells are updated identically in both collections. If $\mathfrak{gs}[t,\delta,\ell]$ is uninitialised in $\mathfrak{F}_0$, then no machine has updated this location. Thus, $\mathcal{T}^{\bowtie}_{\mathcal{S}}$ could not have initialised $\mathfrak{s}_0[t,\delta,\ell]$ nor could any other machine have initialised $\mathfrak{gs}[t,\delta,\ell]$ in $\mathfrak{F}_1$. □

3.3.2. Extended Modification-Awareness

As the adversary can modify shares of corrupted parties and therefore change the values in $\mathcal{M}_0$. Here, we study under which assumptions the adversary can update $\mathfrak{s}_0[t,\delta,\ell]$ itself after modifying shares $\mathfrak{gs}[t,\delta,\ell]$. The concept of modification awareness (Definition 3) is meant to tackle this, but some interactions in the computations are not captured by the definition. We can apply the extractor from the modification awareness in our current setting but its success is not necessarily the same.

We define a collection $\mathfrak{F}_2$ where we place the extractor $\mathcal{E}$ between $\mathcal{M}_0$, $\mathcal{M}$ and A as in Figure 10. The extractor $\mathcal{E}$ which forwards messages between A to $\mathcal{M}$ and simultaneously extracts changes Δ corresponding to share modifications. These modifications are sent to $\mathcal{M}_0$ who updates the state $\mathfrak{s}_0$ accordingly. A slightly modified machine $\mathcal{M}$ does not invalidate a location $\mathfrak{s}_0[t,\delta,\ell]$ when the extractor $\mathcal{E}$ alters the location $\mathfrak{gs}[t,\delta,\ell]$.

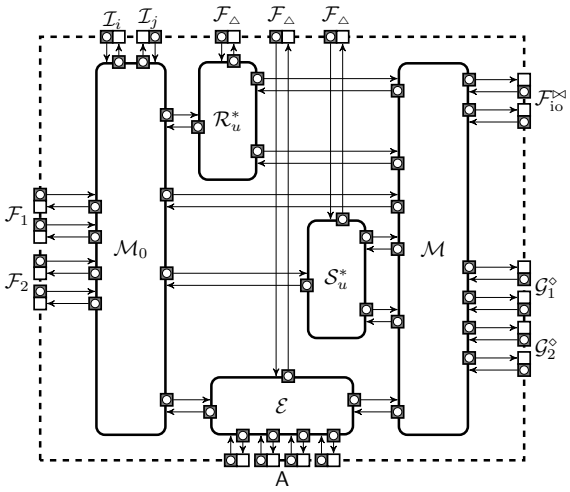


Figure 10. Separating a value modification module $\mathcal{E}$ from $\mathcal{M}$, collection $\mathfrak{F}_2$.

Definition 22. *An adversarial modification of* $\mathfrak{gs}[t,\delta,\ell]$ *is oblique if the extractor* $\mathcal{E}$ *fails to correctly update the value of* $\mathfrak{s}_0[t,\delta,\ell]$.

Theorem 10. *Collections* $\mathfrak{F}_1$ *and* $\mathfrak{F}_2$ *are observationally equivalent for an environment adversary pair provided that the extraction failure is negligible.*

Proof. As the machine $\mathcal{E}$ forwards the communication between A and $\mathcal{M}$ without modification, the states of the configurations can diverge only if the values of $\mathfrak{s}_0[t,\delta,\ell]$ differ when $\mathcal{M}_0$ replies to some query. It is straightforward to see that the latter occurs only if the extraction fails for $\mathfrak{gs}[t,\delta,\ell]$. □

Definition 23. *Local operation is transparent if any modification of output shares can be effectively converted to a modification of inputs and oblique modification goes to oblique modification.*

Common local operations—copying and linear combination—do not contribute to extraction failures as they are reversible and thus oblique modifications can be back-propagated. To bind the probability of extraction failures, we need to reason about the environment Π_{e} and protocol Π together as oblique changes may enter the protocol through inputs. For instance, in verifiable storage domains, an adversary can corrupt a parent party in Π_{e} and invalidate its input shares for Π. As A knows the original shares, it can modify invalid shares to the originals inside Π. The resulting modification is oblique. By definition, $\mathcal{M}_0$ holds $\perp$ in that location meaning that the modification function always results in $\perp$ independently of the extractor outcome.

Non-canonical ideal functionalities are a potential source of oblique modifications. For example, consider a weird flavour of non-fair computations where the adversary learns the output shares of honest parties before it can invalidate some of them. To create an oblique modification, the adversary can later corrupt a party with invalidated share and replace it with the original. Unconventional local functionalities can also create oblique modifications simply by invalidating the input shares. As the adversary knows the original shares, it can create oblique modification by restoring the valid share.

Lemma 4. *Assume that all local operations are transparent and ideal functionalities are in a canonical form. Assume that $\mathcal{S}_\delta$ generates all protocol inputs for domain δ and the inputs of $\mathcal{S}_\delta$ are determined by the adversary. Then, the probability of extraction failures in a well-formed program is negligible for simplistic adversaries provided that every storage domain is modification aware.*

Proof. Let A be a simplistic adversary that with probability ε creates an extraction failure in storage domain δ in some protocol instance. Then, we can define a new adversary B against modification awareness of storage domain δ (see Figure 3b).

First, note that the adversary B can perfectly simulate $\mathcal{S}$ and $\mathcal{R}$, as it has direct access to $\mathcal{S}_\delta$ (inputting values through $\mathcal{L}$ and reading from $\mathcal{L}^*$ through $\mathcal{E}_\delta$) and $\mathcal{R}_\delta$ in the modification game and can run the trusted setup for all other domains. As a consequence, B can perfectly simulate protocol inputs and ideal functionalities $\mathcal{F}_p$. The simulation of the protocol is straightforward, as the flow of the interpreters depends only on local values and new local values can be created by ideal functionalities or local operations. As B learns the entire output of $\mathcal{S}_\tau$ for protection domains $\tau \neq \delta$, the shares of $\mathfrak{gs}[t, \tau, \ell]$ can be directly computed. The requested shares of δ can be computed by backtracking all dependencies to shares generated by $\mathcal{S}_\delta$ and then redoing all local operations.

Next, we add back-propagation of modified shares for the domain δ. By definition, a simplistic adversary A can modify only the inputs of ideal functionalities. If a modification is generated by $\mathcal{S}_\delta$ then this modification can be used to win the modification game. If the modification is computed by a local functionality then due to transparency, B can always back-propagate a change to a change of one input of $\mathcal{G}_q$ and an oblique change is guaranteed to remain oblique. This process can be repeated until B reaches a protocol input or an output of ideal functionality $\mathcal{F}_p$. These are potential challenge shares generated by $\mathcal{S}_\delta$.

Each input modification leads to a modification of a potential challenge. The total number of such modifications is bounded by the number of inputs to ideal functionalities that have domain δ. Let the corresponding number be n. As we cannot check which modification is oblique, B must set one of them randomly as the challenge modification when it generates the shares. As a result, B succeeds in modification awareness game with probability $\frac{\varepsilon}{n}$. The claim follows as ε is negligible whenever $\frac{\varepsilon}{n}$ is negligible. □

Corollary 3. *If all local operations are transparent, ideal functionalities are in a canonical form and the environment is simulatable; then, the probability of extraction failures in a well-formed program with modification aware storage domain is negligible for simplistic adversaries.*

Proof. The proof of Lemma 4 can be generalised to a class of environments that can be simulated in the modification awareness game. Simulation means running the environment

analogously to the protocol as part of the modification game. Therefore, we require the assumption that Π_e does not leak the setup parameters or joint state. In addition, in such simulation we can notice the modifications to inputs of Π done in Π_e that could be oblique. Such modifications can be used or propagated similarly to the actions with values of Π in Lemma 4. □

3.3.3. Meaningful Local Operations

Local operations cause another state update procedure where the information flows from the machine $\mathcal{M}$ to $\mathcal{M}_0$. Meaningful local operations (Definition 6) produce such output shares that the state $\mathfrak{s}_0$ can be updated from itself. Note that a value $\mathfrak{s}_0[t, \delta, \ell]$ in $\mathcal{M}_0$ is read-only by $\mathcal{F}_p$. An adversary can obliviously update $\mathfrak{s}_0[t, \delta, \ell]$ by sending differences Δ to $\mathcal{M}_0$ through $\mathcal{E}$. The canonical protocol description guarantees that $\mathcal{F}_p^*$ reads $\mathfrak{s}_0[t, \delta, \ell]$ only after the location $\mathfrak{gs}[t, \delta, \ell]$ has been initialised. Thus, if a local operation $\mathcal{G}_q$ initialises $\mathfrak{gs}[t, \delta, \ell]$ then it is always completed before the read of $\mathfrak{s}_0[t, \delta, \ell]$.

This allows us to define a new collection $\mathfrak{F}_3$ where machines $\mathcal{G}_q^0$ update the state $\mathfrak{s}_0$ instead of $\mathcal{R}_u^*$. Each $\mathcal{G}_q^0$ interacts with interpreters $\{\mathcal{I}_j\}_{j \in \mathcal{J}_q}$ and the abstract memory $\mathcal{M}_0$ and receives public parameters from $\mathcal{F}_\triangle$. For that we slightly modify the behaviour of $\mathcal{I}_i$, $\mathcal{M}$ and $\mathcal{M}_0$. Whenever $\mathcal{I}_i$ calls $\mathcal{G}_q$ it also calls $\mathcal{G}_q^0$ after it gets the control back. The machine $\mathcal{G}_q^0$ immediately checks if the operation has been computed in $\mathcal{M}_0$ or if there are enough inputs in $\mathcal{M}_0$ to complete the local operation in $\mathcal{M}_0$. If all inputs are present and the operation has not been executed yet, then $\mathcal{G}_q^0$ reads inputs from $\mathfrak{s}_0$, computes the outputs according to g_q and writes it to $\mathfrak{s}_0$. Modified $\mathcal{M}$ does not invalidate a location $\mathfrak{s}_0[t, \delta, \ell]$ when the functionality $\mathcal{G}_q$ alters the location $\mathfrak{gs}[t, \delta, \ell]$ but otherwise behaves as before.

Theorem 11. *Collections $\mathfrak{F}_2$ and $\mathfrak{F}_3$ are observationally equivalent provided that a well-formed protocol specification is in a canonical form and all local operations are meaningful and implement some deterministic functionality.*

Proof. Modifications to the collection $\mathfrak{F}_2$ do not change the order of machine activations. Indeed, changes in $\mathcal{M}$ eliminate only a ping-pong interaction between $\mathcal{M}$ and $\mathcal{M}_0$. Changes in $\mathcal{I}_i$ inject a ping-pong interaction between $\mathcal{I}_i$ and $\mathcal{G}_q^0$. As the communication in these interactions is sender-clocked the other machines can distinguish collections only based on the states of $\mathcal{M}_0$ and $\mathcal{M}$. Clearly, modifications do not change the state of $\mathcal{M}$ as long as responses to queries $\mathfrak{gs}[t, \delta, \ell]$ and $\mathfrak{s}_0[t, \delta, \ell]$ remain same in both collections. Therefore, it is sufficient to consider only the responses to queries $\mathfrak{s}_0[t, \delta, \ell]$.

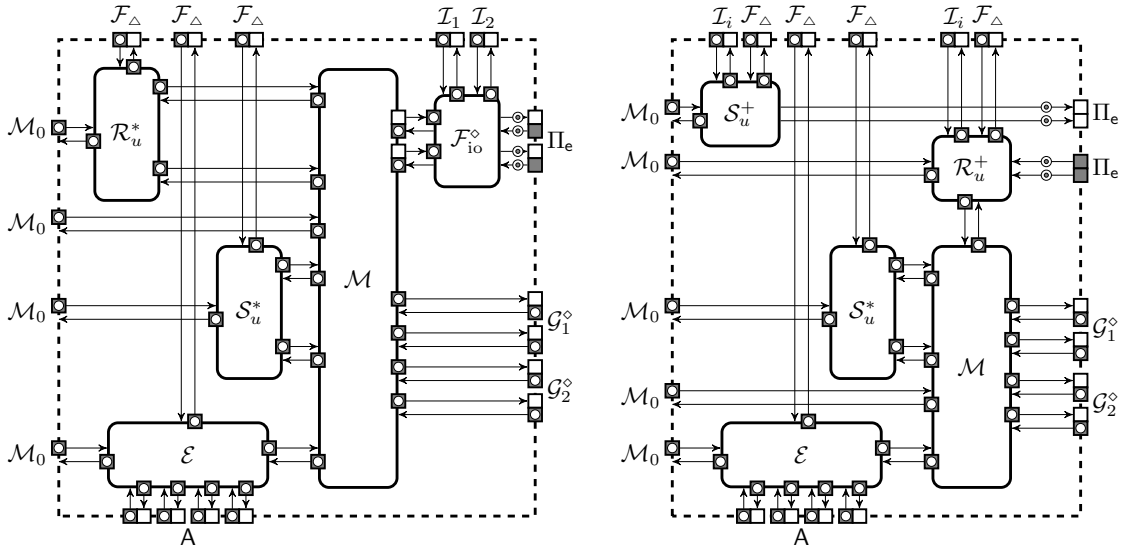
Only an ideal functionality $\mathcal{F}_p^*$ (or $\mathcal{F}_{\text{io}}^\diamond$) can query $\mathfrak{s}_0[t, \delta, \ell]$. Therefore, it is sufficient to consider when $\mathcal{G}_q$ updates $\mathfrak{gs}[t, \delta, \ell]$, as otherwise both collections behave identically. For well-formed programs, the value $\mathfrak{gs}[t, \delta, \ell]$ is never fetched by $\mathcal{F}_p$ before all interpreters have defined their shares. Although $\mathfrak{gs}[t, \delta, \ell]$ may depend on several local operations we know that all of them complete by that time. As the adversary never overwrites inputs of local computations, we can inductively prove that the update chain gives the the same result for $\mathfrak{s}_0[t, \delta, \ell]$ as the reconstruction step in $\mathfrak{F}_2$. □

The result can be generalised to meaningful non-deterministic local functionalities. First, all inputs to $\mathcal{G}_q$ must have a correct distribution, or otherwise we cannot apply the definition. Second, the inputs of local functionalities with non-deterministic outputs must be independent. For instance, if the local functionality $\mathcal{G}_q$ is deterministic and we re-run the same shares we get the same output while g_q gives two independent outputs.

3.3.4. Isolation of Protocol Outputs

By construction all protocol outputs are obtained by releasing shares of type $\mathfrak{gs}[t, \delta, \ell]$ even if outputs are locally private values. As a next step, we modify the construction so that all output shares are generated anew by $\mathcal{S}_u^*$. This a major prerequisite for isolating the adversary from the memory $\mathcal{M}$. We define a new collection $\mathfrak{F}_4$ in Figure 11 where $\mathcal{F}_{\text{io}}^\diamond$ is replaced by $\mathcal{R}_u^+$ to construct inputs and $\mathcal{S}_u^+$ to create outputs. $\mathcal{S}^+$ always gives control

back to $\mathcal{I}_i$ that called it. $\mathcal{R}_u^+$ writes the value to $\mathcal{M}_0$ as soon as sufficient shares have been received and always forwards the signal to the party $\mathcal{I}_i$ whose input was clocked to it. All these added buffers are sender clocked. SEND instructions from $\mathcal{I}_i$ are forwarded only to $\mathcal{S}_u^+$. The behaviour of $\mathcal{S}_u^+$ is different from $\mathcal{F}_{\text{io}}^{\diamond}$. The machine $\mathcal{S}_u^+$ keeps a special cache $\mathcal{L}$ of published values which is initially empty. For each input SEND or DMACALL from $\mathcal{I}_i$, it extracts all locations (t, δ, ℓ) of output shares. If a triple $(t, \delta, \ell) \notin \mathcal{L}$, the machine $\mathcal{S}_u^+$ fetches the corresponding value from $\mathcal{M}_0$, computes the shares and stores them into $\mathcal{L}[t, \delta]$. If $(t, \delta, \ell) \in \mathcal{L}$, then it uses the shares in $\mathcal{L}[t, \delta]$.



(**a**) $\mathfrak{F}_3$ (**b**) $\mathfrak{F}_4$

Figure 11. Memory models for input–output isolation.

Definition 24. *A protocol environment pair* $\mathsf{Env}\langle\Pi\rangle$ *is in an output-isolated configuration for a class of adversaries* $\mathbb{A}$ *and resource consumption constraints if there is a construction* ϕ *such that* $\mathsf{Env}\langle\mathfrak{F}_3, \mathsf{A}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_4, \phi(\mathsf{A})\rangle$ *for* $\mathsf{A} \in \mathbb{A}$ *and the construction* ϕ *satisfies resource constraints. A protocol* Π *is output-isolated if this property holds for any environment.*

Any protocol where all outputs are returned by deterministic $\mathcal{S}$ (e.g., local values) or where no shared outputs are returned are output isolated. The value $\mathcal{L}[t, \delta, \ell]$ is guaranteed to have the same distribution as $\mathfrak{gs}[t, \delta, \ell]$ when outputs are computed by an ideal functionality. However, the behaviour of $\mathcal{F}_p$ may still reveal the discrepancy. For instance, A may decide whether to abort or continue based on the output shares of honest parties if $\mathcal{F}_p$ can reveal them to A. The latter creates a discrepancy that is impossible to resolve in $\mathfrak{F}_4$. It is also an explicit weakness of a protocol. Any protocol can be converted to output-isolated protocol by resharing outputs before returning them. Hence, almost all functionalities used in practice can be implemented with isolated outputs.

Lemma 5. *A canonical well-formed protocol specification is output-isolated for coherent adversaries if all outputs are computed by some ideal functionalities with standard corruption mode, output shares are not used further in computations and they are immediately returned as outputs.*

Proof. It is straightforward to see that $\mathcal{R}^+$ in $\mathfrak{F}_4$ processes inputs identically to $\mathcal{F}_{\text{io}}^{\diamond}$ in $\mathfrak{F}_3$. Therefore, we need to consider only the output generation. In $\mathfrak{F}_3$, output generation starts when $\mathcal{F}_p$ writes its output to $\mathfrak{s}_0[t, \delta, \ell]$. At some point, the adversary clocks the corresponding OK message from $\mathcal{F}_p$ to some $\mathcal{I}_i$. $\mathcal{I}_i$ writes DMACALL or SEND message to push its share of $\mathfrak{gs}[t, \delta, \ell]$ to Π_{e}, and $\mathcal{F}_{\text{io}}$ fetches the corresponding part of $\mathfrak{gs}[t, \delta, \ell]$. By construction, $\mathfrak{gs}[t, \delta, \ell]$ is created by $\mathcal{S}_u^*$ from $\mathfrak{s}_0[t, \delta, \ell]$. In $\mathfrak{F}_4$, $\mathcal{I}_i$ sends DMACALL or SEND message to $\mathcal{S}_u^+$, which uses $\mathcal{L}[t, \delta, \ell]$ instead of $\mathfrak{gs}[t, \delta, \ell]$ to create the output. By construction,

$\mathfrak{gs}[t, \delta, \ell]$ and $\mathcal{L}[t, \delta, \ell]$ have the same distribution. However, there is a discrepancy between $\mathfrak{gs}[t, \delta, \ell]$ and protocol output.

Note that only A can observe the discrepancy, as honest parties do not use output shares in further computations. Coherent A has also corrupted the parent node $\mathcal{P}_i^*$ in the inner environment and thus can fetch shares $\mathfrak{gs}[t, \delta, \ell]$ from the outputs sent to $\mathcal{P}_i^*$. For that, we must guarantee that the outputs arrive earlier than A wants to read the i-th share of $\mathfrak{gs}[t, \delta, \ell]$. The latter is straightforward as the output shares are published as soon as they become available for A and A can always clock the output to $\mathcal{P}_i^*$. □

Lemma 6. *Any protocol is output-isolated for the class of environments that do not use randomised output shares at all.*

Proof. Note that the changes introduced by $\mathfrak{F}_4$ alter only output shares given to the environment and only for non-deterministic $\mathcal{S}$. As a result, the adversary can compute the protocol outputs based on the state of corrupted party $\mathcal{P}_i$ instead of the actual protocol output reaching the parent node $\mathcal{P}_i^*$. This hides the discrepancy between $\mathfrak{F}_3$ and $\mathfrak{F}_4$, as these shares are never used in any other protocol. □

Lemma 7. *Any protocol where output shares have the same distribution as freshly generated shares is output-isolated for the class of adversaries that do not access the protocol state at all.*

Proof. By assumption all protocol outputs are identically distributed in $\mathfrak{F}_3$ and $\mathfrak{F}_4$. As the adversary A learns nothing about the protocol state it cannot detect the discrepancy, nor can the adversary alter the protocol execution as it cannot alter the protocol state. □

Theorem 12. *Let $\mathcal{F}$ be a canonical ideal functionality that does not leak shares to adversary and let Π be a protocol that is as secure as $\mathcal{F}$ for a class of environments $\mathbb{E}$ and adversaries $\mathbb{A}$. Then, Π is also output-isolated for the same classes of adversaries and environments.*

Proof. Let $\mathsf{A} \in \mathbb{A}$ be an adversary against the original protocol and environment pair $\mathsf{Env}\langle\Pi\rangle$, and let ϕ be the construction that proves the security of the protocol. Then, $\phi(\mathsf{A})$ is an ideal adversary that defines protocol inputs and receives protocol outputs in $\mathcal{F}$. As the adversary $\phi(\mathsf{A})$ does not learn values from the ideal functionality $\mathcal{F}$, its interface is compatible with $\mathsf{Env}\langle\Pi\rangle$. By definition, Π is equivalent to $\mathcal{F}$ for all adversaries that behave honestly. Indeed, if an adversary B does not corrupt parties then $\phi(\mathsf{B})$ cannot also corrupt parties. The same must be true for the adversary $\phi(\mathsf{A})$. Although $\phi(\mathsf{A})$ corrupts some parties and alters their inputs, these parties remain honest in the protocol. As a result, $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle \equiv \mathsf{Env}\langle\Pi, \phi(\mathsf{A})\rangle$. The claim follows as $\phi(\mathsf{A})$ satisfies the assumptions of Lemma 7. □

A protocol is secure if any adversary can be converted to an adversary against ideal implementation. Therefore, all adversaries against $\mathfrak{F}_4$ can be modified to adversaries that do not read the shares of the outputs from $\mathcal{M}$ at all. That means that output-isolation is necessary for security in general. The intuition is that since these shares are updated during outputting and not used in the protocol then reading them is only relevant in Π_{e}. Thus, in the following we only consider such adversaries.

3.3.5. Complete Memory Isolation

The transformation defining protocol output isolation in the previous section cuts the last link between the memory $\mathcal{M}$ and the environment Env. Only the actions of the adversary A can now depend on $\mathfrak{gs}$ all local computations and ideal functionalities only use $\mathfrak{s}_0$. Thus, honest execution does not require $\mathfrak{gs}$ and in this section we show how A can simulate $\mathfrak{gs}$ for hiding storage domains. Therefore, we will completely remove $\mathcal{M}$ from the protocol description.

First, we minimise the number of memory locations accessed by A. Memory locations $\mathfrak{gs}[t, \delta, \ell]$ can be divided into three classes: protocol inputs, outputs of ideal functionalities and outputs of local computations. A coherent adversary can always extract protocol inputs from the parent node when the latter submits them to $\mathcal{R}_u^+$. Simulation of local computations $\mathcal{G}_q$ is straightforward provided that we know the inputs of $\mathcal{G}_q$. Only the outputs of ideal functionalities must be fetched from the memory $\mathcal{M}$. In the following, we consider the details of constructing the adversary that only reads these values written by ideal functionalities from $\mathcal{M}$ and does not touch other memory locations.

Let $\mathbb{A}_o$ be the class of adversaries that only read the outputs of $\mathcal{F}_p$ from $\mathcal{M}$. More formally, let $\mathsf{A}_o \in \mathbb{A}_o$ be the new adversary that internally runs A and clones interpreters $\mathcal{I}_1, \ldots, \mathcal{I}_n$ to answer adversarial queries about corrupted shares of $\mathfrak{gs}[t, \delta, \ell]$. To reconstruct the state of a newly corrupted party $\mathcal{P}_i$, A_o has to redo all the computations done by $\mathcal{P}_i$ so far. For that, A_o must get all protocol inputs submitted by $\mathcal{P}_i^*$. We use the assumption that the state of $\mathcal{P}_i^*$ contains enough information for A_o to repeat all computations and thus the input extraction becomes trivial. The adversary A_o must also access local variables and outcomes of random choices of $\mathcal{P}_i$ stored in $\mathcal{M}_0$. The output shares of $\mathcal{F}_p$ are queried from $\mathcal{M}$ as usual. As a result, A_o can rerun a clone of $\mathcal{I}_i$ to recreate the state of $\mathcal{P}_i$. In addition to the real values from $\mathcal{M}$, A_o uses the inputs known to it to fill the simulated memory with protocol inputs and local computation outputs that A can access.

Lemma 8. *Assume that we can rerun all computations for any corrupted parent node. Then,* $\mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}_o\rangle$ *for any coherent simplistic adversary* A.

Proof. By construction of A_o from A, we can just repeat all computations as necessary and only read the outputs of $\mathcal{F}_p$. □

Clearly, the outcome of $\mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}_*\rangle$ depends only on the locations $\mathfrak{gs}[t, \delta, \ell]$ that are read by the adversary which are the outputs of ideal functionalities $\mathcal{F}_p$. As these locations are directly updated by $\mathcal{S}_u^*$, we can remove the remaining update mechanisms for protocol inputs and local operations. Let $\mathfrak{F}_5$ be a simplified collection where R_u^+ does not update $\mathcal{M}$ and is not connected to it. Interpreters $\mathcal{I}_1, \ldots, \mathcal{I}_n$ activate only $\mathcal{G}_1^0, \ldots, \mathcal{G}_m^0$, and machines $\mathcal{G}_1^\diamond, \ldots, \mathcal{G}_m^\diamond$ are removed. Note that while the shares are not read, the adversary can still do modifications to all inputs of ideal functionalities and the outputs returned to Π_e. Therefore, we disconnect $\mathcal{E}$ from $\mathcal{M}$ and join it to A_o so that can supply it with the values that it recomputes that should be in memory $\mathcal{M}$. Let the new adversary combining A_o and $\mathcal{E}$ be $\mathsf{A}_o^\mathcal{E}$ and the respective class of adversaries $\mathbb{A}_o^\mathcal{E}$ is such that they submit direct coherent modifications to $\mathcal{M}_0$ and $\mathcal{M}$.

Lemma 9. *For any* $\mathsf{A}_o \in \mathbb{A}_o$ *defined as above we have* $\mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}_o\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^\mathcal{E}\rangle$.

Proof. By definition, A_o reads only outputs of $\mathcal{F}_p$ from $\mathcal{M}$ and these are not affected by the changes as these are written after $\mathcal{F}_p$ executes. Nothing other than values in $\mathcal{M}$ are affected by the transformation to $\mathfrak{F}_5$. Note that the extractor $\mathcal{E}$ still sees valid shares and works in the same manner as before but is just considered to be part of the adversary. □

To shield $\mathcal{M}$ completely from $\mathsf{A}_o^\mathcal{E}$, we use the simulator $\mathcal{S}_\delta^o$ from the hiding property definition (Definition 2) and assume the adversary A does not read the output shares as discussed for output-isolation. Let A_{lc} be the adversary that runs $\mathsf{A}_o^\mathcal{E}$ internally but uses simulators $\mathcal{S}_\delta^*$ for each protection domain instead of values in $\mathcal{M}$. We denote by $\mathbb{A}_{lc}$ the class of limited control adversaries (Definition 4) that do not interact with $\mathcal{M}$.

Lemma 10. *Let* Env *be an environment that uses storage domains without private parameters. Then,* $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^\mathcal{E}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{lc}\rangle$ *for any limited control adversaries* $\mathsf{A}_o^\mathcal{E} \in \mathbb{A}_o^\mathcal{E}$ *and* A_{lc} *as defined above using* $\mathsf{A}_o^\mathcal{E}$ *and not reading the output memory locations.*

Proof. W.l.o.g. we can consider only the case where the protocol uses only one storage domain δ as there are no parameters that might be shared by domains. Define an adversary B against hiding games so that the first hiding game is equivalent to $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle$ and the second to $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{lc}\rangle$. First, note that B needs to internally simulate all computations of Env. As Env does not use storage domains with private variables, B can internally evaluate $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ for any storage domain δ and recreate all computations inside Env. Second, note that machines $\mathcal{I}_1, \ldots, \mathcal{I}_n, \mathcal{F}_1, \ldots, \mathcal{F}_k, \mathcal{G}_1^0, \ldots, \mathcal{G}_m^0, \mathcal{M}_0, \mathcal{E}, \mathcal{S}_u^+, \mathcal{R}_u^+$ form a subcollection $\mathcal{Z}$ that interacts with $\mathcal{M}$ through the machine $\mathcal{S}_u^*$. The adversary A can clock some buffers inside $\mathcal{Z}$ and interact with $\mathcal{F}_1, \ldots, \mathcal{F}_k$ and $\mathcal{E}$. After simulating Env, the adversary B can produce all inputs to the protocol Π.

For the outputs of ideal functionality $\mathfrak{gs}[t, \delta, \ell]$, the adversary B interacts with the hiding game. It corrupts locations when $\mathsf{A}_o^{\mathcal{E}}$ issues corruption calls. When an ideal functionality $\mathcal{F}_p$ wants to write x to $\mathfrak{gs}[t, \delta, \ell]$, B remaps a location (t, δ, ℓ) to κ and set $\mathfrak{s}_0[\kappa] = x$ by interacting with $\mathcal{L}$. It also sets $\mathfrak{b}[\kappa] = 1$ to mark that the values is inside the protocol scope. In one game this does not affect the outcome but in the second this means that all shares will be simulated. When A_o queries corrupted shares of $\mathfrak{gs}[t, \delta, \ell]$, the adversary B fetches corresponding shares from $\mathcal{L}^*$.

The first hiding game is equivalent to $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle$ as the shares are generated by $\mathcal{S}_\delta$. When B is in the second game then the shares of the protocol are simulated because $\mathfrak{b}[\kappa] = 1$ and the second game is equivalent to the definition of $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{lc}\rangle$. Most importantly, note that the adversary $\mathsf{A}_o^{\mathcal{E}}$ modifying the memory locations does not affect the interaction with the hiding game. B always simulates the computations to learn the new value in $\mathcal{M}_0$ that it can put into $\mathcal{L}$ in the game. B succeeds thanks to $\mathsf{A}_o^{\mathcal{E}}$ that gives the coherent modification of the value together with the share modification to $\mathcal{M}_0$ and $\mathcal{M}$. Note that in case $\mathsf{A}_o^{\mathcal{E}}$ uses the extractor, this proof also covers the case where the hiding property is somehow invalidated by the extraction. □

The same proof can be generalised to storage domains with private setup parameters if the environment can be simulated inside the hiding games, meaning that all sharing operations inside the environment are carried out inside the hiding game with $\mathfrak{b}[\kappa] = 0$. As before, this is possible if the Π_{e} does not reveal setup parameters that would invalidate the hiding property. For the adversary A_{lc}, the memory $\mathcal{M}$ has become meaningless, thus define the limited control execution model $\mathfrak{F}_6$ from $\mathfrak{F}_5$ by removing connections between $\mathcal{M}_0$ and $\mathcal{M}$, including $\mathcal{S}_u^*$. This setup is shown in Figure 12.

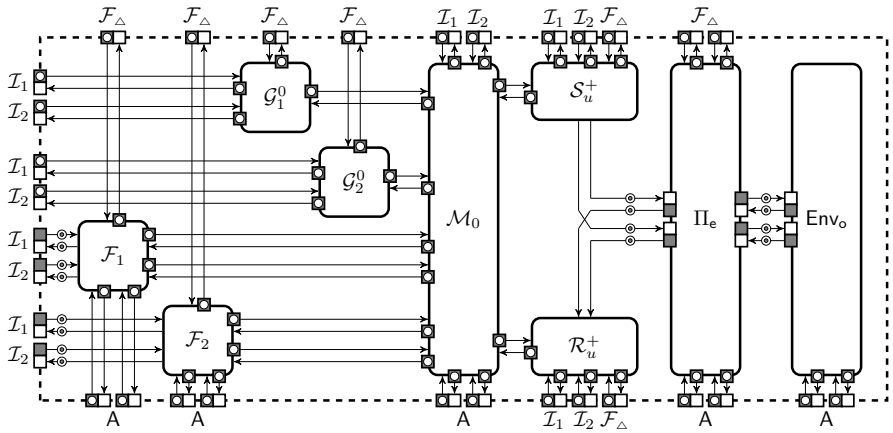


Figure 12. Limited control execution model for two-party protocols.

Lemma 11. *Then* $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{lc}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_6, \mathsf{A}_{lc}\rangle$ *for any* $\mathsf{A}_{lc} \in \mathbb{A}_{lc}$.

Proof. Trivial as A_{lc} does not communicate with the memory $\mathcal{M}$ or $\mathcal{S}_u^*$. □

Corollary 4. *Any coherent adversary against the hybrid model can be transformed to an equivalent coherent limited control adversary against the same protocol in $\mathfrak{F}_6$ for hiding and modification aware storage domains and for well-formed protocols that are in a canonical form and use meaningful transparent local operations.*

Proof. We work with a coherent adversary which works well as a generic adversary (Lemma 1). In Lemma 2 and Corollary 1, we show that if a protocol is robust against malformed inputs then any generic adversary is equivalent to semi-simplistic adversary. In Lemma 3 and Theorem 4, we show equivalence of semi-simplistic and lazy semi-simplistic adversary if all functionalities have tight scheduling and the protocol is secure against rushing. In Theorem 6 and Corollary 2 we conclude that it is sufficient to consider only simplistic adversaries when running well-formed programs. We then show that we can separate the value and share memories (Theorems 8 and 9), limit their interactions (Theorems 10 and 11). Assuming output isolation we continue the simplifications in Lemmas 8–10 to move from real shares to simulated shares. Finally, in Lemma 11 we conclude that it suffices to consider the limited control execution model. □

3.3.6. From Limited Control to the Hybrid Model

Lemma 12. *Any coherent limited control adversary against $\mathfrak{F}_6$ can be transformed to an equivalent coherent simplistic adversary against the same protocol in $\mathfrak{F}_0$ for storage domains with limited control and negligible extraction failure, and for well-formed protocols that are in a canonical form and use meaningful transparent local operations.*

Proof. Theorems 8, 10 and 11 guarantee that $\mathfrak{F}_0$, $\mathfrak{F}_1$, $\mathfrak{F}_2$ and $\mathfrak{F}_3$ are observationally equivalent for any adversary for well-formed protocol specification in a canonical form, using meaningful local functionalities and storage with negligible extraction failure. However, for $\mathfrak{F}_3$ to $\mathfrak{F}_6$ we modified the adversary and have to consider how any coherent adversary against $\mathfrak{F}_6$ can be transformed to a simplistic coherent adversary against $\mathfrak{F}_3$.

Consider the adaptor machine that connects to the abstract adversary that only interacts with $\mathcal{M}_0$ and does clocking. If the protocol in it is using a storage domain with limited access, then all modifications to the values $\mathcal{M}_0$ can be translated to the values of the shares in $\mathcal{M}$ in $\mathfrak{F}_3$. Note that by definition the new adversary against $\mathfrak{F}_3$ reads only the values that are modified from the memory and only uses them to compute the modification. If the extraction has negligible failure then the values in $\mathcal{M}_0$ are the same in $\mathfrak{F}_3$ and $\mathfrak{F}_6$ after this change. The added memory $\mathcal{M}$ and changed interaction with Π_{e} do not change the view of the adversary or the values returned to Π_{e}. □

Corollary 5. *For all coherent limited control adversaries A_{lc} against the protocol Π^* in $\mathfrak{F}_6$ there exists a hybrid adversary $\phi^*(\mathsf{A}_{lc})$ such that $\mathsf{Env}\langle\Pi^*, \mathsf{A}_{lc}\rangle \equiv \mathsf{Env}\langle\Pi, \phi^*(\mathsf{A}_{lc})\rangle$ if the protocols are secure against rushing and malformed inputs, have well-formed specification, are in a canonical form, use meaningful local operations for deterministic functionalities, use storage domains with limited access and negligible extraction failure.*

Proof. Semi-simplistic and lazy semi-simplistic adversary is equivalent to the generic adversary as shown in Corollary 1 and Theorem 4 for protocols secure against rushing. Simplistic adversaries can be transformed to equivalent lazy semi-simplistic adversaries for the same well-formed protocol in the respective protocol description as shown in Theorem 7. Theorem 8 shows that storing values in $\mathcal{M}$ in aligned manner can be reversed to any memory layout and $\mathcal{G}_i$ can be merged to the interpreters. Lemma 12 shows that any limited control adversary can be transformed to equivalent simplistic adversary for the canonical protocol. Therefore, we have shown that any limited control adversary can be transformed to an equivalent adversary in the hybrid protocol. □

3.4. Abstract Model

Results from previous sections allow us to consider an execution model where the environment interacts with a system consisting of interpreters $\mathcal{I}_1, \ldots, \mathcal{I}_n$, a semi-protected memory module $\mathcal{M}_0$, a library of idealised functionalities $\mathcal{G}_1^0, \ldots, \mathcal{G}_m^0$ and $\mathcal{F}_1, \ldots, \mathcal{F}_k$ for local and global operations and modules $\mathcal{R}_u^+$ and $\mathcal{S}_u^+$ for storing and fetching values from the memory $\mathcal{M}_0$. The environment for the protocol in question consists of the rest of the computation represented by Π_{e} and outer $\mathsf{Env}_{\mathsf{o}}$ representing the rest of the world as in Figure 12. The left-hand side of the figure is completely stated in terms of abstract memory and public parameters whereas the right-hand side, especially Π_{e} is still a complex network of nodes exchanging shares.

In Appendix C, we discuss further means to simplify the collection $\mathfrak{F}_6$ for many cases where the adversary does not really need individual buffers to clock each input and output of $\mathcal{F}_p$. For example, this is allowed in the usual case where the order and execution of $\mathcal{F}_p$ is sequential within one protocol instance. These simplifications would also carry over to the abstract execution model.

3.4.1. Abstract Execution Environment

Consider a restricted class of environments $\mathbb{E}_{\mathsf{r}}$ where the inner environment Π_{e} is trivial, i.e., parent parties $\mathcal{P}_i^*$ just forward inputs and outputs between $\mathsf{Env}_{\mathsf{abs}}$ and the protocol Π that we are considering. Each instance of Π_{e} runs only a single instance of Π. All inputs provided by $\mathsf{Env}_{\mathsf{o}}$ are first shared in Π_{e} using $\mathcal{S}_\delta$ for the protection domain where these values are and then reconstructed by $\mathcal{R}_u^+$ in Π. The output of $\mathcal{S}_u^+$ is reconstructed by the relevant reconstruction functionality $\mathcal{R}_\delta$ in Π_{e} before it reaches $\mathsf{Env}_{\mathsf{o}}$. Therefore, the main component of $\mathsf{Env}_{\mathsf{r}}$ is $\mathsf{Env}_{\mathsf{o}}$ and Π_{e} is almost invisible.

For adversaries that do not read the state of corrupted parent nodes $\mathcal{P}_i^*$, we can simplify Π_{e} to $\mathfrak{F}_7$ as depicted in Figure 13. Machines $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ handle protocol inputs and outputs. Buffers between $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ allow instant sharing of their states. In $\mathfrak{F}_6$, $\mathcal{I}_i$ interacts with $\mathcal{S}_u^+$ only when it executes DMACALL or SEND instructions. As there is only one instance of Π for each instance of Π_{e}, we modify the interpreter $\mathcal{I}_i$ in $\mathfrak{F}_7$ so that it would send a message to $\mathcal{O}_{out}$. As the buffer is clocked by the adversary, the interpreter does not lose control and can carry out as usual. By definition the interpreter $\mathcal{I}_i$ expects an input from $\mathcal{R}_u^+$ to launch a new protocol instance or as a reply to a DMACALL instruction. In $\mathfrak{F}_7$, the machine $\mathcal{O}_{in}$ sends the corresponding inputs. No changes are made to $\mathsf{Env}_{\mathsf{o}}$ that still submits and receives plain values.

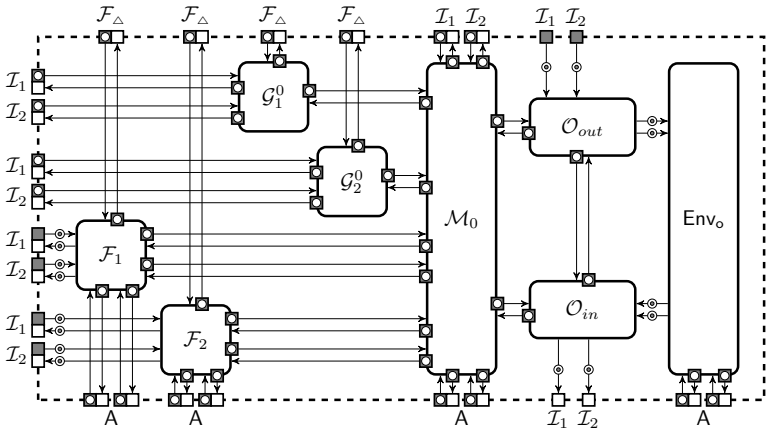


Figure 13. Abstract execution model for two-party protocols, $\mathfrak{F}_7$.

Machines $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ simulate the inner environment Π_{e} with instant clocking. The machine $\mathcal{O}_{out}$ collects incoming instructions and forwards DMACALL to $\mathcal{O}_{in}$ who immediately gives the control back. After that $\mathcal{O}_{out}$ treats the instruction as a message from $\mathcal{S}_u^+$ and simulates the reactions of $\mathcal{P}_i^*$ and $\mathcal{R}_\delta$. When $\mathcal{R}_\delta$ releases some values $\mathfrak{s}_0[t, \delta, \ell]$,

then $\mathcal{O}_{out}$ fetches them from the memory and sends to $\mathsf{Env_o}$. The machine $\mathcal{O}_{in}$ simulates the actions of $\mathcal{S}_\delta$, $\mathcal{P}_i^*$ and $\mathcal{R}_u^+$. Based on the inputs from $\mathcal{O}_{out}$, it knows what messages interpreters $\mathcal{I}_1, \ldots, \mathcal{I}_n$ expect and thus can assign correct memory locations to all messages sent to $\mathcal{R}_u^+$. When some message arrives to $\mathcal{R}_u^+$, $\mathcal{O}_{in}$ writes corresponding values to $\mathcal{M}_0$ and forwards the response of $\mathcal{R}_u^+$ to the correct interpreter. Fast clocking inside the simulation guarantees that $\mathcal{O}_{in}$ creates responses no later than in $\mathfrak{F}_6$, and $\mathcal{O}_{out}$ writes the outputs to $\mathsf{Env_o}$ as soon as all release instructions have arrived. As the adversary A clocks the buffer, A can reorder and delay protocol inputs exactly the same way as in $\mathfrak{F}_6$.

Lemma 13. *Let $\mathbb{A}$ be the class of adversaries against the collection $\mathfrak{F}_6$ that do not observe the state of corrupted parent nodes. Let $\mathbb{A}_*$ be the class of adversaries against $\mathfrak{F}_7$ and Π^* be the protocol. Then there exist transformations ϕ and ϕ^* such that*

$$\begin{aligned} &\forall \mathsf{Env} \in \mathbb{E}_\mathsf{r} : \quad \forall \mathsf{A} \in \mathbb{A} : && \mathsf{Env}\langle \Pi^*, \mathsf{A} \rangle \equiv \mathsf{Env}_*\langle \Pi^*, \phi(\mathsf{A}) \rangle \\ &\forall \mathsf{Env} \in \mathbb{E}_\mathsf{r} : \quad \forall \mathsf{A}_* \in \mathbb{A}_* : && \mathsf{Env}_*\langle \Pi^*, \mathsf{A}_* \rangle \equiv \mathsf{Env}\langle \Pi^*, \phi^*(\mathsf{A}_*) \rangle \end{aligned}$$

where Env *is the restricted environment in* $\mathfrak{F}_6$ *and* Env_* *is the corresponding environment in* $\mathfrak{F}_7$ *that contain the same* $\mathsf{Env_o}$.

Proof. First, convert the adversary A against $\mathfrak{F}_6$ to the adversary A_* against $\mathfrak{F}_7$. W.l.o.g. we can assume that the original adversary A immediately clocks buffers from $\mathcal{P}_i^*$ to $\mathcal{R}_u^+$ as this alters only in which order values will be reconstructed. The adversary A can correct the ordering and timing by clocking the leaky buffer between $\mathcal{R}_\delta$ and $\mathsf{Env_r}$. For the same reason, we can assume that A immediately clocks buffers from $\mathcal{S}_u^+$ to $\mathcal{P}_i^*$.

As adversary A_* clocks buffers from $\mathcal{I}_i$ to $\mathcal{O}_{out}$ instead of buffers from $\mathcal{S}_u^+$ to $\mathcal{P}_i^*$ and buffers from $\mathcal{O}_{in}$ to $\mathcal{I}_i$ instead of buffers from $\mathcal{P}_i^*$ to $\mathcal{R}_u^+$ then machines $\mathcal{O}_{out}$ and $\mathcal{O}_{in}$ carry out a perfect simulation. The claim follows as the transformation is reversible. □

In practical deployments, protocol inputs may come from different input parties. However, as we cannot rule out coalitions between input parties, we must include the class $\mathbb{E}_\mathsf{r}$ into the set of potential environment $\mathbb{E}$. Consequently, we have established that protocol can be secure only if it is secure in the abstract execution model defined as follows.

Definition 25. *Abstract execution environment* $\mathsf{Env_{abs}}$ *for a protocol* Π^* *is defined as the collection of* $\mathcal{O}_{in}$, $\mathcal{O}_{out}$ *and* $\mathsf{Env_o}$ *in* $\mathfrak{F}_7$. *Let* $\mathbb{E}_\mathsf{abs}$ *denote the set of all abstract environments.*

Abstract execution environment is very close to the minimal attack model. The adversary A gets setup information of the corrupted parties and learns the values of protocol inputs and outputs. Adversary A can also read and write some values in $\mathcal{M}_0$ and interact with ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ if there is corresponding interface. Finally, the adversary can influence the order of execution by clocking leaky buffers between the interpreters and other machines. Depending on the environment the adversary can learn additional information about protocol inputs and outputs but nothing more.

Definition 26. *An environment class $\mathbb{E}$ is embeddable into the abstract model for the class of adversaries $\mathbb{A}$ if there exist transformations $\phi : \mathbb{A} \to \mathbb{A}_*$ and $\psi : \mathbb{E} \to \mathbb{E}_{abs}$ such that*

$$\forall \mathsf{Env} \in \mathbb{E} : \quad \forall \mathsf{A} \in \mathbb{A} : \qquad \mathsf{Env}\langle \Pi^*, \mathsf{A} \rangle \equiv \psi(\mathsf{Env})\langle \Pi^*, \phi(\mathsf{A}) \rangle \ .$$

For most protection domains, it is quite easy albeit highly tiresome to prove that relevant environment classes are embeddable into the abstract model. In fact, we have used similar assumptions in Section 3.3 as the simulatability of the Π_e.

For embedding, we need to push all interactions between A and Π_e to $\mathsf{Env_o}$. The modified adversary $\phi(\mathsf{A})$ internally runs A and forwards all queries for Π_e to $\mathsf{Env_o}$ that internally simulates Π_e to provide correct responses. If all parameters generated by the

setup $\mathcal{F}_{\triangle}$ are public, then $\mathsf{Env}_{\mathsf{o}}$ can just redo all computations in Π_{e}. There is a small caveat as Env learns protocol outputs when all parties have sent them to $\mathcal{R}$. Thus, $\mathsf{Env}_{\mathsf{o}}$ must simulate initial shares of corrupted parties without knowing the secrets. The latter implies that storage domains for the protocol Π_{e} outputs must be hiding.

Similar simulation is possible for protocols with private setup parameters as long as $\mathsf{Env}_{\mathsf{abs}}$ can simulate interactions with Π_{e} knowing only protocol outputs and public parameters generated by $\mathcal{F}_{\triangle}$. In particular, note that the generic inner environment Π_{e} is a protocol for which all inputs are generated by $\mathcal{S}_{\delta}$ inside Π_{e} and $\mathcal{S}_u^+$ in Π^* and all outputs are processed by $\mathcal{R}_{\delta}$ when given to Env and $\mathcal{R}_u^+$ when given to Π^*. By placing the same restrictions to Π_{e} as to Π we can carry put exactly the same simplification operations as we did to arrive to Π^* from Π. As modification extractor and share simulator use only the public and corrupted parties parameters, the modified environment does not need parameters it cannot get.

3.4.2. Security in the Abstract Model

Throughout the paper we have transformed the initial protocol Π to the same protocol Π^* in the abstract execution environment. We call Π^* the abstraction of Π.

Definition 27 (Security in the abstract world)**.** *Let Π_1 and Π_2 be protocols and Π_1^* and Π_2^* their abstractions. Let $\mathbb{A}_1^*$ and $\mathbb{A}_2^*$ be the abstractions of classes of adversaries $\mathbb{A}_1$ and $\mathbb{A}_2$ against original protocols. Then, Π_1 is as secure as Π_2 in the abstract model if there exists a function $\rho^* : \mathbb{A}_1^* \to \mathbb{A}_2^*$ such that $\mathsf{Env}_{\mathsf{abs}}\langle\Pi_1^*, \mathsf{A}_1^*\rangle \equiv \mathsf{Env}_{\mathsf{abs}}\langle\Pi_2^*, \rho^*(\mathsf{A}_1^*)\rangle$ for all $\mathsf{A} \in \mathbb{A}_1^*$ and $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs}}$.*

By definition, security can be defined with respect to any other protocol Π_2. In practice, our goal is to prove that the protocol Π is as secure as some canonical ideal functionality $\mathcal{F}_0$. The simplifications from hybrid execution model to the abstract model can be applied to all protocols $\Pi\langle\mathcal{F}_1, \ldots, \mathcal{F}_2\rangle$. Similarly, they could be done for $\Pi_0\langle\mathcal{F}_0\rangle$ for the protocol Π_0 that only calls $\mathcal{F}_0$ once per instance and returns all results to Π_{e}. Note that Π_0 is output-isolated and the transformation is allowed as long as $\mathcal{F}_0$ satisfies the rules we have set for the canonical ideal functionalities inside the protection domain.

In the context of Section 2.3, our current transformations define ϕ_1 and the same ϕ_1 can be applied to adversaries against $\Pi_0\langle\mathcal{F}_0\rangle$. Therefore, in order to prove that $\Pi\langle\mathcal{F}_1, \ldots, \mathcal{F}_2\rangle$ is as secure as $\Pi_0\langle\mathcal{F}_0\rangle$ we can use ϕ_1 for both $\mathbb{A}_1$ and $\mathbb{A}_2$. In order to prove that $\Pi\langle\mathcal{F}_1, \ldots, \mathcal{F}_2\rangle$ is as secure as $\mathcal{F}_0$ we would need a similar transformation ϕ_2. However, note that for coherent adversaries the functionality $\mathcal{F}_0$ is equivalent to $\Pi_0\langle\mathcal{F}_0\rangle$, the details of this can be found in Appendix B. The main intuition is that $\Pi_0\langle\mathcal{F}_0\rangle$ has also the machines of the parties but coherent adversary corrupts them coherently with the parties in Π_{e} and does not gain any access that it does not have to just the $\mathcal{F}_0$ machine. Hence, combining the equivalence of $\mathcal{F}_0$ and $\Pi_0\langle\mathcal{F}_0\rangle$ with ϕ_1 forms the required transformation ϕ_2. It remains to argue that the security in the abstract model indeed suffices for the security in the hybrid execution model. The following theorem achieves this by putting the results of this paper into the relevant context.

Theorem 13. *Let $\mathcal{F}_0$ be ideal functionality and Π is a protocol constructed on top of a hybrid protection domain $\mathcal{F}_1, \ldots, \mathcal{F}_k$ with canonical (Definition 5) $\mathcal{F}_p$ with tight scheduling (Definition 12), meaningful (Definition 6) and transparent (Definition 23) local functionalities. The storage domains in the protection domain are hiding (Definition 2), modification-aware (Definition 3) and have limited control (Definition 4). If the protocol Π satisfies all abstraction assumptions:*

- *is well-formed (Definition 18) and in a canonical form (Definition 21),*
- *is robust against malformed inputs (Definition 14),*
- *is secure against rushing (Definition 16),*
- *is output-isolated (Definition 24)*

and the environment class $\mathbb{E}$ is embeddable into $\mathbb{E}_{\mathsf{abs}}$ for the class of adversaries $\mathbb{A}$ and $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}$ then security in the abstract model is necessary and sufficient for security in the hybrid model.

Proof. NECESSITY. Trivial Π_e that just forwards the inputs and outputs to and from Π and Env has to be a valid protocol. Therefore, $\mathbb{E}_{abs}$ is a valid class of adversaries against Π as they are formed by the trivial Π_e and a generic adversary in $\mathbb{E}$. If $\mathsf{Env}_{abs} \in \mathbb{E}$ then by security definition, the protocol must be secure against $\mathbb{E}_{abs}$ among all other environments and from Corollary 4 the protocol running with environment in $\mathbb{E}_{abs}$ in the hybrid model can be transformed to the abstract model.

SUFFICIENCY. Sufficiency in the abstract model results from the reversability of the series of transformations we have made from the hybrid to the abstract world. If all environments of interest are embeddable, then we can apply Lemma 13 showing equivalence of the abstract execution model and the limited control model. The rest of the sufficiency follows from Corollary 5 showing that we can move from limited control model to the hybrid model. Hence, in total a security proof in the abstract model can be translated to a security argument of the same protocol in the hybrid model. □

4. Discussion

With the transformations in Section 3 we have established the required transformations ϕ_1, ϕ_2 and ψ from Section 2.3 and shown their semi-inverses. Therefore, we have shown the abstract execution model and that it suffices to prove security in the abstract model when making a series of assumptions about the protection domain and the protocol.

On the side of the protection domain, we assume that all ideal functionalities are in a canonical form (Definition 5) and have tight scheduling (Definition 12), local functionalities are meaningful (Definition 6) and transparent (Definition 23), secure storage domains are hiding (Definition 2), modification-aware (Definition 3) and have limited control (Definition 4). These are mostly reasonable properties, however, they should be explicitly shown for the protection domain (the framework for programmable MPC) that is used. On the other hand, one can simply assume these properties as the requirements of their new algorithm for secure computation. If no other preconditions are made, then the algorithm can be securely implemented on any framework meeting these requirements. However, note that often it is reasonable to make additional requirements for the protection domain, for example, to specify the data types for which the algorithm works.

On the protocol side, we assume that it is well-formed (Definition 18) and in a canonical form (Definition 21), robust against malformed inputs (Definition 14), secure against rushing (Definition 16) and output-isolated (Definition 24). We have argued that some of these are natural or easily achievable requirements of the protocol or the concrete protocol implementation. Security against rushing is achievable for protocols where some honest party is always included in the subprotocols (Theorem 5). The main open question is the output isolation, which we have shown to hold for several special cases but for some protocols this property should be proven in addition to the security proof in the abstract model. Nevertheless, we have shown that secure protocols are output isolated (Theorem 12), thus such a proof must exist for all protocols both in order to use our framework as well as to prove security at all. In addition, most primitive functionalities return outputs as soon as they are computed and are therefore always output-isolated (Lemma 5).

We can consider the proposed sorting algorithm in Algorithm 2 as a concrete example of these properties and their use. This algorithm assumes that we have a protection domain that can shuffle values, compute comparisons and publish values. We expect all these to be represented as some canonical ideal functionalities. A common way to read this protocol is that parties execute these operations together. We also expect that the parties only start the computations once they have the input values and that they only write the outputs of the computations to the derived variables. In addition, they continue with the next instruction only when they have completed the previous one. Therefore, when reading this protocol description we already assume it to be executed so that it is well-formed. It is also in a canonical form because the conditional decision in the output is done based on a public value and we can assume that the implementation assures that

the memory is aligned. We do not usually think of robustness against malformed inputs, however, we indirectly assume that the inputs are correct and all errors from incorrect formats are handled by the computation implementation outside the algorithm itself. In secure multiparty computation, we are concerned with cases where some participant is honest, thus we also achieve security against rushing when executing this algorithm. The main problem with this algorithm is output isolation. It is not immediately clear that it is output isolated since the values $[\![k]\!]$ and $[\![m]\!]$ are computed several steps before they are returned. Hence, we should either modify the algorithm to include a step to re-randomise the secure representation of the values right before returning them or prove output-isolation independently.

The abstract model is quite restricted, the adversary can manage the timing of the ideal functionalities $\mathcal{F}_p$ used by the protocol Π as well as the input and output timing of Π. In addition, the adversary has access to the corrupted parties values in $\mathcal{M}_0$ and other access depending on the storage domain. For reasonable schemes, either we consider a passive adversary that does not modify $\mathcal{M}_0$ or we use a robust or verifiable scheme that ensures that the adversary has limited control over the shared values as long as the set of corrupted parties satisfies the bounds set by the storage domain. Therefore, this satisfies the part of the intuition that in security proofs of the protocol, the focus should be on the values that are made available to any party and can therefore also be seen by the adversary corrupting that party. Note that this also means that, in fact, the part of the protocol that needs to be analysed is the semantics of the published values and not really the cryptographic properties of the secure computation framework where the protocol is executed. One could say that we are left with security proofs without any cryptography.

If the protocol has one round, e.g., it receives some inputs and then gives outputs and finishes its work like common for protocols implementing arithmetic operations, then they are usually output-isolated. If, in addition, the execution of subprotocols in one instance of the protocol has sequential scheduling then the adversary can not alter the order in which the values are published. Therefore, being able to simulate all visible values is sufficient. Note that most protocols for secure computation fall into this category and, therefore, the intuitive proofs are easy to carry to formal proofs. However, if the scheduling is concurrent or there are many rounds, then at the very least the values should be simulated within the rounds in which they are computed and their order may depend on the scheduling. More details might be required depending on the model.

Author Contributions: Conceptualisation, S.L. and P.P.-R.; Methodology, S.L. and P.P.-R.; Writing—original draft, S.L. and P.P.-R.; Writing—review and editing, S.L. and P.P.-R. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Estonian Centre of Excellence in IT (EXCITE) and by the Estonian Personal Research Grants PRG49 and PRG920.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in article.

Conflicts of Interest: The authors declare no conflicts of interest.

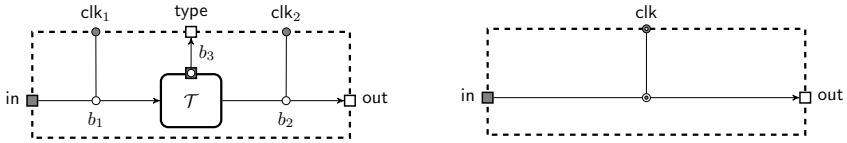
Abbreviations

The following abbreviations are used in this manuscript:

MDPI	Multidisciplinary Digital Publishing Institute
MPC	Secure multiparty computation
ABB	Arithmetic black box
RSIM	Reactive simulatability
UC	Universal composability
$[\![x]\!]$	Secret sharing of value x
Env	Environment
$\mathbb{E}$	Class of environments
A	Adversary
$\mathbb{A}$	Class of adversaries
Π	Protocol
Π_{e}	Inner environment representing computations in the secure computation framework
$\mathbb{P}$	Class of protocols
$\mathcal{P}_i$	Protocol participant i
$\mathcal{I}_i$	Code interpreter of $\mathcal{P}_i$
$\mathcal{Z}_i$	Corruption manager for $\mathcal{P}_i$
$\mathcal{F}_{\triangle}$	Secure setup functionality
$\mathcal{F}_p$	Ideal functionality
$\mathcal{F}_{pd}$	Ideal functionality of a protection domain
$\mathcal{F}_{\text{io}}$	Input–output functionality
$\mathcal{G}_{p,i}$	Local functionality p for $\mathcal{P}_i$
$\mathcal{G}_p^0$	Local functionality on values
$\mathcal{S}$	Secret sharing functionality
$\mathcal{R}$	Reconstruction functionality
$\mathcal{T}_{\mathcal{S}}$	Machine inside $\mathcal{F}_p$ that manages timing of $\mathcal{S}$
$\mathcal{T}_{\mathcal{R}}$	Machine inside $\mathcal{F}_p$ that manages timing of $\mathcal{S}$
$\mathcal{T}_{\mathcal{M}}$	Machine inside $\mathcal{F}_p$ that manages access to $\mathcal{M}$
$\mathcal{E}$	Extractor
$\mathcal{L}$	Storage of values in a storage domain
$\mathcal{L}^*$	Storage of shares in a storage domain
$\mathcal{M}$	Memory
$\mathcal{M}_0$	Memory holding only values
$\mathfrak{s}$	Internal state of the protocol participant
$\mathfrak{gs}$	Global state combining $\mathfrak{s}$ of all participants
$\mathfrak{s}_0$	State kept in $\mathcal{M}_0$
m	Protocol message
ϕ	Transformations of the adversary
ψ	Transformations of the environment
ρ	Transformation defined in the security proof
δ	Storage domain called δ
$\mathcal{A}_\delta$	Adversary structure for storage domain δ
$\ominus_\delta$	Modification operator in storage domain δ
$\perp$	Failure symbol denoting invalid values
$\mathfrak{F}_i$	Configuration i

Appendix A. Buffers Leaking Message Annotations

We specify leaky buffers using standard RSIM components. The hearth of the construction is a tag leaking machine $\mathcal{T}$ and three buffers for input, output and leakage, see Figure A1. The machine $\mathcal{T}$ accepts pairs of strings as inputs. When $\mathcal{T}$ receives (m, t) from the buffer b_1, the pair (m, t) is written to an output buffer b_2 and the annotation t is written to the sender-clocked buffer b_3. There are two ports for clocking the leaky buffer. The first port clk_1 determines when the annotation arrives to b_3. The second port clk_2 controls when and in which order message pairs (m, t) are written to the port out.

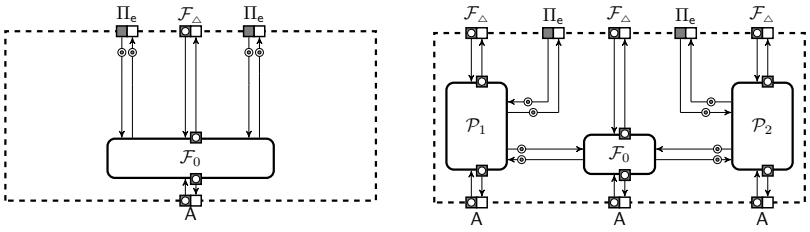


(**a**) Implementation (**b**) Notation

Figure A1. Collection implementing a leaky buffer and the corresponding notation.

Appendix B. Two Ways to Specify Ideal Functionalities

Two equivalent ways to depict ideal functionality are given in Figure A2. Commonly, one specifies the ideal functionality as a single machine $\mathcal{F}_0$. Alternatively, we can specify the ideal functionality as a special case of the hybrid protocol execution from Section 3.1.1 with just one ideal functionality The environment provides inputs to parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ who forward these directly to $\mathcal{F}_0$. Whenever $\mathcal{F}_0$ sends a message to $\mathcal{P}_i$, it forwards it to the environment. Let $\mathfrak{F}_1$ and $\mathfrak{F}_2$ denote the these alternative configurations.



(**a**) $\mathfrak{F}_1$ (**b**) $\mathfrak{F}_2$

Figure A2. Two alternatives for specifying ideal functionality.

Lemma A1. *For coherent adversaries configurations $\mathfrak{F}_1$ and $\mathfrak{F}_2$ in Figure A2 are equivalent.*

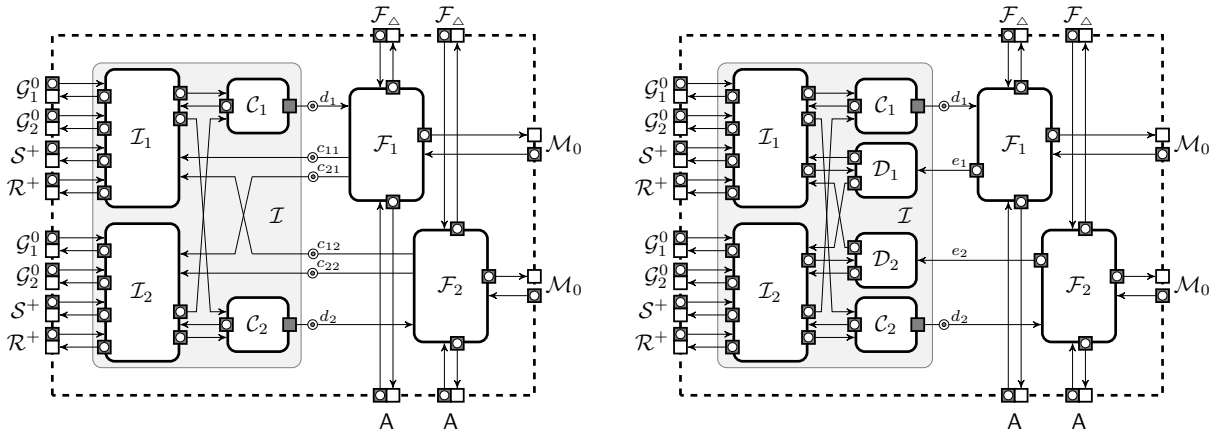
Proof. In both configurations, $\mathcal{F}_0$ gets exactly the same inputs including tags and the adversary controls when $\mathcal{F}_0$ receives its inputs. However, the message travels through two leaky buffers in the configuration $\mathfrak{F}_2$ instead of one in $\mathfrak{F}_1$. Adaptation of an adversary A_1 from $\mathfrak{F}_1$ to $\mathfrak{F}_2$ is straightforward. The new adversary clocks buffers between $\mathcal{P}_i$ and $\mathcal{F}_0$ as soon as possible and then clocks buffers to Π_* the same as A_1.

It is straightforward to simulate clocking for A_2 in $\mathfrak{F}_1$. However, in $\mathfrak{F}_1$ there is no way to corrupt the party directly in $\mathcal{F}_0$. For coherent adversary corrupting $\mathcal{P}_i$ in $\mathfrak{F}_2$ also means corrupting $\mathcal{P}_i^*$ in Π_*, hence any modifications can be done in $\mathcal{P}_i^*$ in $\mathfrak{F}_1$. □

Appendix C. Combined Interpreter with Simplified Clocking

The new memory-isolated model makes many clocking signals redundant. Adversarial control over buffer clocking is necessary only if this allows to control the execution order for the protocols or provides a time slot to carry out adversarial actions. Therefore, we simplify the model further by replacing all interpreters $\mathcal{I}_i$ with a joint interpreter $\mathcal{I}$ that combines some buffers. The simplest construction is such where the interpreter $\mathcal{I}$ is just a collection of interpreters $\mathcal{I}_i$. For most protocol specifications, this model can be further simplified to the configuration depicted in the left of Figure A3 and most sequential protocol specifications to the configuration depicted on the right.

As the original $\mathcal{F}_p$ contains modules $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$ which combine and broadcast DMACALL-s and potentially interact with the adversary, we can extract machines $\mathcal{C}_p$ and $\mathcal{D}_p$ which only combine or broadcast DMACALL-s. We push these into the interpreter $\mathcal{I}$. The use of sender clocked buffers forces us to add dummy buffers for passing control from $\mathcal{C}_p$ to $\mathcal{I}_i$ and from $\mathcal{I}_i$ to $\mathcal{D}_p$. However, the corresponding changes are straightforward and guarantee equivalence for passive adversaries who ignore leaks.



(**a**) Introducing $\mathcal{C}_i$ (**b**) Introducing $\mathcal{D}_i$

Figure A3. Joining the interpreters by merging outgoing and incoming buffers.

In many cases, the adversary can simulate the leaks of $b_{i,p}$ and mimic the effect of multiple clockings with a single buffer. The adversary must always know what is the next DMACALL when $\mathcal{I}_i$ receives an input and when $\mathcal{C}_p$ inside $\mathcal{F}_p$ is going to start a new round of computations. This is clearly true for sequential protocol implementations, but it also holds for many concurrent implementations. We can introduce $\mathcal{D}_p$ if we additionally show that the outcome of the execution cannot be influenced by clockings of $c_{i,p}$. This is evident for sequential protocol implementations, as only a single subprotocol instance is active at all times and thus delays in clockings just pause the protocol execution.

References

1. Bogdanov, D.; Laur, S.; Willemson, J. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Lecture Notes in Computer Science, Proceedings of the Computer Security—ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, 6–8 October 2008*; Jajodia, S., López, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5283, pp. 192–206. [CrossRef]
2. Damgård, I.; Pastro, V.; Smart, N.P.; Zakarias, S. Multiparty Computation from Somewhat Homomorphic Encryption. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2012—32nd Annual Cryptology Conference, Santa Barbara, CA, USA, 19–23 August 2012*; Safavi-Naini, R., Canetti, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7417, pp. 643–662. [CrossRef]
3. Bogdanov, D. Sharemind: Programmable Secure Computations with Practical Applications. Ph.D. Thesis, University of Tartu, Tartu, Estonia, 2013.
4. Demmler, D.; Schneider, T.; Zohner, M. ABY—A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, 8–11 February 2015.
5. Alexandra Institute. FRESCO—A Framework for Efficient Secure Computation. Available online: http://github.com/aicis/fresco (accessed on 20 August 2021).
6. KU Leuven. SCALE-MAMBA Software. Available online: https://github.com/KULeuven-COSIC/SCALE-MAMBA (accessed on 20 August 2021).
7. Keller, M. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security—CCS'20, Virtual Event, 9–13 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1575–1590. [CrossRef]
8. Bogetoft, P.; Christensen, D.L.; Damgård, I.; Geisler, M.; Jakobsen, T.P.; Krøigaard, M.; Nielsen, J.D.; Nielsen, J.B.; Nielsen, K.; Pagter, J.; et al. Secure Multiparty Computation Goes Live. In *Lecture Notes in Computer Science, Proceedings of the Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, 23–26 February 2009*; Revised Selected Papers; Dingledine, R., Golle, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5628, pp. 325–343. [CrossRef]
9. Mohassel, P.; Zhang, Y. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In Proceedings of the 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, 22–26 May 2017; pp. 19–38. [CrossRef]
10. Bogdanov, D.; Kamm, L.; Kubo, B.; Rebane, R.; Sokk, V.; Talviste, R. Students and Taxes: A Privacy-Preserving Study Using Secure Computation. *PoPETs* **2016**, *2016*, 117–135. [CrossRef]
11. Archer, D.W.; Bogdanov, D.; Lindell, Y.; Kamm, L.; Nielsen, K.; Pagter, J.I.; Smart, N.P.; Wright, R.N. From Keys to Databases—Real-World Applications of Secure Multi-Party Computation. *Comput. J.* **2018**, *61*, 1749–1771. [CrossRef]

12. Mohassel, P.; Rindal, P. ABY^3: A Mixed Protocol Framework for Machine Learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018; Lie, D., Mannan, M., Backes, M., Wang, X., Eds.; ACM: New York, NY, USA, 2018; pp. 35–52. [CrossRef]
13. Laud, P.; Pankova, A. Privacy-preserving record linkage in large databases using secure multiparty computation. *BMC Med. Genom.* **2018**, *11*, 35–55. [CrossRef] [PubMed]
14. Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In Proceedings of the 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, Las Vegas, NV, USA, 14–17 October 2001; pp. 136–145. [CrossRef]
15. Damgård, I.; Nielsen, J.B. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2003*; Boneh, D., Ed.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2729, pp. 247–264. [CrossRef]
16. Lipmaa, H.; Toft, T. Secure Equality and Greater-Than Tests with Sublinear Online Complexity. In *Lecture Notes in Computer Science, Proceedings of the Automata, Languages, and Programming—40th International Colloquium, ICALP 2013, Riga, Latvia, 8–12 July 2013*; Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7966, pp. 645–656. [CrossRef]
17. Escudero, D.; Ghosh, S.; Keller, M.; Rachuri, R.; Scholl, P. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2020—40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, 17–21 August 2020*; Micciancio, D., Ristenpart, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12171, pp. 823–852. [CrossRef]
18. Damgård, I.; Escudero, D.; Frederiksen, T.K.; Keller, M.; Scholl, P.; Volgushev, N. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In Proceedings of the 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, 19–23 May 2019; pp. 1102–1120. [CrossRef]
19. Kamm, L.; Willemson, J. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. *Int. J. Inf. Secur.* **2015**, *14*, 531–548. [CrossRef]
20. Veugen, T.; Abspoel, M. Secure integer division with a private divisor. *Proc. Priv. Enhancing Technol.* **2021**, *2021*, 339–349. [CrossRef]
21. Catrina, O.; de Hoogh, S. Improved Primitives for Secure Multiparty Integer Computation. In *Lecture Notes in Computer Science, Proceedings of the Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, 13–15 September 2010*; Garay, J.A., Prisco, R.D., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6280, pp. 182–199. [CrossRef]
22. Nishide, T.; Ohta, K. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *Lecture Notes in Computer Science, Proceedings of the Public Key Cryptography—PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, 16–20 April 2007*; Okamoto, T., Wang, X., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4450, pp. 343–360. [CrossRef]
23. Damgård, I.; Fitzi, M.; Kiltz, E.; Nielsen, J.B.; Toft, T. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *Lecture Notes in Computer Science, Proceedings of the Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, 4–7 March 2006*; Halevi, S., Rabin, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3876, pp. 285–304. [CrossRef]
24. Canetti, R.; Rabin, T. Universal Composition with Joint State. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2003*; Boneh, D., Ed.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2729, pp. 265–281. [CrossRef]
25. Pfitzmann, B.; Waidner, M. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In Proceedings of the 2001 IEEE Symposium on Security and Privacy—SP'01, Oakland, CA, USA, 14–16 May 2001; pp. 184–200.
26. Backes, M.; Pfitzmann, B.; Waidner, M. A General Composition Theorem for Secure Reactive Systems. In *Lecture Notes in Computer Science, Proceedings of the Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, 19–21 February 2004*; Naor, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2004; Volume 2951, pp. 336–354. [CrossRef]
27. Backes, M.; Pfitzmann, B.; Waidner, M. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.* **2007**, *205*, 1685–1720. [CrossRef]
28. Goldreich, O. *The Foundations of Cryptography—Volume 2: Basic Applications*; Cambridge University Press: Cambridge, UK, 2004; [CrossRef]
29. Canetti, R. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.* **2000**, *13*, 143–202. [CrossRef]
30. Micali, S.; Rogaway, P. Secure Computation (Abstract). In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO'91, 11th Annual International Cryptology Conference, Santa Barbara, CA, USA, 11–15 August 1991*; Feigenbaum, J., Ed.; Springer: Berlin/Heidelberg, Germany, 1991; Volume 576, pp. 392–404. [CrossRef]
31. Beaver, D. Secure Multiparty Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *J. Cryptol.* **1991**, *4*, 75–122. [CrossRef]
32. Canetti, R.; Feige, U.; Goldreich, O.; Naor, M. Adaptively Secure Multi-Party Computation. In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; Miller, G.L., Ed.; ACM: New York, NY, USA, 1996, pp. 639–648. [CrossRef]

33. Zikas, V.; Hauser, S.; Maurer, U.M. Realistic Failures in Secure Multi-party Computation. In *Lecture Notes in Computer Science, Proceedings of the Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, 15–17 March 2009*; Reingold, O., Ed.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5444, pp. 274–293. [CrossRef]
34. Gordon, S.D.; Katz, J. Complete Fairness in Multi-party Computation without an Honest Majority. In *Lecture Notes in Computer Science, Proceedings of the Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, 15–17 March 2009*; Reingold, O., Ed.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5444, pp. 19–35. [CrossRef]
35. Cohen, R.; Lindell, Y. Fairness Versus Guaranteed Output Delivery in Secure Multiparty Computation. *J. Cryptol.* **2017**, *30*, 1157–1186. [CrossRef]
36. Kiraz, M.; Schoenmakers, B. A protocol issue for the malicious case of Yao's garbled circuit construction. In Proceedings of the 27th Symposium on Information Theory in the Benelux, Noordwijk, The Netherlands, 8–9 June 2006; pp. 283–290.
37. Mohassel, P.; Franklin, M.K. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Lecture Notes in Computer Science, Proceedings of the Public Key Cryptography—PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, 24–26 April 2006*; Yung, M., Dodis, Y., Kiayias, A., Malkin, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3958, pp. 458–473. [CrossRef]
38. Aumann, Y.; Lindell, Y. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In *Lecture Notes in Computer Science, Proceedings of the Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, 21–24 February 2007*; Vadhan, S.P., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4392, pp. 137–156. [CrossRef]
39. Küsters, R.; Datta, A.; Mitchell, J.C.; Ramanathan, A. On the Relationships between Notions of Simulation-Based Security. *J. Cryptol.* **2008**, *21*, 492–546. [CrossRef]
40. Goyal, V.; Gupta, D.; Sahai, A. Concurrent Secure Computation via Non-Black Box Simulation. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2015—35th Annual Cryptology Conference, Santa Barbara, CA, USA, 16–20 August 2015*; Gennaro, R., Robshaw, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9216, pp. 23–42. [CrossRef]
41. Kiyoshima, S. Non-black-box Simulation in the Fully Concurrent Setting, Revisited. *J. Cryptol.* **2019**, *32*, 393–434. [CrossRef]
42. Pass, R. Simulation in Quasi-Polynomial Time, and Its Application to Protocol Composition. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, 4–8 May 2003*; Biham, E., Ed.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2656, pp. 160–176. [CrossRef]
43. Barak, B.; Sahai, A. How To Play Almost Any Mental Game Over The Net—Concurrent Composition via Super-Polynomial Simulation. In Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), Pittsburgh, PA, USA, 23–25 October 2005; pp. 543–552. [CrossRef]
44. Oren, Y. On the Cunning Power of Cheating Verifiers: Some Observations about Zero Knowledge Proofs (Extended Abstract). In Proceedings of the 28th Annual Symposium on Foundations of Computer Science, Los Angeles, CA, USA, 27–29 October 1987; pp. 462–471. [CrossRef]
45. Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. 2000. Available online: https://eprint.iacr.org/2000/067 (accessed on 20 August 2021).
46. Maurer, U.; Renner, R. Abstract Cryptography. In Proceedings of the Innovations in Computer Science—ICS 2011, Beijing, China, 7–9 January 2011; Chazelle, B., Ed.; Tsinghua University Press: Beijing, China, 2011; pp. 1–21.
47. Küsters, R. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In Proceedings of the 19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), Venice, Italy, 5–7 July 2006; pp. 309–320. [CrossRef]
48. Hofheinz, D.; Shoup, V. GNUC: A New Universal Composability Framework. *J. Cryptol.* **2015**, *28*, 423–508. [CrossRef]
49. Böhl, F.; Unruh, D. Symbolic universal composability. *J. Comput. Secur.* **2016**, *24*, 1–38. [CrossRef]
50. Camenisch, J.; Krenn, S.; Küsters, R.; Rausch, D. iUC: Flexible Universal Composability Made Simple. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—ASIACRYPT 2019—25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, 8–12 December 2019*; Galbraith, S.D., Moriai, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11923, pp. 191–221. [CrossRef]
51. Barak, B.; Canetti, R.; Lindell, Y.; Pass, R.; Rabin, T. Secure Computation Without Authentication. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2005*; Shoup, V., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3621, pp. 361–377. [CrossRef]
52. Canetti, R.; Cohen, A.; Lindell, Y. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2015—35th Annual Cryptology Conference, Santa Barbara, CA, USA, 16–20 August 2015*; Gennaro, R., Robshaw, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9216, pp. 3–22. [CrossRef]
53. Yao, A.C. Protocols for Secure Computations (Extended Abstract). In Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, Chicago, IL, USA, 3–5 November 1982; pp. 160–164. [CrossRef]
54. Beaver, D. Foundations of Secure Interactive Computing. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO'91, 11th Annual International Cryptology Conference, Santa Barbara, CA, USA, 11–15 August 1991*; Feigenbaum, J., Ed.; Springer: Berlin/Heidelberg, Germany, 1991; Volume 576, pp. 377–391. [CrossRef]

55. Bellare, M.; Rogaway, P. Robust Computational Secret Sharing and a Unified Account of Classical Secret-Sharing Goals. In Proceedings of the 14th ACM Conference on Computer and Communications Security—CCS'07, Alexandria, VA, USA, 2 November–31 October 2007; Association for Computing Machinery: New York, NY, USA, 2007; pp. 172–184. [CrossRef]
56. Damgård, I.; Nielsen, J.B. Adaptive versus static security in the UC model. In Proceedings of the International Conference on Provable Security, Hong Kong, China, 9–10 October 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 10–28.
57. Genkin, D.; Ishai, Y.; Prabhakaran, M.; Sahai, A.; Tromer, E. Circuits resilient to additive attacks with applications to secure computation. In Proceedings of the Symposium on Theory of Computing, STOC 2014, New York, NY, USA, 31 May–3 June 2014; Shmoys, D.B., Ed.; ACM: New York, NY, USA, 2014; pp. 495–504. [CrossRef]