Brandon Broadnax, Alexander Koch, Jeremias Mechler, Tobias Müller, Jörn Müller-Quade, and Matthias Nagel

Fortified Multi-Party Computation: Taking Advantage of Simple Secure Hardware Modules

Abstract: In practice, there are numerous settings where mutually distrusting parties need to perform distributed computations on their private inputs. For instance, participants in a first-price sealed-bid online auction do not want their bids to be disclosed. This problem can be addressed using secure multi-party computation (MPC), where parties can evaluate a publicly known function on their private inputs by executing a specific protocol that only reveals the correct output, but nothing else about the private inputs. Such distributed computations performed over the Internet are susceptible to remote hacks that may take place during the computation. As a consequence, sensitive data such as private bids may leak. All existing MPC protocols do not provide any protection against the consequences of such remote hacks.

We present the first MPC protocols that protect the remotely hacked parties' inputs and outputs from leaking. More specifically, unless the remote hack takes place before the party received its input or all parties are corrupted, a hacker is unable to learn the parties' inputs and outputs, and is also unable to modify them. We achieve these strong (privacy) guarantees by utilizing the fact that in practice parties may not be susceptible to remote attacks at every point in time, but only while they are online, i.e. able to receive messages.

To this end, we model communication via explicit channels. In particular, we introduce channels with an *air-gap switch* (disconnect-able by the party in control of the switch), and unidirectional *data diodes*. These channels and their isolation properties, together with very few, similarly simple and plausibly remotely unhackable hardware modules serve as the main ingredient for attaining such strong security guarantees. In order to formalize these strong guarantees, we propose the *UC with Fortified Security* (UC#) framework, a variant of the Universal Composability (UC) framework.

Keywords: universal composability, remotely unhackable hardware modules, security notions, isolation

DOI 10.2478/popets-2021-0072
Received 2021-02-28; revised 2021-06-15; accepted 2021-06-16.

1 Introduction

Secure multi-party computation (MPC), which allows mutually distrusting parties to securely evaluate a predefined function on their private inputs via a protocol, is a central privacy-enhancing technology that can be used in a wide range of scenarios. Examples include

- auctions (protecting the bids' confidentiality) [8],
- contact discovery (protecting the contacts' confidentiality) [35],
- toll collection (protecting e.g. location data) [21],
- medicine (protecting highly sensitive data such as (parts of) the genome) [3, 48],
- electronic payment (ensuring privacy and e.g. unlinkability) or [7],
- voting (e.g. confidentiality of the votes and correctness of the vote count) [46].

Protocol parties may become corrupted, i.e. fall under adversarial control. This can happen prior to the start of the protocol execution, which is called *static corruptions*. Furthermore, protocol parties may also become corrupted during the protocol execution via remote hacks. This type of corruption is called *adaptive corruptions*, first proposed in [13]. Up to now, all MPC constructions leak the private inputs of parties corrupted during protocol execution. However, the fact that the adversary *learns all secrets* of a corrupted party should be of a major concern, as there are plenty of sensitive MPC applications where this results in substantial real-world damage. In this work, we provide constructions where the adversary does not necessarily learn the inputs of parties corrupted via remote hacks during the protocol execution. Furthermore, this also holds for the corrupted parties' outputs.

As an example, consider two government agencies from different countries that want to perform a private set intersection to learn who is on both countries'

Alexander Koch, Jeremias Mechler, Jörn Müller-Quade: Competence Center for Applied Security Technology (KASTEL), Karlsruhe Institute of Technology (KIT)

wanted lists. In such a setting, it is very plausible that i) both parties can protect their devices against attacks requiring direct physical access but ii) are, at the same time, still susceptible to remote hacks from a nation-state attacker. Previous MPC protocols, even ones that are adaptively UC-secure, would not protect the remotely hacked party's inputs' and outputs' privacy and integrity. Surprisingly, such strong and novel guarantees are possible and provided by our constructions.

We achieve this by exploiting the so-far overlooked observation that in all likely real-world scenarios, protocol parties who are not connected to the Internet just cannot be remotely "hacked". For instance, a party may use *data diodes* (unidirectional channels) or disconnect itself via *air-gap switches*. With this, we can now introduce a distinction (motivated from the real world but with considerable theoretical interest) between remote hacks (e.g. sending malware), called *online attacks* in the following, and so-called *physical attacks* (e.g. replacing a part of the hardware or exploiting the larger attack surface provided by physical access to the device).

In conjunction with data diodes and air-gap switches, a protocol party may also use additional remotely unhackable hardware modules. An example of such an additional hardware module is a simple encryption unit that only implements a specific public key encryption scheme. All hardware modules used in our constructions are of very limited functionality and could be potentially formally verified for correctness, making it plausible to assume them to be resilient against remote hacking. In particular, an adversary can only corrupt such modules if he has direct physical access. Our assumptions are backed by commercially available devices with the required or similar functionality, see Section 5.

Utilizing only few and simple remotely unhackable hardware modules, we provide constructions with very strong security guarantees against online attacks. More specifically, online attacks i) mounted after a party received its first input and ii) mounted before a party received input if the attack comes from the "outside" do not allow an adversary to learn a corrupted party's inputs and outputs nor to *modify* them, unless all parties are corrupted. Here, the "outside" denotes all channels except one at a party's input port[1]. In more detail, the parties in our protocols are disconnected from the outside while waiting for input and can therefore not be corrupted via online attacks from the outside at that point. After receiving input (via the input port), the parties authenticate, mask and share their secrets in such a way that mounting online attacks gives the adversary control over a party but not the ability to learn the party's inputs or outputs, nor to modify them unless he gains control over all parties. This stands in contrast to adaptive corruptions where an adversary may learn and modify the inputs and outputs of corrupted parties after they received input. Although the possibility for parties to perform a secure erasure seems necessary for such a strong protection, we show that this assumption can be dropped in the full version. Also, the security of some of our constructions *gracefully degrades* to (essentially) standard adaptive UC security if the assumptions about the remotely unhackable hardware modules, channels and secure erasures turn out to be wrong (cf. Appendix A). This illustrates that security is not a binary property. We thus see our work as a first step towards a nuanced view of cryptographic security, which can be seen as a step towards the quantification of security.

In a bit more detail, we introduce a method for *fortifying* any generic MPC protocol[2] secure in the presence of adaptive corruptions, so that the resulting protocol provides the above-mentioned strong (privacy) guarantees. The constructions we present provide these guarantees even when arbitrary (possibly malicious) protocols are executed concurrently.

In order to adequately capture the guarantees provided by our remotely unhackable hardware modules in a concurrent setting, we propose *UC with Fortified Security*, a variant of the UC framework [11]. Like UC security, the notion of UC with Fortified Security is based on the simulation paradigm where the execution of the protocol under analysis is compared to an ideal protocol execution. In the ideal protocol, all (honest) parties do not communicate with each other but only hand their private inputs to a trusted third party which locally evaluates the desired function and privately sends the results back to the parties. The protocol under analysis, called the real protocol, is said to be secure if it "emulates" the ideal protocol, i.e. any attack that can be mounted in the real protocol can also be carried out in the ideal protocol. Stated differently, for any given adversary against the real protocol, there exists an adversary against the ideal protocol, called the *simulator*, that can cause the same damage in the ideal protocol

[1] We use the informal notion of an input port to denote the *single* channel via which a party can receive its first input.

[2] Informally, a *generic* MPC protocol such as [14, 29] can be used to securely realize practically *any* efficiently computable function.

as the given adversary can cause in the real protocol. As a consequence, all security guarantees of the ideal world such as privacy carry over to an execution of the protocol under analysis. This way, there is no need to define a security notion for each desired security property, as they are captured by the specification of the ideal protocol.

For conciseness, we introduce the abbreviation *UC#* for "UC with Fortified Security", where the added fence # symbolizes our "fortified" security. In this new variant of the UC framework, we deviate from the standard communication model and restrict the allowed communication in order to incorporate isolation properties. A party's current *online state* (capturing whether the party can receive messages) is determined by the type and state of its channels, e.g., the (opened/closed) state of its air-gap switches. This will be made precise in Definition 1. The hardware modules used in a protocol and their connections are part of what we call the *protocol architecture*.

These changes of the communication model make it necessary to re-prove the composition theorem as well as other properties of the UC framework. We can show that our notion is equivalent to standard adaptive UC security if no remotely unhackable hardware units and isolation assumptions are used.

We stress that our remotely unhackable hardware modules should not be confused with the tamper-proof hardware tokens proposed in [30], as our hardware modules are a substantially *weaker* assumption. In particular, they can be tampered with if one has direct physical access. They cannot be passed to other, possibly malicious parties, but are *only* used and trusted by their owner. They are thus not sufficient and not intended to be used to circumvent the impossibility results of [12, 15]. In summary, our protocols provide the best possible protection against online attacks in a setting where parties cannot be protected while waiting for input.

1.1 Our Protocols in a Nutshell

Our protocols proceed mainly according to the following template, which consists of three phases. In the initial phase, starting for each party after it obtains input, the parties go offline (including their input port) and send encrypted and signed shares of their inputs via data diodes. In this phase, the encrypted shares are received by hackable buffers since the parties are offline and hence cannot receive messages themselves. Being offline, they also cannot retrieve the public keys necessary for encrypting the shares. This is therefore done by a remotely unhackable encryption unit (connected to its party via a data diode), which encrypts and sends the shares to the other parties' buffers. At the end of this phase, each party erases each share except for its own and also erases the signing key used for authenticating their shares. Subsequently, a new phase begins where the parties are online and use the shares they received to obtain an encryption of the desired results. If an adversary corrupts a party via an online attack during this phase he only learns one share of each party's input and only an encryption of that party's output. Also, he is unable to modify the party's input since the signing key has been erased. Finally, the desired result is decrypted and output in a final phase. This is done by modularizing each party into a hackable "core component", which is referred to as the "party", and a remotely unhackable output interface module (OIM) that verifies and decrypts the desired result. In more detail, the core component sends two symmetric keys, one for encryption and one for message authentication, to its OIM in the first phase. These two keys are shared along with the input. In the second phase, the core component receives the desired result (from the generic MPC protocol) encrypted and authenticated with these two keys. In the final phase, the core component hands the encrypted and authenticated result to its OIM, which verifies the authenticity of the encrypted result and if correct, decrypts and outputs the result. Since the OIM is remotely unhackable and the two keys stored in the OIM are shared among the parties, the adversary would have to corrupt every party, i.e. every core component, in order to be able to learn and modify a party's output.

In order to highlight the necessity of remotely unhackable hardware modules and their isolation properties, we present two simple constructions in the following. Consider the execution of a secret-sharing-based MPC protocol such as [14] where the protocol parties P_1 and P_2 are initially honest. First, P_1 creates random shares s_1, s_2 of its input x_1, erases x_1 and sends s_2 to P_2 via a secure channel. At this point, the adversary remotely hacks P_1 and learns both s_1 and s_2 and is able to reconstruct the secret input x_1. This attack is not possible in our protocol as the machine holding s_1 is still offline at this point and not susceptible to remote hacks. This inability to protect the initially honest parties' secret in case of remote hacks is intrinsic to previous MPC protocols.

Second, consider a protocol for secure two-party computations where P_1 and P_2 have access to a trusted execution environment (TEE). In the protocol of [43],

P_1 and P_2 perform a remote attestation with the TEE, establishing a secure channel. This channel is used by the parties to send their inputs x_i $(i = 1, 2)$ to the TEE, which performs the actual computation and returns the results y_i $(i = 1, 2)$ via the secure channel to the parties. In this setting, an adversary then can remotely hack e.g. P_1 and thus learn or modify the outputs, as the TEEs protect the computation, but not the I/O. Furthermore, the adversary could also choose to remotely hack P_1 before it sends its input to the TEE. As P_1 has to be online in order to sends its input to the TEE, it is vulnerable at this point.

1.2 Our Contribution

We introduce the so-far overlooked distinction between physical and online attacks to protect the privacy-sensitive inputs and outputs of the participants in an MPC protocol against the latter, vastly more common type of attack. Furthermore, we utilize realistic simple remotely unhackable hardware modules that, to the best of our knowledge, have not been used for secure generic multi-party computation so far. With this, we also argue for a shift in how the security of the user's device is incorporated more concretely into secure computations.

Using only very few simple remotely unhackable hardware modules, we construct MPC protocols that provide very strong guarantees against online attacks. More specifically, an adversary mounting online attacks i) after a party received its first input and ii) before a party received input if the attack comes from the "outside", i.e. from all channels except one at a party's input port, is unable to

- learn a party's inputs and outputs (**Strong Privacy**)
- or to modify a party's inputs and outputs (**Strong Integrity**)

unless he gains control over *all* parties.

We present a construction for non-reactive functionalities (Theorem 1) using only two simple remotely unhackable hardware modules (apart from air-gap switches and data diodes) per party and a protocol for reactive functionalities (Theorem 3) that uses only one additional simple remotely unhackable hardware module. Both constructions can be proven secure in our framework for adversaries that gain control over all but one party and feature graceful degradation. We also present an augmentation of these constructions that allow simulation also in the case that all parties are under adversarial control (Theorems 2 and 4). For simplicity, we assume erasing parties. However, we show how this assumption can be dropped in the full version.

To properly formalize our above-mentioned guarantees, we propose a variant of the UC framework that adequately captures the advantages provided by remotely unhackable hardware modules. In this endeavor, we have to take additional care to obtain a composable security notion, with some modeling choices that will be discussed in Section 2. Our security notion is equivalent to adaptive UC security for protocols that do not use any remotely unhackable hardware modules. As a consequence, UC-secure protocols can be used as building blocks in our framework.

1.3 Related Work

Adaptive Security [13] captures security against adversaries that can corrupt parties at any point in the protocol. This notion has received considerable attention in the literature, see e.g. [14, 16, 27, 29]. In contrast to adaptive security where an adversary may learn all secrets of a corrupted party, we achieve that remotely hacking a party after it received its first input does not impact the confidentiality and integrity of a party's inputs and outputs, unless all parties have been corrupted.

Mobile Adversaries [4, 39], a notion strictly stronger than adaptive security, models an adversary taking over participants (in a way similar in spirit to our "remote hacks/virus attacks", although not modeling the online state) and possibly undoing the corruption later.

Concerning the used *trusted building blocks*, we assume data diodes, which are channels which allow for communication only in one specified direction. Garg et al. [23] analyze the cryptographic power of unidirectional channels as a building block, whereas we use unidirectional channels as a shield against dangerous incoming data packets. Achenbach et al. [1] make use of other trusted building blocks, such as a secure equality check module, to ensure the correct, UC-secure functioning of a parallel firewall setup in case of one malicious firewall.

Tamper-proof hardware tokens, first proposed by Katz [30], are an interesting research direction for finding plausible and minimal UC setup assumptions. Along this line of research, Goyal et al. [25] showed strong feasibility results of what can be done with these tokens. Moreover, Döttling et al. [18] showed that UC security is possible with a constant number of untrusted and resettable hardware tokens. Furthermore, [28] gives constructions of constant-round adaptively secure protocols which allow all parties to be corrupted. Most tokens considered in the literature require token functionalities that are custom and do not capture readily available hardware (sign-and-commit in [30], bit one-time memory in [25], combined zero-knowledge and signature verification in [18] and PRF-from-commitment in [28]). Such tokens' functionalities are, apart from input/output capability, of similar complexity when compared to our remotely unhackable hardware modules. However, the latter constitute a significantly weaker assumption, as they do not have to be physically tamper-proof and only need to be trusted by their owner. Also, none of the protocols aim to protect the parties' secrets in case of corruption.

Isolation is a general principle in IT security, with lots of research on isolation through virtualization, see e.g. [36]. Isolation in this way can be seen as a software analog of a trusted, remotely unhackable encryption module. Moreover, there is a wealth of literature on data exfiltration/side channel attacks to air-gaps including attacks based on acoustic, electromagnetic and thermal covert channels, cf. [55], which are not relevant to our work, as they are for protecting against outgoing communication from malicious internal parties, while we use data diodes/air-gap switches for the purpose of not being *hackable* from the outside. As an example for isolation, the Qubes OS provides strict separation between application domains, allowing to use an isolated GNU Privacy Guard (GPG) environment safely [47].

Trusted Execution Environments (TEEs) such as Intel SGX promise the existence of an incorruptible *secure enclave* where arbitrary computations can be performed in a secure and isolated way, even if the host system is compromised. Moreover, parties can establish a secure channel with a TEE to provide inputs and receive outputs of computations. Using *attestation*, a cryptographic proof for the identity of code running inside a TEE can be obtained. TEEs have been used in conjunction with a global common reference string to achieve composable generic multi-party computation [43]. The resulting protocol is practically efficient, as it essentially only requires protocol parties to perform attestation, send their inputs encrypted to the TEE and receive the encrypted output. All (complex) computations are then performed inside the TEE without additional cryptographic overhead. However, if the TEE is insecure, all security is lost as inputs and outputs will be exposed to the adversary. This is in stark contrast to classic distributed MPC protocols that at least protect inputs and outputs of honest parties in case of corruptions. Today's available implementations such as Intel SGX have very complex (closed-source) implementations and suffer from a number of vulnerabilities. In order to be considered *remotely unhackable*, the whole software stack exposed to the outside would have to be considered secure, in particular when handling possibly malicious messages. Due to the high complexity of current implementations, this may be unrealistic *per se*. In any case, it becomes less plausible with more complex code running inside the TEE, unless special precautions (such as formal verification or memory-safe languages) are taken.

Tinfoil Chat (TFC) is a peer-to-peer messaging system that uses multiple devices and data diodes to protect endpoints from remote hacks resp. their consequences (by using data diodes to block outgoing messages of machines that may have been remotely hacked) [40]. In contrast to previous frameworks such as the UC framework, which do e.g. not capture the isolation properties provided by data diodes, our framework is suitable to analyze the security of TFC. Compared to the novel security guarantees of our protocols, the security of TFC (for which there is no formal proof) is weaker as the *integrity* of the parties' outputs is not protected.

1.4 Outline

We start by explaining the framework that is suitable to capture our guarantees in Section 2, albeit in a simplified way that hides some of the subtleties and technicalities that are quick to arise in the UC setting, but are formally necessary to achieve composability of the security notion. Section 3 then states our main construction for the case of non-reactive functionalities, whereas the case of reactive functionalities is given in Section 4. For a discussion on graceful degradation, we refer to Appendix A. For the security proof of the presented construction, see Appendix B. In our exposition, we assume familiarity with the UC framework, but include a brief introduction to the UC framework in Appendix C.

2 Universal Composability with Fortified Security (UC#)

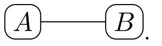
The advantages provided by unhackable hardware modules, e.g. the isolation properties of data diodes and air-gap switches, have not been considered in the context of secure multi-party computation before. The resulting challenge is twofold: First, the *framework* has to be able to express such properties adequately. Secondly, the *security notion* must be meaningful and exhibit properties such as universal composability. At the same time, both framework and security notion should be compatible with existing UC-secure protocols, allowing to re-use them without having to re-prove their security in the new framework.

Here, we give an overview of how we capture the advantages provided by remotely unhackable hardware modules, by explicitly modeling the connection topology between parties using standard channels, data diodes and air-gap switches, and their corresponding "online status". Finally, we introduce necessary trusted *interface modules* that handle a (honest) party's output, and describe our new security guarantees formally.

2.1 Network Topology, Channel Types and Online Status

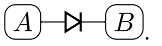
In our model, we make use of the following channels:

Standard Channels between two entities[3] allow ordinary bi-directional communication. They are depicted by a straight line connecting them:



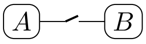
We say that *A and B are connected* (via a standard channel).

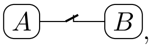
Data Diodes between two entities allow for communication in one direction only. For entities A and B, we depict this with a connection containing a diode sign in the direction of the permissible data flow, here from A to B:



We say that *A is connected to B via a data diode* (but not vice versa).

Air-gap Switches are channels between two entities that can be *connected* or *disconnected* once per activation by the entity that is in control of the switch. For entities A and B, where A is in control of the switch, we depict this with a switch sign, where the operating entity is next to the hinge:



or


On the left the switch is disconnected, while on the right it is connected. Disconnected air-gap switches allow no data transmissions at all, while connected ones allow bi-directional communication. In both cases, we say that *A is connected to B via an air-gap switch* (but not vice versa; and we may add that it is connected or disconnected).

We allow for the possibility to have multiple channels of different type between two entities. We say that *A can send messages* or provide input/output *to B* (via C) if there is a channel C between A and B that allows the communication from A to B. Note that if there is no channel between two entities, then no communication is allowed between them. Channels can be between (sub-)parties of a protocol or between a (sub-)party and an ideal functionality. In addition, channels can also be between a party and the environment or the adversary. Channels between a party and the environment model the allowed communication with calling parties (from other protocols). Channels between a party and the adversary model possible communication to the "outside world" that can be "delivered" by the adversary.

As in the UC framework, the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ may freely interact with each other. The same applies to the communication between $\mathcal{A}$ and ideal functionalities. Formally, we always assume standard channels between these *instances of interactive Turing machines* (ITIs). Communication between these ITIs is therefore independent of the given protocol architecture.

Online State of the Parties. We now define the *online state* of the parties via the possibility of receiving messages or input/output:

Definition 1 (Online State)**.** A (sub-)party P of protocol π is *online* (via C) if there is a channel C such that one of the following holds:

1. P can receive messages from the adversary via C.
2. P can receive output from a functionality $\mathcal{F}$ via C.
3. P can receive in- or output via C from a sub-party or calling party M, resp., and M is online via another channel C' between M and a party distinct from P.
4. P can receive input from the environment $\mathcal{Z}$ via C.

If there is no such channel C, we say that P is *offline*. Here, item 1 models that P can receive messages from the "outside world". Item 2 captures that messages can be received from a trusted third party $\mathcal{F}$ that "lives" somewhere in the outside world, such as a public bulletin board or a common reference string. Item 3 models a party being *transitively* online via connections to other parties who are online. Item 4 models a party being

[3] Interactive Turing machine instances such as (sub-)parties, ideal functionalities $\mathcal{F}$, the environment $\mathcal{Z}$, or the adversary $\mathcal{A}$.

transitively online via connections to a calling party (i.e. a party from another protocol which provides input).

Each time the adversary $\mathcal{A}$ is activated, he gets informed via which channels each party is online. This is called the *status*. As will be described in Section 2.3, $\mathcal{A}$ will be able to gain control over (hackable) parties when they are online.

2.2 The Protocol Architecture

The *protocol architecture* of a protocol π specifies the set of all channels involving the parties of π, together with the initial connection state of each air-gap switch, and for each (sub-)party in π whether it is (remotely) hackable or unhackable. Formally, the architecture of π is part of π's code.

In our *figures of protocol architectures*, main parties are represented by boxes with rounded corners, subparties by cornered ones, ideal functionalities are enclosed in a cloud, and the adversary is enclosed in a circle. Boxes with a double border denote that the corresponding (sub-)party is unhackable. Channels that end at the bottom are channels to the environment.[4]

Example 1 (Online State)**.** An example of a protocol architecture is given in Fig. 1 (left). There, the protocol starts with two main parties P_1, P_2, where P_1 and P_2 are initially online (P_1 and P_2 due to item 4 and P_2 also due to items 1 and 3), as well as online subparties Q_1, Q_2 (online due to item 2, Q_2 also due to item 3). We use the same figures to also depict later protocol states. Typically, a party P_1 will disconnect its air-gap switch to the environment $\mathcal{Z}$ as soon as it has received input (cf. Fig. 1 middle), making it *offline*. Later, P_1 may connect its air-gap switch to the adversary $\mathcal{A}$ at a specific point, say, after having erased its input (cf. Fig. 1 right), making it online again.

2.3 Corruption Model

As motivated above, we make use of the fact that in reality there is a significant difference between *remote hacks* (e.g. by an exploit that is executed upon a computer parsing some input message, such as an email with attached malware), and the much rarer and more difficult to perform *physical attacks*, where a computer's hardware is tampered with, or the adversary has physical access to the device. This allows us to provide stronger protection in the first case, whereas previous corruption models over-pessimistically assume the latter, hence resorting to less remaining privacy and security guarantees in cases of a corruption. This fits to our plausible assumption that some simple hardware modules can be implemented such that they cannot be remotely hacked. Thus, in our model, an adversary $\mathcal{A}$ is given the option to perform

Physical Attacks, where the targeted *main* party, and all its subparties, fall under adversarial control, as usual in adaptive security, and

Online Attacks, where only the targeted (sub-)party falls under adversarial control, but *only* if it is online *and not* assumed to be (remotely) *unhackable.*

If $\mathcal{A}$ has gained control over a (sub-)party P through one of these attacks, we say that P is *corrupted.* In this case, we (pessimistically) assume that $\mathcal{A}$ has access to P even if P disconnects its air-gap switches, by formally creating a new standard channel between P and $\mathcal{A}$.

In order to keep the proofs and discussions simple, we will restrict ourselves to a simplified adversarial model where physical attacks are only allowed prior to the start of the protocol. This is motivated by the fact that physical attacks (i.e. tampering with hardware) are time consuming and therefore typically must be mounted before the start of the protocol execution. However, our constructions for up to $N-1$ corrupted parties remain secure even if physical attacks are allowed throughout the execution, which we will only cover in the full version to keep the exposition simple.

In the following, we describe our new corruption model in more detail.[5] Let $\mathcal{P}$ be the set of main parties of a protocol π. At the first activation, the adversary $\mathcal{A}$ may only send a `physical-attack` instruction that enables him to gain control over parties regardless of the protocol architecture. Formally, $\mathcal{A}$ writes $(\texttt{physical-attack}, \mathcal{M})$, where $\mathcal{M} \subseteq \mathcal{P}$, on his outgoing message tape. Each $P \in \mathcal{M}$ and all of its sub-parties

[4] These figures capture interactive Turing machines (ITMs) only. When an ITM *instance* (ITI) is invoked, run-time information such as the session ID is accounted for when determining the channels between ITIs of the same protocol, the environment (resp. the calling protocol) and the adversary.

[5] Note that the following describes the behavior of protocol parties in the real model upon receiving corruption messages. As in the UC framework, in ideal protocols the behavior upon party corruption is determined by the ideal functionality.

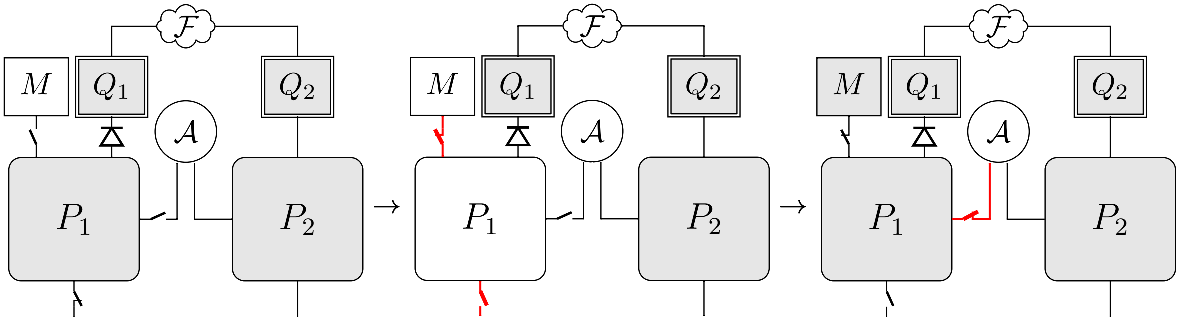


Fig. 1. Protocol architecture example *(left)*, depicted as described in Section 2.2. Online parties are in *gray*, and changes are in red. In step two *(middle)* P_1 disconnects from the environment $\mathcal{Z}$ (after having received its input), and connects to the machine M. In step three *(right)* P_1 connects to the adversary, causing P_1 (and now also M) to be online.

are then connected to the adversary via a standard channel and all air-gap switches controlled by and data diodes coming from these parties are replaced with standard channels. From then on, $\mathcal{A}$ has full control over all $P \in \mathcal{M}$ and *all* of their sub-parties.

From the second activation on, the adversary may not send a `physical-attack` instruction anymore. $\mathcal{A}$ may send `online-attack` instructions that enable $\mathcal{A}$ to gain control over *hackable* parties when they are online. Formally, if $\mathcal{A}$ writes $(\texttt{online-attack}, P)$ on his outgoing message tape and P is a (sub-)party of π that is online and hackable, then a standard channel between P and $\mathcal{A}$ is created and all air-gap switches controlled by P are connected. P then sends its entire local state to $\mathcal{A}$. From then on, $\mathcal{A}$ has full control over P. If P is *unhackable*, then this instruction is ignored.

Finally, if a (sub-)party P is corrupted, then each ideal functionality $\mathcal{F}$ which is connected to P is informed about P being corrupted through a special message $(\texttt{corrupt}, P)$ that is written on $\mathcal{F}$'s incoming message tape. Also, each main party immediately informs the environment after being corrupted.

2.4 Interface Modules

In order to achieve the strong security guarantees mentioned previously, a party's result of the MPC must remain unmodified and hidden from the adversary $\mathcal{A}$ even if the party is corrupted via an online attack *after* receiving input. This is not possible if a party learns its result and outputs it itself since $\mathcal{A}$ would learn this result if he corrupts the party and could then also instruct the party to output a modified value. Furthermore, for reactive tasks, a party corrupted after receiving input (via an online attack) must also not be able to learn or modify its input(s) for the rounds ≥ 2.

Deviating from the UC framework, we therefore allow the main parties to invoke special sub-parties called *interface modules* that are connected to their main party as well as to the environment via channels specified by the protocol architecture. These interface modules may thus give subroutine output to or receive input from the environment subject to the protocol architecture. Intuitively, interface modules model simple hardware modules connected to, e.g., a PC. During the protocol execution, a user does not trust his PC since it may have been remotely hacked (in particular, the output of his PC may have been altered by a hacker). Instead, he only trusts the unhackable interface modules and, in particular, the outputs given by them, e.g. via a display.

In our constructions, interface modules will be unhackable sub-parties with very limited functionality. We will assume an interface module called *output interface module* (OIM) that is used for ensuring that a party's result of the MPC remains unmodified and hidden from the adversary even in the case that the party is corrupted via an online attack after receiving input. More specifically, a party's result will only be learned by its OIM, which outputs the result instead of the party.

For reactive tasks, we will also assume an *input interface module* (IIM) that is used for ensuring that a party's input for the rounds ≥ 2 remain secret and unmodified even in the case that the party is corrupted after receiving (its first) input. In the ideal execution, the ideal functionality may also interact with dummy parties corresponding to interface modules (Definition 2).

2.5 Our Formal Security Notion

In this section, we give an *ideal-model protocol architecture* tailored to our security guarantees and define *fortified functionalities*. Recall that, as described in

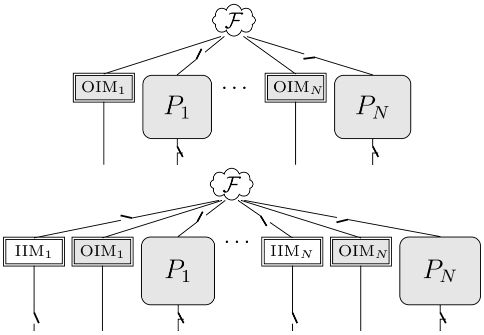


Fig. 2. Architecture of the ideal protocol $\texttt{AG}(\mathcal{F})$ for non-reactive functionalities *(top)* and for reactive functionalities *(bottom)*.

Section 2.3, $\mathcal{F}$ is informed through a special message $(\texttt{corrupt}, P)$, which is written on its incoming message tape, when a party P connected to $\mathcal{F}$ is corrupted. We now define the protocol architecture for our ideal protocol $\texttt{AG}(\mathcal{F})$, which stands for "Air-Gap switches" that are used there. It is defined differently dependent on whether $\mathcal{F}$ is *non-reactive* or *reactive*, i.e. whether $\mathcal{F}$ interacts with the parties in a single round, taking at most one input from each party and providing at most one output to each party, or whether it may receive inputs and provide outputs in multiple rounds (possibly keeping state between rounds), respectively.

For a non-reactive functionality $\mathcal{F}$, $\texttt{AG}(\mathcal{F})$ is the ideal protocol where N hackable "dummy main parties" $P_1, \ldots, P_N$ are connected to $\mathcal{F}$ via an initially disconnected air-gap switch and to the environment via an initially connected air-gap switch and additionally N unhackable "dummy output interface modules" $\text{OIM}_1, \ldots, \text{OIM}_N$ which are connected to $\mathcal{F}$ and the environment via standard channels, see also Fig. 2 (left).

Upon input v_i, each party P_i disconnects its air-gap switch to $\mathcal{Z}$, connects its air-gap switch to $\mathcal{F}$, and passes v_i to $\mathcal{F}$. Each P_i connects its air-gap switch to $\mathcal{Z}$ again upon receiving a special message $\texttt{connect}$ from $\mathcal{F}$. Furthermore, if $\mathcal{F}$ is *reactive*, $\texttt{AG}(\mathcal{F})$ additionally contains N unhackable "dummy input interface modules" $\text{IIM}_1, \ldots, \text{IIM}_N$ which are connected to $\mathcal{F}$ via initially disconnected air-gap switches controlled by $\mathcal{F}$ and to the environment via initially disconnected air-gap switches, see Fig. 2 (right) for reference. Each IIM_i connects its air-gap switch to the environment upon receiving $\texttt{connect}$ from $\mathcal{F}$, after $\mathcal{F}$ has connected its air-gap switch to IIM_i.

As the air-gap switch between a party P_i and $\mathcal{F}$ is disconnected before P_i has received input, the parties P_i in $\texttt{AG}(\mathcal{F})$ cannot be corrupted by an online attack "coming from the outside" (i.e., through channels except the input port) prior to receiving input. More specifically, each P_i can only be corrupted by an online attack prior to receiving input if it is online via its channel to the environment. In the following we will also refer to OIM_i (and IIM_i) as the "dummy OIM (resp. IIM) of P_i". Moreover, we call an ideal functionality $\mathcal{F}$ *standard* if $\mathcal{F}$ i) immediately notifies the adversary upon receiving input from an (honest) party, and ii) is standard corruption[6], and iii) only gives delayed outputs to parties.

2.5.1 Fortified Functionalities

In contrast to functionalities in the adaptive UC security model, *fortified functionalities* do *not* pass the inputs and outputs of a party P_i corrupted *after* receiving input to the adversary $\mathcal{A}$ and also do not allow him to modify P_i's input or the *output to P_i's dummy* OIM, unless all parties P_j $(j = 1, \ldots, N)$ are corrupted. $\mathcal{A}$ can only block an output or instruct the functionality to pass either the computed output or an error symbol $\perp$ to P_i's dummy OIM. If all parties are corrupted, $\mathcal{A}$ learns all inputs and outputs and may modify them arbitrarily (including the outputs to the dummy OIMs). This is formally captured in the following definition, where we omit session IDs for simplicity.

Definition 2 (Fortified Functionality)**.** Let $\mathcal{G}$ be a *non-reactive* standard ideal functionality interacting with N parties $P_1, \ldots, P_N$ and $\mathcal{A}$. Define the *fortified functionality* $[\mathcal{G}]$ of $\mathcal{G}$ interacting with $\mathcal{A}$, $P_1, \ldots, P_N$, and their dummy OIMs $\text{OIM}_1, \ldots, \text{OIM}_N$ as follows:

– $[\mathcal{G}]$ internally runs an instance of $\mathcal{G}$.
– $[\mathcal{G}]$ initializes a counter $c = 0$.
– Upon receiving input from P_i, $[\mathcal{G}]$ forwards it to $\mathcal{G}$.
– Each time $\mathcal{G}$ sends a notification to $\mathcal{A}$ upon receiving input from an (honest) party, $[\mathcal{G}]$ forwards that notification to $\mathcal{A}$.
– $[\mathcal{G}]$ forwards all delayed outputs of $\mathcal{G}$ for P_i to $\mathcal{A}$. Upon confirmation by $\mathcal{A}$, $[\mathcal{G}]$ forwards the output to OIM_i.
– Upon receiving $(\texttt{corrupt}, P_i)$, $[\mathcal{G}]$ increments c, marks P_i as corrupted before input if $[\mathcal{G}]$ has not

[6] It proceeds as follows upon receiving a $(\texttt{corrupt}, P)$ message from $\mathcal{A}$. First, $\mathcal{F}$ marks P as corrupted and outputs $\texttt{corrupted}$ to P. In the next activation, $\mathcal{F}$ sends to $\mathcal{A}$ all the inputs and outputs of P so far. In addition, from this point on, whenever $\mathcal{F}$ gets an input value v from P, it forwards v to $\mathcal{A}$, who may then send a "modified input value" v' that overwrites v. Also, all output values intended for P are sent to $\mathcal{A}$ instead.

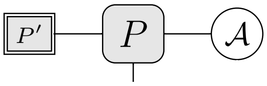
yet received input from P_i, *or* as corrupted after input, otherwise, and forwards $(\texttt{corrupt}, P_i)$ to $\mathcal{G}$. The subsequent output $(\texttt{corrupted})$ for P_i from $\mathcal{G}$ is forwarded to P_i.

- Handling parties marked as corrupted before input:
  - If $\mathcal{G}$ sends the input of P_i to $\mathcal{A}$, $[\mathcal{G}]$ forwards that input to $\mathcal{A}$. Furthermore, if $\mathcal{A}$ sends a modified input value for P_i, $[\mathcal{G}]$ forwards that value to $\mathcal{G}$.
  - If $\mathcal{G}$ sends an output intended for P_i to $\mathcal{A}$, $[\mathcal{G}]$ sends that output to $\mathcal{A}$. $\mathcal{A}$ may instruct $[\mathcal{G}]$ to pass any output of his choice to OIM_i.
- Handling parties marked as corrupted after input:
  - If $c < N$ and $\mathcal{G}$ sends the input of P_i to $\mathcal{A}$ after receiving $(\texttt{corrupt}, P_i)$, $[\mathcal{G}]$ ignores this message. Furthermore, if $\mathcal{A}$ sends a modified input value for P_i, ignore this value.
  - If $c < N$ and $\mathcal{G}$ sends the output intended for P_i to $\mathcal{A}$, $[\mathcal{G}]$ first notifies $\mathcal{A}$ that OIM_i is about to receive output. $\mathcal{A}$ may then instruct $[\mathcal{G}]$ to pass that output or $\perp$ to OIM_i.
- Upon reaching $c = N$, send all inputs and outputs to $\mathcal{A}$. $\mathcal{A}$ may then determine the outputs of all OIMs.
- All other messages between $\mathcal{A}$ and $\mathcal{G}$ are forwarded.
- If $\mathcal{A}$ sends $(\mathsf{output}, \tilde{y}, P_i)$, $[\mathcal{G}]$ outputs $\tilde{y}$ to P_i if $[\mathcal{G}]$ has marked P_i as corrupted. (While the adversary cannot modify the output of the dummy OIM of a party corrupted after receiving input, unless $c = N$, he is able to determine what a corrupted party itself outputs).

Fortified functionalities of reactive functionalities are defined in the full version. By construction, $\texttt{AG}([\mathcal{G}])$ captures our desired security goal: i) $[\mathcal{G}]$ ensures that corrupting a party P_i via an online attack *after* it has received its input does not enable the adversary to learn or modify P_i's input(s) and result(s) of the MPC (i.e. outputs of P_i's dummy OIM), unless *all* N parties are corrupted, and ii) the initially disconnected air-gap switches between the parties P_i and $[\mathcal{G}]$ ensure that the adversary cannot corrupt a party P_i via an online attack "coming from the outside" before P_i has received input, i.e. each P_i can only be corrupted via an online attack at that point if it is online via its channel to the environment.

2.5.2 Achieving a Meaningful Notion

In the UC framework, the adversary is not activated when a party provides input or receives subroutine output from a sub-party and is therefore not able to corrupt it during this communication. In our setting, this is undesirable as it does not capture the possibility of parties being hacked when they are online during this immediate communication. This can yield obviously insecure protocols that would be secure in our framework. To see this, consider a party P connected to $\mathcal{Z}$ and $\mathcal{A}$ as well as an unhackable sub-party P' via standard channels:



Upon receiving input, P sends secret data to P'. P' then sends a notification message to P who immediately *erases* all secret data after being activated again. As this message delivery is *immediate*, i.e. $\mathcal{A}$ is not activated during this communication, he is unable to corrupt P before P has erased its secret data even though P has been *online the entire time.* To address this problem, we introduce a *notify transport mechanism* in the full framework that activates the adversary (under certain conditions) upon immediate message delivery.

We can now define security in UC# in analogy to the UC framework:

Definition 3 (Emulation in the UC# Framework)**.** Let π, ϕ be protocols. π *emulates* ϕ *in the UC# framework*, denoted by $\pi \geq_{\mathrm{UC\#}} \phi$, if for every PPT adversary $\mathcal{A}$ there is a PPT adversary $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$ there is a negligible function negl such that for all $n \in \mathbb{N}, a \in \{0,1\}^*$ it holds that

$$\begin{aligned}|\mathsf{Pr}[\mathrm{Exec}_{\mathrm{UC\#}}\big(\pi, \mathcal{A}, \mathcal{Z}\big)(n,a) = 1] - \\ \mathsf{Pr}[\mathrm{Exec}_{\mathrm{UC\#}}\big(\phi, \mathcal{S}, \mathcal{Z}\big)(n,a) = 1]| \leq \mathsf{negl}(n),\end{aligned}$$

where $\mathrm{Exec}_{\mathrm{UC\#}}\big(\pi, \mathcal{A}, \mathcal{Z}\big)(n,a)$ denotes the random variable for the environment $\mathcal{Z}$'s output in the UC# execution experiment with protocol π and adversary $\mathcal{A}$ on input a and security parameter n.

Let π be a protocol with main parties $P_1, \ldots, P_N$. Then, π *emulates* ϕ *for up to* $L \leq N$ *parties under adversarial control* if emulation holds for all (real-model) PPT adversaries $\mathcal{A}$ corrupting $\leq L$ main parties of $P_1, \ldots, P_N$.

2.6 Properties of the Framework

As in the UC framework, the dummy adversary is complete in UC#. Our security notion is also transitive. UC# security is equivalent to adaptive UC security for "plain" UC protocols that are appropriately embedded into UC#, i.e., their security also holds in UC#. For the full properties and proofs, we refer to the full version.

Achieving Composability. Note that for universal composability, we have to introduce a few additional twists to the framework that allow us to prove the composition theorem. Let us elaborate. In the UC framework, the environment models all protocols that run concurrently with the protocol under analysis (challenge protocol). Naturally, the parties in the protocols incorporated by the environment also have online states that may change dynamically during the execution of these protocols. The online states of the parties incorporated by the environment may influence the online states of the parties in the challenge protocol. Therefore, the environment must be able to modify the online states of its channels to the challenge protocol depending on the online states of the parties it incorporates.

Consequently, we allow the environment to additionally *set* its channels to the parties to *activated* or *deactivated*. We therefore change item 4 in Definition 1 by stipulating that a party is online via a channel C if it can receive input from the environment via C and C has been set to "activated" by the environment. Additionally, we provide the environment, for each of its channels, the information whether it can currently receive output from that channel. With these changes, our security notion can be shown to be closed under protocol composition just like UC security. This is given in the full version.

3 Construction for Non-Reactive Functionalities

In this section, we present the construction $\Pi_{\mathcal{G}}$ for realizing the fortified functionality $[\mathcal{G}]$ of any non-reactive, standard, and adaptively well-formed[7] functionality $\mathcal{G}$. The broad idea is to have the parties $P_1, \dots, P_N$ send *encrypted shares* of their inputs via data diodes in an *offline sharing phase* and subsequently use these shares to compute the desired function in an *online compute phase*. This, however, cannot be done straightforwardly. To begin with, the parties are not able to retrieve public keys themselves in the sharing phase since this would necessitate going online, making them susceptible to online attacks. Therefore, each party P_i sends its shares to an unhackable sub-party called *encryption unit* (Enc unit) via a data diode. The Enc unit retrieves the public keys and sends encrypted shares to hackable sub-parties of the designated receivers, called *buffers*.

Furthermore, each message has to be authenticated so that the adversary cannot change the input of a party. One could do this with an additional unhackable "authentication unit" which signs each ciphertext or have the Enc unit sign all ciphertexts. However, since we want to use as few and as simple unhackable sub-parties as possible, we take a different approach. Each party P_i sends its shares together with valid *signatures* to its Enc unit. The verification key is sent, over an intermediary sub-party called *join* (J), to a hackable sub-party called *registration module* (RM) that disconnects itself from J after receiving input and forwards the verification key to a *public bulletin board* ($\mathcal{F}_{\mathsf{reg}}$) via a data diode. Once a party P_i has sent all of its shares, it erases everything except for its own share, its verification key and its decryption key. In order for this sign-then-encrypt approach to be secure, we assume that the PKE scheme is non-malleable (*IND-parallel-CCA-secure*) [6] and that the digital signature is unforgeable (*EUF-naCMA secure*) and also satisfies a property we call *length-normality*, guaranteeing that signatures of messages of equal length are also of equal length. The latter property prevents an adversary from learning information of plaintexts based on the length of their ciphertexts. Each party P_i is connected to its sub-party J via an *initially disconnected* air-gap switch in order to prevent the adversary from corrupting P_i's RM but not P_i *before* P_i has received its input.

In the compute phase, the adversary must be prevented from using values that are *different* from the shares sent by the honest parties to the corrupted parties in the sharing phase. Otherwise, he would be able to modify the inputs of the parties who were honest during the sharing phase. The parties P_i therefore not only use the shares they received but also the signatures of these shares and the registered verification keys during the compute phase. The result of the compute phase is a special "error symbol" if not all signatures are valid. Since the signing keys were erased at the end of the sharing phase, the adversary cannot generate new valid signatures for parties P_i corrupted after receiving input. He is also unable to revoke the verification key of such

[7] An ideal functionality is *adaptively well-formed* if it consists of a "shell" and a "core". The core is an arbitrary PPT TM. The shell is a TM that acts as a "wrapper": All incoming messages are forwarded to the core except for `corrupt` messages. Furthermore, outputs generated by the core are forwarded by the shell. Additionally, the shell sends the random tape of the core to the adversary if all parties are corrupted at some activation.

parties since this would require corrupting the respective RM, which is impossible since that party is offline.

Moreover, an adversary could *swap* a message in the sharing phase addressed to (the buffer of) an honest party P_j with a ciphertext of a share and signature received by a corrupted party (by encrypting that tuple with the respective public key). Furthermore, an adversary controlling at least two parties P_i, P_j knows two shares and valid signatures of each party and could use one of these tuples *twice* in the compute phase. To prevent these attacks, a party P_i signs each share *along with the designated receiver's PID*, which will be denoted by "P_j" in the following. In addition, a party P_i also includes its own PID in each message it sends to prevent the adversary from reusing messages sent by honest parties for parties corrupted before receiving input.

Finally, one cannot simply send the result of the compute phase to a party P_i since this would allow the adversary to learn and modify the output of the parties corrupted after receiving input. Instead, we use an unhackable *output interface module* (OIM). Each party P_i sends not only the shares of its input x_i but also shares of a *random pad* r_i and of a *MAC key* k_i in the sharing phase. Furthermore, each party P_i sends r_i and k_i to its OIM via a data diode. In the compute phase, the parties will then use these shares to compute $(y_i + r_i, \mathrm{Mac}(k_i, y_i + r_i))$, where y_i is the resulting output value (of party P_i) and $+$ denotes the bit-wise XOR. Each party then sends its result to its OIM, which will check authenticity by verifying the MAC tag and, if correct, reconstruct and output the value y_i.

In the following, we will take a modular approach (as facilitated by the composition theorem of UC#) and define – for an ideal functionality $\mathcal{G}$ representing an MPC – an ideal functionality $\mathcal{F}_{\mathcal{G}}$ that implements the verification of the input values in the compute phase as well as the subsequent multi-party computation (of $\mathcal{G}$) on the shares. Applying the UC# composition theorem, we are able to replace $\mathcal{F}_{\mathcal{G}}$ with an existing adaptively UC-secure protocol, e.g. [14]. Note that this will require an additional setup assumption, e.g. a common reference string, as our unhackable sub-parties, channels, and $\mathcal{F}_{\mathsf{reg}}$ are not UC#-complete.

Functionality 1 ($\mathcal{F}_{\mathcal{G}}$)**.** Let $\mathcal{G}$ be a *non-reactive* standard adaptively well-formed ideal functionality. $\mathcal{F}_{\mathcal{G}}$ proceeds as follows, running with parties $P_1, \dots, P_N$ and adversary $\mathcal{A}$ and parametrized with a digital signature scheme SIG and a message authentication code MAC.

1. Initialize the Boolean variable $\texttt{verify} = \texttt{true}$.
2. Upon receiving input from P_i, store it, mark P_i as $\textsf{input given}$ and send $(\texttt{received}, P_i)$ to $\mathcal{A}$. Ignore further input of P_i.

Consistency Check:

3. Once all parties are marked as $\textsf{input given}$, check if each stored input is of form $\overline{\mathsf{vk}}_i = (\mathsf{vk}_1^{(i)}, \dots, \mathsf{vk}_N^{(i)})$, $(s_{ji}, r_{ji}, k_{ji}, \sigma_{ji})$ $(j = 1, \dots, N)$. If not, set $\texttt{verify} = \texttt{false}$. Otherwise, check if $\overline{\mathsf{vk}}_1 = \cdots = \overline{\mathsf{vk}}_N$.
   (a) If the check fails, set $\texttt{verify} = \texttt{false}$.
   (b) Else, set $(\mathsf{vk}_1, \dots, \mathsf{vk}_N) = (\mathsf{vk}_1^{(1)}, \dots, \mathsf{vk}_N^{(1)})$. Check if $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, (P_i, s_{ji}, r_{ji}, k_{ji}), \sigma_{ji}) = 1$ for all $i, j \in \{1, \dots, N\}$,
      i. If the check fails, set $\texttt{verify} = \texttt{false}$.
      ii. Else, proceed with item 4.

Reconstruction and Computation:

4. For $i = 1, \dots, N$, compute $x_i = s_{i1} + s_{i2} + \cdots + s_{iN}$, $k_i = k_{i1} + k_{i2} + \cdots + k_{iN}$ and $r_i = r_{i1} + r_{i2} + \cdots + r_{iN}$.
5. Internally run $\mathcal{G}$ on input $(x_1, \dots, x_N)$. Let $(y_1, \dots, y_N)$ be the output of $\mathcal{G}$. For all $i = 1, \dots, N$, compute $o_i = y_i + r_i$ and $\theta_i \leftarrow \mathrm{Mac}(k_i, o_i)$.
6. If party P_i requests an output, proceed as follows:
   (i) If $\texttt{verify} = \texttt{false}$, send a private delayed output $\perp$ to P_i.
   (ii) Else, if item 5 has already been carried out, send a private delayed output (o_i, θ_i) to P_i.
7. $\mathcal{F}_{\mathcal{G}}$ is standard corruption (cf. Footnote 6). Once all parties are corrupted, $\mathcal{F}_{\mathcal{G}}$ sends its private randomness to $\mathcal{A}$. (This ensures that $\mathcal{F}_{\mathcal{G}}$ is also adaptively well-formed).
8. All other messages between $\mathcal{A}$ and $\mathcal{G}$ are ignored.

Let $\mathcal{F}_{\mathsf{reg}}$ be the public bulletin board functionality (cf. Appendix C.1 for a definition). Let $\mathrm{PKE} = (\mathrm{Gen}_{\mathrm{PKE}}, \mathrm{Enc}, \mathrm{Dec})$ be a public-key encryption scheme, $\mathrm{SIG} = (\mathrm{Gen}_{\mathrm{SIG}}, \mathrm{Sig}, \mathrm{Vrfy}_{\mathrm{SIG}})$ a digital signature scheme and $\mathrm{MAC} = (\mathrm{Gen}_{\mathrm{MAC}}, \mathrm{Mac}, \mathrm{Vrfy}_{\mathrm{MAC}})$ a message authentication code. Given a *non-reactive* standard adaptively well-formed functionality $\mathcal{G}$, we next define our protocol $\Pi_{\mathcal{G}}$ for realizing $\texttt{AG}([\mathcal{G}])$.

Construction 1. Define the protocol $\Pi_{\mathcal{G}}$ as follows: *Architecture:* See Fig. 3 for a graphical depiction. Figs. 4 and 5 show the changes of $\Pi_{\mathcal{G}}$'s state throughout the execution.

Offline Sharing Phase

Upon input x_i, each party P_i $(i = 1, \dots, N)$ does:

- *Disconnect* the air-gap switch to the environment.
- Generate $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$, $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$, $(\mathsf{sgk}_i, \mathsf{vk}_i) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$ and a random pad $r_i \leftarrow \{0,1\}^{p_i(n)}$, where $p_i(n)$ is an unspeci-

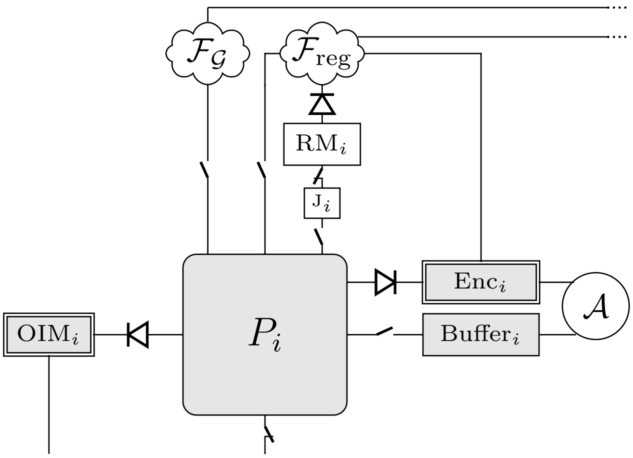


Fig. 3. Architecture of $\Pi_\mathcal{G}$. Each party P_i has 3 hackable sub-parties, called *buffer*, *registration module* (RM) and *join* (J), and 2 unhackable sub-parties, called Enc (unit) and OIM. Buffer and Enc unit are connected to the adversary via standard channels. All air-gap switches, except for P_i's air-gap switch to the environment and the RM's air-gap switch to J, are initially *disconnected*.

fied polynomial denoting the length of the one-time pad used for masking party P_i's output.

- Generate shares $s_{i1} + s_{i2} + \cdots + s_{iN} = x_i$ and $k_{i1} + k_{i2} + \cdots + k_{iN} = k_i$ and $r_{i1} + r_{i2} + \cdots + r_{iN} = r_i$.
- *Connect* the air-gap switch to J$_i$.
- Send (k_i, r_i) to OIM$_i$ and $(\mathsf{pk}_i, \mathsf{vk}_i)$ to J$_i$.
- Create signatures $\sigma_{ij} \leftarrow \text{Sig}(\mathsf{sgk}_i, (P_j, s_{ij}, r_{ij}, k_{ij}))$ $(j = 1, \ldots, N)$
- Send $(P_j, (P_i, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij}))$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ to Enc$_i$
- *Erase* everything except $s_{ii}, r_{ii}, k_{ii}, \sigma_{ii}$, vk_i and sk_i.

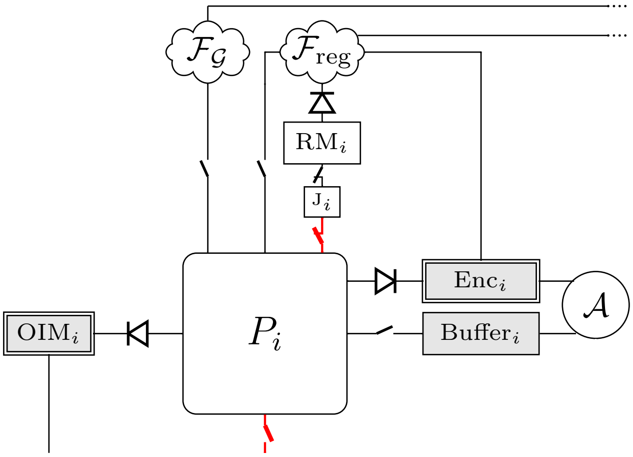


Fig. 4. $\Pi_\mathcal{G}$ after the offline sharing phase: P_i has disconnected its air-gap switch to the environment (making it offline) and connected its air-gap switch to J_i.

Registration module (RM$_i$) and J$_i$*:* On input $(\mathsf{pk}_i, \mathsf{vk}_i)$ to J$_i$, it forwards the input to *RM$_i$*. *RM$_i$* then *disconnects* the air-gap switch to J$_i$ and registers pk_i and vk_i by sending these keys to the public bulletin-board functionality $\mathcal{F}_{\mathsf{reg}}$.

Enc unit Enc$_i$*:* Receive a list $L = \{(P_j, v_j)\}_{j=\{1,\ldots,N\}\setminus\{i\}}$ from one's main party P_i. At each activation, for each $(P_j, v_j) \in L$, request pk_j of P_j from $\mathcal{F}_{\mathsf{reg}}$. If retrievable, compute $c_{ij} \leftarrow \text{Enc}(\mathsf{pk}_j, v_j)$, send (P_i, c_{ij})[8] to buffer of P_j and delete (P_j, v_j) from L. Then, go into idle mode.

Buffer: Store each message received. On input `retrieve`, send all stored messages to one's main party.

Online Compute Phase

After the sharing phase, a party P_i does the following:

- Connect air-gap switches to Buffer$_i$, $\mathcal{F}_{\mathsf{reg}}$ and $\mathcal{F}_\mathcal{G}$.
- Request all verification keys $\{\mathsf{vk}_l\}_{l\in\{1,\ldots,N\}\setminus\{i\}}$ from $\mathcal{F}_{\mathsf{reg}}$ registered by the other parties' RMs. If not all verification keys can be retrieved yet, go into idle mode and request again at the next activation.
- Send `retrieve` to Buffer$_i$ and check if it sends at least $N-1$ messages. If not, go into idle mode and, when activated again, send `retrieve` and check again. If yes, check if one has received from each party P_j a set $\mathcal{M}_j = \{(P_j, \tilde{c})\}$ with the following property $(*)$ *(Validity Check)*:
  There exists a tuple $(P_j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$ and a $(P_j, c) \in \mathcal{M}_j$ such that:
  - $\text{Dec}(\mathsf{sk}_i, c) = (P_j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$ *and* $\text{Vrfy}_{\text{SIG}}(\mathsf{vk}_j, (P_i, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}), \hat{\sigma}_{ji}) = 1$
  - For all $(P_j, \tilde{c}) \in \mathcal{M}_j$ either $\text{Dec}(\mathsf{sk}_i, \tilde{c}) = (P_j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$, or $(P_j, \tilde{c})$ is *"invalid"*, i.e., either decrypts to a tuple $(P_j, \tilde{s}_{ji}, \tilde{r}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}_{ji})$ such that $\text{Vrfy}_{\text{SIG}}(\mathsf{vk}_j, (P_i, \tilde{s}_{ji}, \tilde{r}_{ji}, \tilde{k}_{ji}), \tilde{\sigma}_{ji}) = 0$, or decrypts to a tuple $(P', \tilde{s}_{ji}, \tilde{r}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}_{ji})$ such that $P' \neq P_j$, or does not decrypt correctly.

  If $(*)$ does not hold, send $\perp$ to $\mathcal{F}_\mathcal{G}$. Else, send all retrieved verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ as well as all tuples $(\hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ as well as $(s_{ii}, r_{ii}, k_{ii}, \sigma_{ii})$ to $\mathcal{F}_\mathcal{G}$.

Online Output Phase

Having completed its last step in the compute phase, a party P_i requests output from $\mathcal{F}_\mathcal{G}$ and forwards that output to OIM$_i$.

[8] Sending the sender's PID as prefix is not necessary but simplifies the discussion. Note that for (P_i, c) we also say that "c is addressed as coming from party P_i".

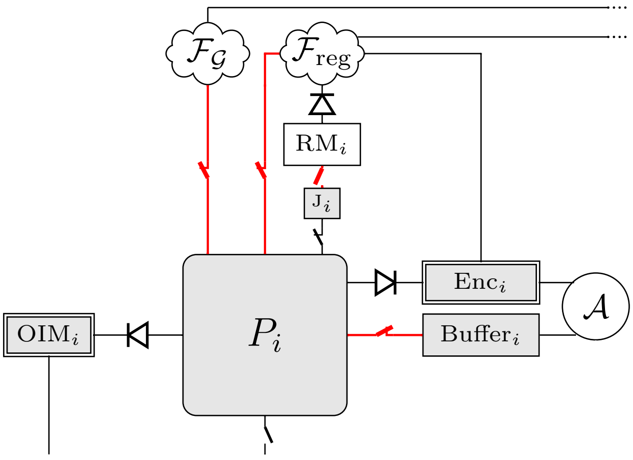


Fig. 5. $\Pi_{\mathcal{G}}$ during and after the online compute phase: First, RM$_i$ has disconnected its air-gap switch to J_i, keeping it offline. Then, P_i has connected its air-gap switches to $\mathcal{F}_{\mathcal{G}}$, $\mathcal{F}_{\text{reg}}$ and Buffer$_i$.

OIM$_i$: Store the first input (k_i, r_i) from one's main party. On second input (o_i, θ_i) or $\perp$ from one's main party, do the following: If the received value equals $\perp$, output $\perp$. Otherwise, check if $\text{Vrfy}_{\text{MAC}}(k_i, o_i, \theta_i) = 1$ and output $y_i = o_i + r_i$ if this holds, and $\perp$ otherwise.

Remark 1. Note that we do not model how to reuse modules such as *RM$_i$* that stay disconnected throughout the protocol execution. In practice, one may assume, e.g., a physical reset mechanism for these modules.

Theorem 1 (Up to $N - 1$ Corruptions, Non-Reactive)**.** Let $\mathcal{G}$ be a *non-reactive* standard adaptively well-formed functionality. Assume PKE is IND-pCCA-secure and SIG is EUF-naCMA-secure and length-normal, and MAC is EUF-1-CMA-secure. Then, for up to $N - 1$ parties under adversarial control, it holds that

$$\Pi_{\mathcal{G}} \geq_{\text{UC\#}} \texttt{AG}([\mathcal{G}]).$$

Proof. The proof will be given in Appendix B □

3.1 Up to N Corrupted Parties

One can augment Construction 1 to obtain a protocol $\Pi^{\mathsf{N}}_{\mathcal{G}}$ that is also secure if the adversary corrupts *all* parties at the expense of one additional unhackable subparty called *decryption unit* (Dec unit). Note that all other constructions given in this paper are summarized in Fig. 6. The main idea in the new construction is that parties do not decrypt ciphertexts themselves but send them to their Dec unit. Each Dec unit receives the secret key from its main party during the sharing phase.

In the compute phase, each Dec unit accepts a single vector of ciphertexts from its main party. Since the Dec units are unhackable and do not leak the secret keys, the simulator can report plaintext tuples to $\mathcal{Z}$ in such a way that the shares they contain are consistent with the parties' inputs and outputs even if all parties are corrupted.

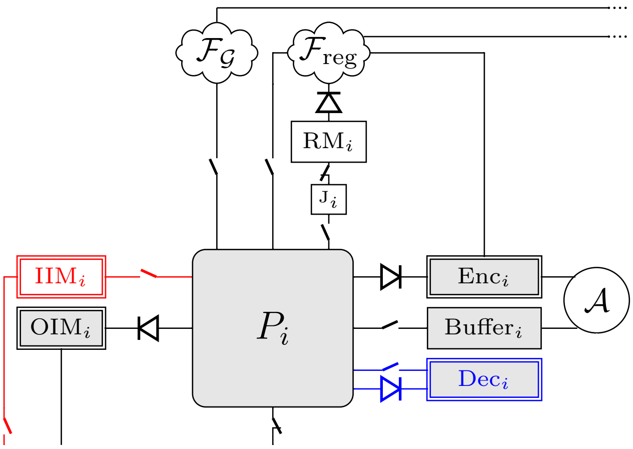


Fig. 6. Architecture for a) non-reactive functionalities and up to $N - 1$ parties under adversarial control, if red IIM/blue Dec unit parts are absent, b) non-reactive functionalities and up to N parties under adversarial control, if the blue Dec unit part is absent, c) reactive functionalities and up to $N - 1$ parties under adversarial control, if the red IIM part is absent, and d) reactive functionalities and up to N parties under adversarial control, if the red IIM/blue Dec unit parts are present.

Theorem 2 (Up to N Corruptions, Non-Reactive)**.** Let $\mathcal{G}$ be a *non-reactive* standard adaptively well-formed functionality. Assume PKE, SIG, MAC are as in Theorem 1. Then, for up to N parties under adversarial control, it holds that

$$\Pi^{\mathsf{N}}_{\mathcal{G}} \geq_{\text{UC\#}} \texttt{AG}([\mathcal{G}]).$$

4 Construction for Reactive Functionalities

For reactive functionalities, a new problem arises because parties are online after the first round. The input(s) for the next round(s) can therefore not just be given to a party since it may have been corrupted. We therefore need to find a way to insert the input(s) for round $u \geq 2$ into the protocol without allowing a party to learn or modify them.

To this end, we introduce an additional unhackable hardware module called *input interface module* (IIM) that acts as the counterpart of the OIM for inputs. Inputs for round(s) $u \geq 2$ have to be inserted into the protocol via the IIM which masks each input it receives and computes a MAC tag of the padded input. In the compute phase, these MAC tags are verified along with the signatures of the shares (of random pads for the inputs/outputs and of a MAC key) (cf. the full version for details). Denote by $\Pi_{\mathcal{G}}^{\mathsf{reac}}$ this new protocol.

Theorem 3 (Up to $N-1$ Corruptions, Reactive)**.** Let $\mathcal{G}$ be a reactive standard adaptively well-formed functionality. Let PKE and SIG be as in Theorem 1 and assume that MAC is EUF-CMA-secure. Then, for up to $N-1$ parties under adversarial control it holds that

$$\Pi_{\mathcal{G}}^{\mathsf{reac}} \geq_{\mathrm{UC}\#} \mathtt{AG}([\mathcal{G}]).$$

4.1 Up to N Corrupted Parties

With the same augmentation as described in Section 3.1, one can obtain a protocol $\Pi_{\mathcal{G}}^{\mathsf{N,reac}}$ that is also secure if the adversary corrupts *all* parties.

Theorem 4 (Up to N Corruptions, Reactive)**.** Let $\mathcal{G}$ be a reactive standard adaptively well-formed functionality. Let PKE, SIG, MAC be as in Theorem 3. Then, for up to N parties under adversarial control, we have

$$\Pi_{\mathcal{G}}^{\mathsf{N,reac}} \geq_{\mathrm{UC}\#} \mathtt{AG}([\mathcal{G}]).$$

5 Implementations of Remotely Unhackable Hardware Modules

In contrast to hardware that can be used as UC setup, e.g., tamper-proof hardware tokens or TEEs, our remotely unhackable hardware modules constitute a much weaker assumption as i) they only have to be trusted by their owner, ii) do not need to be sent to other parties and iii) only have to be secure against remote hacks, meaning that they do not have to be tamper-proof.

In order to protect from remotely exploitable vulnerabilities, we suggest to use formal verification wherever possible. There exists a vast body in the literature that is applicable to the implementations we discuss below, like smart cards ([2, 5]), cryptographic implementations in the IoT world [45], FPGAs and ASICs [9, 20] or microkernels [26, 31]. Due to the very low complexity of the cryptographic core, formal verification is applicable in practice for our modules.

Virtualization and TEEs. As a first attempt, we consider the case where some or all modules are implemented on the same machine in software. To this end, there are two main approaches. The first considers a hypervisor [26, 31] that implements each component in a dedicated virtual machine. Assuming that i) the hypervisor is secure and provides isolation between the individual VMs and the host and ii) the components are implemented correctly, the resulting system can be considered a secure implementation of our construction. When not using dedicated hardware, security can be strengthened by using a TEE such as Intel SGX to implement some of the components, in particular the Dec unit, the OIM and the IIM in case of reactive computations.

Data Diodes and Air-Gap Switches. In principle, data diodes and air-gap switches can be implemented in *software* using packet-filtering firewalls. However, this approach is susceptible to vulnerabilities in the packet filter and the associated software stack. There also exist numerous *commercial off-the-shelf hardware* solutions for data diodes (e.g. [22, 24, 44]) and air-gap switches [32] that can be controlled remotely. However, some are rather complex and not easily verifiable for correctness. In contrast, there exist numerous open-source *DIY implementations* (e.g. [40, 52]) for data diodes that exploit physical principles and thus require very little trust. The data diode proposed by [40] can be easily adapted to different communication technologies.

A remotely controlled air-gap switch could also be achieved by using a micro-controller to control a relay which (dis)connects one or several wires of an Ethernet connection (cf. [41]) or, e.g., a RS-232 connection. Alternatively, one could also use any manageable Ethernet switch or a non-managed switch combined with a remotely controllable power strip.

Enc and Dec Unit. In contrast to data diodes and air-gap switches where only correctness and no privacy is required, the Enc unit additionally has to keep the share it receives secret. Due to the use of public-key encryption schemes only, the Enc unit does not have to handle secret keys. Additional complexity is introduced by having to retrieve public keys from a public key infrastructure (PKI), for which a network interface is required. As long as the PKI's answers can be verified, e.g. by using certificates, the interface can be untrusted.

Due to its low complexity, the Enc unit can be implemented using *off-the-shelf hardware with low complexity* such as microcontrollers or single-board computers. While we are not aware of such hardware with formally verified or at least audited firmware, we believe that the risk originating from firmware vulnerabilities is acceptable in this case—especially if the programming interface is not exposed. At the OS layer, implementations in memory-safe language such as Tock [33] in Rust are available. Such a memory-safe language could also be used for the actual protocol implementation. As current *TEEs* such as Intel SGX communicate via the host OS, we do not consider them appropriate for the Enc unit, whose purpose is to keep the host system offline at the onset of the computation. Instead, it would be susceptible to remote hacks, contradicting the purpose of the Enc unit.

For the *Dec unit*, being very similar to *off-the-shelf hardware* like hardware security module (HSM) (e.g. [54]) or a smart card that can store private keys and perform decryptions, we believe that HSMs and smart cards are very natural candidates. This additionally assumes an appropriate interface that also enforces that only a single ciphertext vector is decrypted. There also exist a number of *open-source solutions* that can be adapted. An example is the NetHSM by Nitrokey [38], whose whole software is written in the memory-safe language OCaml. As OS, MirageOS with the formally verified microkernel Muen [26] is used. The security token Solo 2 [50] has an extensible open-source firmware, Trussed [51], written in Rust and can similarly be adjusted.

The Dec unit could also be implemented using a *field-programmable gate array* (FPGA). In our setting, the FPGA does not have to be tamper-proof, as our goal is to protect from the consequences of remote hacks without physical access to the device. This also holds for side-channels, unless they can be exploited via the connection to the FPGA. The code on the FPGA could be designed using a special-purpose software like Cryptol [20], which allows the generation of VHDL code which provably adheres to its specification. Also, the security of the FPGA firmware has to be considered. Recent attacks such as [19] have shown that caution is necessary. Of course, the user also has to trust that its code is correctly deployed and executed by the FPGA. A number of FPGA-based HSMs with open-source firmware are available [17, 42, 49], which could also serve as basis.

It is also conceivable to implement the Dec unit using a *TEE*. However, this approach has the drawback that current implementations are very complex, relying on a combination of software, firmware and hardware that have to be trusted by the user due to their closed-source nature and may all contain potential vulnerabilities or even backdoors. In particular, side-channel attacks [37] that allow the extraction of secrets from a TEE may be a major problem in our setting, assuming that the adversary is able to compromise the host system where the Dec unit is running.

Common to complex building blocks such as TEEs or FPGAs is the difficulty or even inability of the user to verify that the correct code is actually executed.

OIM and IIM. Remember that OIM and IIM accept one or several one-time pads and MAC keys and are responsible for the verification and decryption of results (OIM) resp. encrypting and authenticating inputs (IIM). As OIM and IIM only provide interfaces for input resp. output, no network capability is required. Thus, the host connection can be realized using some simple protocol such as RS-232 that can be safely implemented.

We can also extend an HSM, security token or FPGA with the required functionality and connect it via an appropriate interface that provides secure I/O, e.g. using a simple dot-matrix display and a keyboard. By augmenting TEEs with secure I/O, e.g. [34, 53] for Intel SGX, OIM and IIM can be in principle realized using a TEE. However, we believe that these approaches for TEEs with secure I/O should be considered proof-of-concepts not yet fit for practice.

6 Conclusion

When using MPC in practice, one has to protect well against the consequences of remote hacks. Utilizing only few and simple remotely unhackable hardware modules and their accompanying isolation assumptions, we constructed general MPC protocols providing very strong composable security guarantees against online attacks.

Some of our protocols even exhibit graceful degradation, which is a step towards more nuanced, quantifiable security.

Acknowledgements

Brandon Broadnax was funded by the Helmholtz Association's Supercomputing and Big Data program. Alexander Koch was supported by the Competence Center for Applied Security Technology (KASTEL). Jeremias Mechler was partially supported by funding of the Helmholtz Association (HGF) through the Competence

Center for Applied Security Technology (KASTEL). Matthias Nagel was supported by the German Federal Ministry of Education and Research within the framework of the project "Sicherheit vernetzter Infrastrukturen (SVI)" in the Competence Center for Applied Security Technology (KASTEL).

References

[1] D. Achenbach, J. Müller-Quade, and J. Rill. Universally composable firewall architectures using trusted hardware. In B. Ors and B. Preneel, editors, *BalkanCryptSec 2014*, volume 9024 of *LNCS*, pages 57–74. Springer, 2014. 10.1007/978-3-319-21356-9_5.

[2] J. Andronick, B. Chetali, and C. Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, volume 3582 of *LNCS*, pages 302–317. Springer, 2005. 10.1007/11526841_21.

[3] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin. Privacy-preserving search of similar patients in genomic data. *Proc. Priv. Enhancing Technol.*, 2018(4):104–124, 2018. 10.1515/popets-2018-0034.

[4] J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In M. M. Halldórsson and S. Dolev, editors, *PODC 2014*, pages 293–302. ACM, 2014. 10.1145/2611462.2611474.

[5] G. Barthe and G. Dufay. Formal methods for smartcard security. In A. Aldini, R. Gorrieri, and F. Martinelli, editors, *Foundations of Security Analysis and Design III, FOSAD 2004/2005 Tutorial Lectures*, volume 3655 of *LNCS*, pages 133–177. Springer, 2005. 10.1007/11554578_5.

[6] M. Bellare and A. Sahai. Non-malleable encryption: Equivalence between two notions, and an indistinguishability-based characterization. In M. J. Wiener, editor, *CRYPTO '99*, volume 1666 of *LNCS*, pages 519–536. Springer, 1999. 10.1007/3-540-48405-1_33.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy, SP 2014*, pages 459–474. IEEE Computer Society, 2014. 10.1109/SP.2014.36.

[8] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In R. Dingledine and P. Golle, editors, *Financial Cryptography and Data Security, FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, 2009. 10.1007/978-3-642-03549-4_20.

[9] T. Braibant and A. Chlipala. Formal verification of hardware synthesis. In N. Sharygina and H. Veith, editors, *Computer Aided Verification, CAV 2013*, volume 8044 of *LNCS*, pages 213–228. Springer, 2013. 10.1007/978-3-642-39799-8_14.

[10] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. Concurrently composable security with shielded super-polynomial simulators. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 351–381, 2017. 10.1007/978-3-319-56620-7_13.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145, 2001. 10.1109/SFCS.2001.959888.

[12] R. Canetti and M. Fischlin. Universally composable commitments. In J. Kilian, editor, *CRYPTO 2001*, pages 19–40. Springer, 2001. 10.1007/3-540-44647-8_2.

[13] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *STOC 1996*, pages 639–648, 1996. 10.1145/237814.238015.

[14] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In J. H. Reif, editor, *STOC 2002*, pages 494–503. ACM, 2002. 10.1145/509907.509980.

[15] R. Canetti, E. Kushilevitz, and Y. Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In E. Biham, editor, *EUROCRYPT 2003*, pages 68–86, 2003. 10.1007/3-540-39200-9_5.

[16] R. Canetti, O. Poburinnaya, and M. Venkitasubramaniam. Equivocating Yao: constant-round adaptively secure multiparty computation in the plain model. In H. Hatami, P. McKenzie, and V. King, editors, *STOC 2017*, pages 497–509. ACM, 2017. 10.1145/3055399.3055495.

[17] CrypTech. CrypTech Alpha. URL https://cryptech.is/.

[18] N. Döttling, T. Mie, J. Müller-Quade, and T. Nilges. Implementing resettable uc-functionalities with untrusted tamper-proof hardware-tokens. In A. Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 642–661. Springer, 2013. 10.1007/978-3-642-36594-2_36.

[19] M. Ender, A. Moradi, and C. Paar. The unpatchable silicon: A full break of the bitstream encryption of Xilinx 7-series FPGAs. In S. Capkun and F. Roesner, editors, *USENIX Security 2020*, pages 1803–1819. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/ender.

[20] L. Erkök, M. Carlsson, and A. Wick. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 188–191. IEEE, 2009. 10.1109/FMCAD.2009.5351121.

[21] V. Fetzer, M. Hoffmann, M. Nagel, A. Rupp, and R. Schwerdt. P4TC - provably-secure yet practical privacy-preserving toll collection. *Proc. Priv. Enhancing Technol.*, 2020(3): 62–152, 2020. 10.2478/popets-2020-0046.

[22] Fibersystem. Data diodes. URL https://www.fibersystem.com/product-category/data-diodes/.

[23] S. Garg, Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with one-way communication. In R. Gennaro and M. Robshaw, editors, *CRYPTO 2015*, volume 9216 of *LNCS*, pages 191–208. Springer, 2015. 10.1007/978-3-662-48000-7_10.

[24] genua. Data diode cyber-diode: High-security industrial monitoring of plants, machinery and critical infrastructure. URL https://www.genua.de/en/it-security-solutions/data-diode-cyber-diode.

[25] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In D. Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 308–326. Springer, 2010. 10.1007/978-3-642-11799-

2_19.

[26] I. Haque, D. D'Souza, H. P, A. Kundu, and G. Babu. Verification of a generative separation kernel. In D. V. Hung and O. Sokolsky, editors, *Automated Technology for Verification and Analysis, ATVA 2020*, volume 12302 of *LNCS*, pages 305–322. Springer, 2020. 10.1007/978-3-030-59152-6_17.

[27] C. Hazay, Y. Lindell, and A. Patra. Adaptively secure computation with partial erasures. In C. Georgiou and P. G. Spirakis, editors, *PODC 2015*, pages 291–300. ACM, 2015. 10.1145/2767386.2767400.

[28] C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam. Constant round adaptively secure protocols in the tamper-proof hardware model. In S. Fehr, editor, *PKC 2017*, volume 10175 of *LNCS*, pages 428–460. Springer, 2017. 10.1007/978-3-662-54388-7_15.

[29] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In D. A. Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, 2008. 10.1007/978-3-540-85174-5_32.

[30] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In M. Naor, editor, *EUROCRYPT 2007*, LNCS, pages 115–128. Springer, 2007. ISBN 978-3-540-72540-4. 10.1007/978-3-540-72540-4_7.

[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *ACM Symposium on Operating Systems Principles, SOSP 2009*, pages 207–220. ACM, 2009. 10.1145/1629575.1629596.

[32] L-com. Physical layer air gap network switches. URL https://www.l-com.com/secure-data-physical-layer-air-gap-network-switches.

[33] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kB computer safely and efficiently. In *Symposium on Operating Systems Principles, 2017*, pages 234–251. ACM, 2017. 10.1145/3132747.3132786.

[34] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang. Establishing trusted I/O paths for SGX client systems with Aurora. *IEEE Trans. Inf. Forensics Secur.*, 15:1589–1600, 2020. 10.1109/TIFS.2019.2945621.

[35] M. Marlinspike. Technology preview: Private contact discovery for Signal, 2017. URL https://signal.org/blog/private-contact-discovery/.

[36] H. Nemati. *Secure System Virtualization: End-to-End Verification of Memory Isolation*. PhD thesis, Royal Institute of Technology, Stockholm, 2017. URL http://nbn-resolving.de/urn:nbn:se:kth:diva-213030.

[37] A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on Intel SGX. *CoRR*, abs/2006.13598, 2020. URL https://arxiv.org/abs/2006.13598.

[38] Nitrokey. NetHSM - The Open Hardware Security Module. URL https://www.nitrokey.com/products/nethsm.

[39] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *PODC 1991*, pages 51–59. ACM, 1991. 10.1145/112600.112605.

[40] M. Ottela. Tinfoil Chat. URL https://github.com/maqp/tfc.

[41] J. E. Park and S. M. Ragan. Build an internet kill switch. URL https://makezine.com/projects/internet-kill-switch/.

[42] D. Parrinha and R. Chaves. Flexible and low-cost HSM based on non-volatile FPGAs. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, pages 1–8. IEEE, 2017. 10.1109/RECONFIG.2017.8279795.

[43] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 260–289, 2017. 10.1007/978-3-319-56620-7_10.

[44] Patton. 1-Gigabit Data Diode SFP Module. URL https://www.patton.com/sfx-1dd/.

[45] A. Peyrard, N. Kosmatov, S. Duquennoy, I. Lille, and S. Raza. Towards formal verification of Contiki: Analysis of the AES-CCM* modules with Frama-C. In D. Giustiniano, D. Koutsonikolas, A. Banchs, E. Mingozzi, and K. R. Chowdhury, editors, *Embedded Wireless Systems and Networks, EWSN 2018*, pages 264–269. Junction Publishing, Canada/ ACM, 2018. URL http://dl.acm.org/citation.cfm?id=3234910.

[46] S. Popoveniuc and B. Hosp. An introduction to PunchScan. In D. Chaum, M. Jakobsson, R. L. Rivest, P. Y. A. Ryan, J. Benaloh, M. Kutylowski, and B. Adida, editors, *Towards Trustworthy Elections, New Directions in Electronic Voting*, volume 6000 of *LNCS*, pages 242–259. Springer, 2010. 10.1007/978-3-642-12980-3_15.

[47] Qubes OS Project. Qubes split GPG, 2018. URL https://www.qubes-os.org/doc/split-gpg/. User Documentation.

[48] A. Salem, P. Berrang, M. Humbert, and M. Backes. Privacy-preserving similar patient queries for combined biomedical data. *Proc. Priv. Enhancing Technol.*, 2019(1):47–67, 2019. 10.2478/popets-2019-0004.

[49] SKUDO. KRYPTOR - FPGA Board. URL https://skudo.tech/products/kryptor.

[50] SoloKeys. Solo 2. URL https://www.indiegogo.com/projects/solo-v2-safety-net-against-phishing.

[51] Trussed. Modern cryptographic firmware. URL https://trussed.dev/.

[52] Wavestone - Cybersecurity & Digital Trust. Dyode : Do your own diode. URL https://github.com/wavestone-cdt/dyode.

[53] S. Weiser and M. Werner. SGXIO: generic trusted I/O path for Intel SGX. In G. Ahn, A. Pretschner, and G. Ghinita, editors, *ACM Conference on Data and Application Security and Privacy, CODASPY 2017*, pages 261–268. ACM, 2017. 10.1145/3029806.3029822.

[54] Yubico. YubiHSM. URL https://www.yubico.com/products/hardware-security-module/.

[55] E. Zheng, P. Gates-Idem, and M. Lavin. Building a virtually air-gapped secure environment in AWS. In M. P. Singh, L. Williams, R. Kuhn, and T. Xie, editors, *HoTSoS 2018*, pages 11:1–11:8. ACM, 2018. 10.1145/3190619.3190642.

Appendix

A Graceful Degradation

In the following, we discuss the security implications of our constructions if the assumptions about the remotely unhackable hardware modules (including enhanced channels) turn out to be wrong. For our constructions for up to $N-1$ corrupted parties, we can show that their security *gracefully degrades* to essentially adaptive UC security.

Let $\Pi_{\mathcal{G}}$ be the protocol of Construction 1 for realizing $\mathtt{AG}([\mathcal{G}])$ for some ideal functionality $\mathcal{G}$. We consider the protocol $\Pi'_{\mathcal{G}}$ that is identical to $\Pi_{\mathcal{G}}$, except that all unhackable sub-parties are now hackable and that all enhanced channels are replaced by standard channels. As a consequence, all main parties P_i as well as their sub-parties, including e.g. the previously unhackable Enc unit or the OIM, are always online and susceptible to online attacks.

As $\Pi'_{\mathcal{G}}$ still makes use of a (now hackable) interface module, it is not a legal protocol in the standard UC framework. For the same reason, we cannot hope to prove that $\Pi'_{\mathcal{G}}$ realizes the ideal protocol $\mathtt{SC}(\mathcal{G})$ in UC#, i.e. provides standard adaptive UC security. Instead, we have to consider an appropriate ideal functionality along with an appropriate ideal protocol.

PID-wise Corruption

We now discuss the case of PID-wise corruption, i.e. the case where the adversary may only corrupt a main party and its sub-parties together and defer more fine-grained corruption models to the full version. In the setting of PID-wise corruption, it suffices to consider the ideal functionality $\mathcal{G}'$ that is identical to $\mathcal{G}$, except that it provides the output of parties via their respective dummy OIM. The corresponding ideal protocol is identical to $\mathtt{SC}(\mathcal{G})$, except that for each party P_i, there also exists a (hackable) dummy OIM connected to the environment and $\mathcal{G}'$ via standard channels. As the sub-protocol $\mathcal{F}_{\mathcal{G}}$ is adaptively UC-secure, it can be shown that $\Pi'_{\mathcal{G}}$ securely realizes $\mathtt{SC}'(\mathcal{G}')$ in the UC# framework.

B Proof of Theorem 1

We will prove that $\Pi_{\mathcal{G}}$ emulates the ideal protocol $\mathtt{AG}([\mathcal{G}])$ in the UC# framework for adversaries corrupting at most $N-1$ parties $P \in \{P_1, \ldots, P_N\}$ under the assumptions that PKE is IND-parallel-CCA-secure, SIG is EUF-naCMA-secure and length-normal and MAC is EUF-1-CMA-secure.

Next, we define the following experiment, which will be used in the proof to show that an environment $\mathcal{Z}$ cannot send "fake messages" (P_i, c') to an honest party P_j addressed as coming from a party P_i that has *not* been corrupted before receiving input such that i) c' was not generated by the Enc unit of P_i and ii) (P_i, c') is accepted by P_j.

Definition 4 (Auxiliary Experiment)**.** The experiment $\mathsf{Exp}^{\text{aux}}_{\mathcal{A}(z),\text{PKE},\text{SIG}}(n)$ is defined as follows: At the beginning, the experiment generates keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \text{Gen}_{\text{PKE}}(1^n)$ and $(\mathsf{vk}, \mathsf{sgk}) \leftarrow \text{Gen}_{\text{SIG}}(1^n)$. On input $1^n, z$ and pk, the adversary $\mathcal{A}$ may then *non-adaptively* send queries to a signing oracle $\mathcal{O}_{\text{Sig}(\mathsf{sgk},\cdot)}$. Afterwards, the experiment sends vk to $\mathcal{A}$. $\mathcal{A}$ may then send a message of the form $(\mathtt{prf}_1, \mathtt{prf}_2, m)$ to the experiment. The experiment then computes $\sigma \leftarrow \text{Sig}(\mathsf{sgk}, (\mathtt{prf}_2, m))$, $c^* \leftarrow \text{Enc}(\mathsf{pk}, (\mathtt{prf}_1, m, \sigma))$, and sends c^* to $\mathcal{A}$. During the experiment, $\mathcal{A}$ may send a single *parallel* query to a decryption oracle $\mathcal{O}_{\text{Dec}(\mathsf{sk},\cdot)}$ subject to the restriction that the query does not contain c^*. At the end of the experiment, $\mathcal{A}$ sends a tuple (m', σ') to the experiment. The experiment then checks if $\text{Vrfy}_{\text{SIG}}(vk, m', \sigma') = 1$ and m' has not been sent to $\mathcal{O}_{\text{Sig}(\mathsf{sgk},\cdot)}$ before. If this holds, the experiment outputs 1 and 0 otherwise.

We have the following lemma.

Lemma 1. If PKE is IND-pCCA-secure and SIG EUF-naCMA-secure, then for every PPT adversary $\mathcal{A}$ and all $z \in \{0,1\}^*$, there exists a negligible function negl such that

$$\Pr[\mathsf{Exp}^{\text{aux}}_{\mathcal{A}(z),\text{PKE},\text{SIG}}(n) = 1] \leq \mathsf{negl}(n).$$

Sketch. Assume there exists an adversary $\mathcal{A}$ that wins in the experiment $\mathsf{Exp}^{\text{aux}}_{\text{PKE},\text{SIG},\mathcal{A}(z)}(n)$ with non-negligible probability. Since PKE is IND-pCCA-secure, one can replace c^* by $c' \leftarrow \text{Enc}(\mathsf{pk}, 0^L)$, where $L = |(\mathtt{prf}_1, m, \sigma)|$, incurring only a negligible loss in $\mathcal{A}$'s success probability. Then, one can directly construct an adversary $\mathcal{A}'$ out of $\mathcal{A}$ that breaks the EUF-naCMA-security of SIG with non-negligible probability. $\mathcal{A}'$ simply internally simulates the experiment $\mathsf{Exp}^{\text{aux}}_{\text{PKE},\text{SIG},\mathcal{A}(z)}(n)$ for $\mathcal{A}$ using his signing oracle and c' for c^*. Once $\mathcal{A}$ sends a tuple (m, σ) to the experiment $\mathsf{Exp}^{\text{aux}}_{\text{PKE},\text{SIG},\mathcal{A}(z)}(n)$, $\mathcal{A}'$ sends (m, σ) to the EUF-naCMA experiment. $\mathcal{A}'$ then wins in the

EUF-naCMA experiment if and only if $\mathcal{A}$ wins in the experiment $\mathsf{Exp}^{\mathrm{aux}}_{\mathrm{PKE,SIG},\mathcal{A}(z)}(n)$. □

Next, we define the simulator for the dummy adversary.

Definition 5 (Definition of the Simulator)**.** Define the simulator Sim interacting with an environment $\mathcal{Z}$ and a fortified ideal functionality $[\mathcal{G}]$ as follows:

1. At the beginning, Sim internally defines N parties corresponding to the parties in $\Pi_{\mathcal{G}}$. Throughout the simulation, Sim will keep track of the online state of these parties by marking them as online or offline. At the beginning, Sim marks these parties according to the initial online states of the dummy parties in the ideal protocol.
2. Sim initializes a Boolean variable $\texttt{verify} = \texttt{true}$.
3. Sim carries out the physical-attack instruction received from $\mathcal{Z}$ on its first activation. Sim carries out an $(\texttt{online-attack}, P_i)$ instruction only if Sim has marked party P_i as online.
4. Each time $\mathcal{Z}$ sends status, Sim sends the set of markings of each party.
5. Throughout the simulation, Sim reports the respective notify transport tokens to $\mathcal{Z}$ (note that we will not mention them anymore in the following).
6. Sim generates $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$, $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ and $(\mathsf{sgk}_i, \mathsf{vk}_i) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$ for each party P_i that was not corrupted *before* receiving input.
7. For each i such that party P_i is honest, Sim reports $(\texttt{registered}, \mathit{sid}', RM_i, \mathsf{pk}_i, \mathsf{vk}_i)$. If $\mathcal{Z}$ answers with "ok", Sim stores $(\mathsf{pk}_i, \mathsf{vk}_i)$ as "registered".
8. Each time Sim is activated by $[\mathcal{G}]$ after an *honest* party P_i received its input, Sim generates $3N$ random strings $s'_{ij}, r'_{ij}, k'_{ij}$, computes $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}))$ $(j = 1, \ldots, N)$ and $c_{ij} \leftarrow \mathrm{Enc}(\mathsf{pk}_j, (P_i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}))$. Each time party $\mathcal{Z}$ activates the Enc unit of P_i, Sim reports the respective tuple (P_i, c_{ij}) if pk_j is stored as "registered".
9. Once Sim has reported all (P_i, c_{ij}) for $j = 1, \ldots, N$ as well as $(\texttt{registered}, \mathit{sid}', RM_i, \mathsf{pk}_i, \mathsf{vk}_i)$ for an honest P_i, Sim marks P_i as online.
10. If a party P_i is corrupted after receiving input, Sim sends $(s'_{ii}, r'_{ii}, k'_{ii}, \sigma'_{ii}, \mathsf{vk}_i, \mathsf{sk}_i)$ to $\mathcal{Z}$.
11. If $\mathcal{Z}$ sends a value $(\mathsf{pk}_l, \mathsf{vk}_l)$ to $\mathcal{F}_{\mathsf{reg}}$ for a party P_l corrupted before receiving input, Sim stores $(\mathsf{pk}_l, \mathsf{vk}_l)$ as "registered".
12. Each time $\mathcal{Z}$ sends a message addressed to buffer of a party P_i, Sim stores that message as a message "received by P_i".
13. If $\mathcal{Z}$ activates an honest party P_j who is marked as online and has received at least $N - 1$ messages and all vk_l $(l = 1, \ldots, N)$ are stored as "registered", then Sim stores $\overline{\mathsf{vk}}_j = (\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ and reports $(\texttt{received}, P_i)$ to $\mathcal{Z}$. Upon receiving $(\texttt{confirmed}, P_i)$ from $\mathcal{Z}$, Sim marks P_j as input given.
14. If $\mathcal{Z}$ sends a tuple consisting of a vector $\overline{\mathsf{vk}}_j$ and $(s'_{lj}, r'_{lj}, k'_{lj}, \sigma'_{lj})$ $(l = 1, \ldots, N)$ as the input to $\mathcal{F}_{\mathcal{G}}$ for a corrupted party P_j, then Sim stores that input (if an input has already been stored for P_j then Sim overwrites it) and, if not done yet, marks P_j as input given.
15. Once all parties are marked as input given, Sim does the following:
    (i) If not all $\overline{\mathsf{vk}}_i$ $(i = 1, \ldots, N)$ are equal, Sim sets $\texttt{verify} = \texttt{false}$.
    (ii) For each j such that P_j is honest, Sim checks the following conditions:
       - P_j has received for each i such that party P_i is *not* corrupted before receiving input the tuple (P_i, c_{ij}), where c_{ij} is the respective ciphertext generated by Sim.
       - P_j has received for each l such that party P_l is corrupted before receiving input a set $\mathcal{M}_l$ fulfilling property $(*)$ from p. 324.

       If at least one of these conditions does not hold, Sim sets $\texttt{verify} = \texttt{false}$.
    (iii) For each tuple consisting of a vector $\overline{\mathsf{vk}}_j$ and $(s'_{lj}, r'_{lj}, k'_{lj}, \sigma'_{lj})$ $(l = 1, \ldots, N)$ which was stored by Sim as the input to $\mathcal{F}_{\mathcal{G}}$ for a corrupted party P_j, Sim checks the following:
       - for each i such that party P_i was *not* corrupted before receiving input, Sim checks if $(s'_{ij}, r'_{ij}, k'_{ij}) = (s_{ij}, r_{ij}, k_{ij})$, where (s_{ij}, r_{ij}, k_{ij}) is the respective tuple generated by Sim. If this does not hold or $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 0$, Sim sets $\texttt{verify} = \texttt{false}$.
       - for each l such that party P_l was corrupted before receiving input, Sim sets $\texttt{verify} = \texttt{false}$ if $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_l, (P_j, s'_{lj}, r'_{lj}, k'_{lj}), \sigma'_{lj}) = 0$.
16. Sim *extracts* the input, MAC key and random pad of each party P_l *corrupted before receiving input* by decrypting all ciphertexts coming from $\mathcal{Z}$, and by observing the inputs $\mathcal{Z}$ sends to $\mathcal{F}_{\mathcal{G}}$ for corrupted parties. Sim sends each extracted input to $[\mathcal{G}]$.
17. Once all parties are marked as input given and $\mathcal{Z}$ activates an honest party P_i, then Sim instructs $[\mathcal{G}]$ to

(i) send the output to P_i's dummy OIM if `verify` = `true`.

(ii) output $\perp$ to P_i's dummy OIM if `verify` = `false`.

18. Once all parties are marked as input given and $\mathcal{Z}$ requests the output of $\mathcal{F}_\mathcal{G}$ for a party P_i *corrupted after receiving input*, then
    (i) If `verify` = `true`, Sim generates a random string $\tilde{y}_i \leftarrow \{0,1\}^{p_i(n)}$ and sends $(\tilde{y}_i, \mathrm{Mac}(k_i, \tilde{y}_i))$ to $\mathcal{Z}$.
    (ii) If `verify` = `false`, Sim sends $\perp$ to $\mathcal{Z}$.
19. If $\mathcal{Z}$ sends a message (m', t') addressed to OIM of a party P_i *corrupted after receiving input*, then
    (i) If $\mathcal{Z}$ has not yet requested the output of $\mathcal{F}_\mathcal{G}$ for P_i yet, Sim instructs $[\mathcal{G}]$ to output $\perp$ to the dummy OIM of P_i.
    (ii) If $\mathcal{Z}$ has already requested the output of $\mathcal{F}_\mathcal{G}$ for P_i and Sim sent $(\tilde{y}_i, \mathrm{Mac}(k_i, \tilde{y}_i))$ (in item 18) to $\mathcal{Z}$, then
        - If $m' \neq \tilde{y}_i$, Sim instructs $[\mathcal{G}]$ to output $\perp$ to the dummy OIM of P_i.
        - If $m' = \tilde{y}_i$ and $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 1$, then Sim instructs $[\mathcal{G}]$ to send the output to the dummy OIM of party P_i. Otherwise, Sim instructs $[\mathcal{G}]$ to output $\perp$ to the dummy OIM of P_i.
    (iii) If $\mathcal{Z}$ has already requested the output of $\mathcal{F}_\mathcal{G}$ for P_i and Sim sent $\perp$ (in item 18) to $\mathcal{Z}$, then Sim instructs $[\mathcal{G}]$ to output $\perp$ to P_i's dummy OIM.
20. Once all parties are marked as input given and $\mathcal{Z}$ requests the output of $\mathcal{F}_\mathcal{G}$ for a party P_i *corrupted before receiving input*, then
    (i) If `verify` = `true`, Sim sends $(y_i + r_i, \mathrm{Mac}(k_i, y_i + r_i))$ to $\mathcal{Z}$, where y_i is the output of $[\mathcal{G}]$ for party P_i and k_i, r_i are the MAC key and random pad extracted in item 16.
    (ii) If `verify` = `false`, Sim sends $\perp$ to $\mathcal{Z}$.
21. Sim lets $\mathcal{Z}$ determine the output of the dummy OIM of each party *corrupted before receiving input.*

Proof. It suffices to find a simulator for the dummy adversary. In the following proof, we will consider a sequence of hybrids $\mathrm{H}_0, \ldots, \mathrm{H}_4$. Starting from the real protocol $\Pi_\mathcal{G}$, we will define ideal protocols that gradually reduce the simulator's abilities (i.e. restrict the set of parties for which he may learn/modify the inputs/outputs). The final hybrid H_4 will be the ideal protocol $\mathtt{AG}([\mathcal{G}])$ and the simulator as defined in Definition 5.

Let $\mathcal{Z}$ be an environment that instructs $\mathcal{D}$ to corrupt at most $N-1$ parties $P \in \{P_1, \ldots, P_N\}$. Let $\mathrm{out}_i(\mathcal{Z})$ be the output of $\mathcal{Z}$ in the hybrid H_i. In the following, we will say *corrupted before input* and *corrupted after input* for brevity.

Hybrid H_0

Let H_0 be the execution experiment between the environment $\mathcal{Z}$, the dummy adversary $\mathcal{D}$ and the real protocol $\Pi_\mathcal{G}$.

Hybrid H_1

Let H_1 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol $\mathtt{AG}(\mathcal{F}_1)$ and the ideal-model adversary Sim_1, where $\mathcal{F}_1$ and Sim_1 are defined as follows: Define $\mathcal{F}_1$ to be identical to $[\mathcal{G}]$ except for the following: $\mathcal{F}_1$ hands the adversary the inputs and outputs of *every* party (honest and corrupted) and allows him to determine the outputs of the dummy OIMs of *all corrupted* parties (i.e. all parties corrupted before *and* after input). Note that, like $[\mathcal{G}]$, $\mathcal{F}_1$ does *not* allow the adversary to modify the inputs of parties corrupted *after* input (unless all parties are corrupted) and also does not allow him to modify the inputs of honest parties.

Define Sim_1 to be like the simulator in Definition 5 except for the following: In item 8, Sim_1 reports the ciphertexts as they are generated in the real protocol (in particular, generates shares of the actual inputs). Also, if a party P_i is corrupted after having received input, Sim_1 reports the respective shares as they are generated in the real protocol in item 10 along with a valid signature and $\mathsf{vk}_i, \mathsf{sk}_i$. In item 18, if `verify` = `true`, Sim_1 reports $(y_i + r_i, \mathrm{Mac}(k_i, y_i + r_i))$ to $\mathcal{Z}$, where y_i is the output Sim_1 receives for the respective party from $\mathcal{F}_1$ and k_i, r_i are the MAC key and one-time pad generated in items 6 and 8. If `verify` = `false`, Sim_1 reports $\perp$. In item 19, if $\mathcal{Z}$ sends a message (m', t') addressed to OIM of a party P_i (corrupted after input), Sim_1 carries out the program of the OIM (using the MAC key and one-time pad generated in items 6 and 8), computing a value $y' \in \{0,1\}^{p_i(n)} \cup \{\perp\}$, and then instructs $[\mathcal{G}]$ to output y' to P_i's dummy OIM.

Consider the following events:

Let $\mathbf{E}_{\mathrm{fakemess}}$ be the event that there exists an *honest* party P_j that retrieves a tuple (P_i, c') in its buffer such that party P_i is *not* corrupted before input and (P_i, c') is "valid", i.e. $\mathrm{Dec}(\mathsf{sk}_j, c') = (P_i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ and $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 1$, but either $c' \neq c_{ij}$ or c_{ij} has not been generated yet (by the Enc unit of party P_i).

Let $\mathbf{E}_{\mathrm{fakein}}$ be the event that $\mathcal{Z}$ sends an input $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ for a corrupted party P_j to

$\mathcal{F}_\mathcal{G}$ such that party P_i is *not* corrupted before input and $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 1$, but $(s'_{ij}, r'_{ij}, k'_{ij}) \neq (s_{ij}, r_{ij}, k_{ij})$ (where vk_i and (s_{ij}, r_{ij}, k_{ij}) were generated by P_i).

Let $\mathbf{E} = \mathbf{E}_{\text{fakemess}} \cup \mathbf{E}_{\text{fakein}}$. It holds that

$$\mathsf{Pr}[\text{out}_0(\mathcal{Z}) = 1 \wedge \neg\mathbf{E}] = \mathsf{Pr}[\text{out}_1(\mathcal{Z}) = 1 \wedge \neg\mathbf{E}].$$

This is because if $\mathbf{E}_{\text{fakemess}}$ does not occur then a message in the buffer of a party P_j that is addressed as coming from a party P_i who was *not* corrupted before input decrypts to a valid message/signature pair if and only if it equals the ciphertext c_{ij} sent by P_i. Moreover, for each corrupted party P_i, since $\mathbf{E}_{\text{fakein}}$ does not occur, $\mathcal{Z}$ only sends inputs $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ to $\mathcal{F}_\mathcal{G}$ such that either $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 0$ or $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 1$ and $(s'_{ij}, r'_{ij}, k'_{ij}) = (s_{ij}, r_{ij}, k_{ij})$ was generated by party P_i (who was not corrupted before input).

Therefore, it holds that

$$\begin{aligned}|\mathsf{Pr}[\text{out}_0(\mathcal{Z}) = 1] - \mathsf{Pr}[\text{out}_1(\mathcal{Z}) = 1]| &\leq \mathsf{Pr}[\mathbf{E}] \\ &\leq \mathsf{Pr}[\mathbf{E}_{\text{fakemess}}] + \mathsf{Pr}[\mathbf{E}_{\text{fakein}}].\end{aligned}$$

Claim 1: $\mathsf{Pr}[\mathbf{E}_{\mathbf{fakemess}}]$ is negligible

Consider the following adversary $\mathcal{A}$ in the auxiliary experiment $\mathsf{Exp}^{\text{aux}}_{\text{PKE,SIG},\mathcal{A}(z)}(n)$: At the beginning, $\mathcal{A}$ randomly selects a tuple $(i, j) \in \{1, \ldots, N\} \times \{1, \ldots, N\}$ such that $i \neq j$. $\mathcal{A}$ then simulates hybrid H_0 using the public key pk from the experiment for pk_j in its internal simulation. When $\mathcal{Z}$ gives the party P_i its input x_i, $\mathcal{A}$ generates shares s_{il}, r_{il}, k_{il} of x_i, of a random pad r_i and of a MAC key k_i just like in H_0. $\mathcal{A}$ sends the tuples $(P_l, s_{il}, r_{il}, k_{il})$ for $l \neq j$ to the signing oracle $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$, receiving signatures σ_{il}. After receiving the verification key vk from the experiment, $\mathcal{A}$ uses vk for vk_i in its internal simulation. Using pk, $\mathcal{A}$ encrypts all tuples $(P_i, s_{il}, r_{il}, k_{il}, \sigma_{il})$ $(l \notin \{i, j\})$ and sends them to the respective party in its internal simulation. Once the message (P_i, c_{ij}) is supposed to be sent in the internal simulation, $\mathcal{A}$ sends $(\mathtt{P_i}, \mathtt{P_j}, s_{ij}, r_{ij}, k_{ij})$ to the experiment, receiving c^*. $\mathcal{A}$ then uses (P_i, c^*) for (P_i, c_{ij}) in its simulation. When P_j is activated and is online and has received at least $N - 1$ messages, $\mathcal{A}$ sends all ciphertexts addressed as coming from P_i such that $c \neq c^*$ to the decryption oracle $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$ (if c^* has not been generated yet, $\mathcal{A}$ sends all ciphertexts addressed as coming from P_i). For each message (P_l, m, σ) he receives from the oracle $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$, $\mathcal{A}$ checks if $\mathrm{Vrfy}_{\mathrm{SIG}}(vk, (P_j, m), \sigma) = 1$. If this holds for a message (P_l, m', σ'), then $\mathcal{A}$ sends (P_j, m', σ') to the experiment. If during the simulation, P_i is corrupted before input or P_j is corrupted (before or after input) or if no message $\mathcal{A}$ receives from $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$ is valid, then $\mathcal{A}$ sends $\perp$ to the experiment.

By construction, it holds that if $\mathbf{E}_{\text{fakemess}}$ occurs and $\mathcal{A}$ has correctly guessed an index (i, j) for which $\mathbf{E}_{\text{fakemess}}$ occurs, then $\mathcal{A}$ sends a message c' to $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$ such that $c \neq c^*$ or c^* has not been generated yet and $\mathrm{Dec}(\mathsf{sk}, c') = (P_i, m', \sigma')$ and $\mathrm{Vrfy}_{\mathrm{SIG}}(vk, (P_j, m'), \sigma') = 1$. Since $\mathcal{A}$ does not send a message of the form $(\mathtt{P_j}, m)$ to the signing oracle $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$, it follows that $\mathsf{Exp}^{\text{aux}}_{\text{PKE,SIG},\mathcal{A}(z)}(n) = 1$. Furthermore, the probability that $\mathcal{A}$ correctly guesses an index (i, j) for which $\mathbf{E}_{\text{fakemess}}$ occurs is at least $1/(N \cdot (N - 1))$. Hence,

$$\mathsf{Pr}[\mathsf{Exp}^{\text{aux}}_{\text{PKE,SIG},\mathcal{A}(z)}(n) = 1] \geq \mathsf{Pr}[\mathbf{E}_{\text{fakemess}}]/(N \cdot (N - 1)).$$

Therefore, since $\mathsf{Pr}[\mathsf{Exp}^{\text{aux}}_{\text{PKE,SIG},\mathcal{A}(z)}(n) = 1]$ is negligible by Lemma 1 and $N \cdot (N - 1)$ is polynomial in n, it follows that $\mathsf{Pr}[\mathbf{E}_{\text{fakemess}}]$ is also negligible.

Claim 2: $\mathsf{Pr}[\mathbf{E}_{\mathbf{fakein}}]$ is negligible

Consider the following adversary $\mathcal{A}$ against the EUF-naCMA security of SIG: At the beginning, $\mathcal{A}$ randomly selects an index $i \in \{1, \ldots, N\}$. $\mathcal{A}$ then simulates hybrid H_0. When $\mathcal{Z}$ gives the party P_i its input x_i, $\mathcal{A}$ generates shares s_{ij}, r_{ij}, k_{ij} of x_i, of a random pad r_i and of a MAC key k_i just like in H_0. $\mathcal{A}$ sends the tuples $(P_j, s_{ij}, r_{ij}, k_{ij})$ $(j \neq i)$ to the signing oracle $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$, receiving signatures σ_{ij}. After receiving vk, $\mathcal{A}$ then uses vk for vk_i, encrypts all tuples $(P_i, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij})$ $(j = 1, \ldots, N)$ and sends them to the respective party in its internal simulation. Each time $\mathcal{Z}$ sends a tuple $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ as input for a corrupted party P_j to $\mathcal{F}_\mathcal{G}$ such that $(s'_{ij}, r'_{ij}, k'_{ij}) \neq (s_{ij}, r_{ij}, k_{ij})$, $\mathcal{A}$ checks if $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, (P_j, s'_{ij}, r'_{ij}, k'_{ij}), \sigma'_{ij}) = 1$. If this holds, $\mathcal{A}$ sends $(P_j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ to the experiment. If during the simulation, P_i is corrupted before input or if no message $\mathcal{A}$ checks is valid, then $\mathcal{A}$ sends $\perp$ to the experiment.

By construction, it holds that if $\mathbf{E}_{\text{fakein}}$ occurs and $\mathcal{A}$ has correctly guessed an index i for which $\mathbf{E}_{\text{fakein}}$ occurs, then $\mathsf{Exp}^{\mathsf{euf\text{-}nacma}}_{\mathrm{SIG},\mathcal{A}(z)}(n) = 1$ because the tuple $(P_j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma_{ij})$ is valid and $(P_j, s'_{ij}, r'_{ij}, k'_{ij}) \neq (P_j, s_{ij}, r_{ij}, k_{ij})$ has not been sent to the signing oracle $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$. Furthermore, the probability that $\mathcal{A}$ correctly guesses an index i for which $\mathbf{E}_{\text{fakein}}$ occurs is at least $1/N$. Hence,

$$\mathsf{Pr}[\mathsf{Exp}^{\mathsf{euf\text{-}nacma}}_{\mathrm{SIG},\mathcal{A}(z)},(n) = 1] \geq \mathsf{Pr}[\mathbf{E}_{\mathrm{fakein}}]/N.$$

Therefore, since $\mathsf{Pr}[\mathsf{Exp}^{\mathsf{EUF\text{-}naCMA}}_{\mathrm{SIG},\mathcal{A}(z)}(n) = 1]$ is negligible because SIG is EUF-naCMA-secure by assumption and N is polynomial in n, it follows that $\mathsf{Pr}[\mathbf{E}_{\mathrm{fakein}}]$ is also negligible. Hence, there exist a negligible function negl_1 such that

$$|\mathsf{Pr}[\mathrm{out}_0(\mathcal{Z}) = 1] - \mathsf{Pr}[\mathrm{out}_1(\mathcal{Z}) = 1]| \leq \mathsf{negl}_1(n).$$

Hybrid H_2

Let H_2 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol $\mathtt{AG}(\mathcal{F}_1)$ (again) and the ideal-model adversary Sim_2, where Sim_2 is defined as follows:

Define Sim_2 to be like Sim_1 except for the following: In item 8, each time Sim_2 is activated by $\mathcal{F}_1$ after an *honest* party P_i received its input, Sim_2 generates N random strings k'_{ij} and computes $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, (P_j, s_{ij}, r_{ij}, k'_{ij}))$ $(j = 1, \ldots, N)$, where the s_{ij} and r_{ij} $(j = 1, \ldots, N)$ are still the shares of the input x_i and a random pad r_i, respectively. Sim_2 then iteratively reports $(P_i, \mathrm{Enc}(\mathsf{pk}_j, (P_i, s_{ij}, r_{ij}, k'_{ij}, \sigma'_{ij})))$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ to $\mathcal{Z}$. If a party P_i is corrupted after having received input, Sim_2 sends $(s_{ii}, r_{ii}, k'_{ii}, \sigma'_{ii}, \mathsf{vk}_i, \mathsf{sk}_i)$ to $\mathcal{Z}$ in item 10. (Note that in item 18 Sim_2 still uses the MAC key $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ generated in item 6 for the output of $\mathcal{F}_{\mathcal{G}}$ to a party P_i corrupted after input (if that output is $\neq \bot$)).

Let $\mathrm{H}_{2,0}, \ldots, \mathrm{H}_{2,N}$ be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol $\mathtt{AG}(\mathcal{F}_1)$ and the ideal-model adversary $\mathsf{Sim}_{2,0}, \ldots, \mathsf{Sim}_{2,N}$, respectively, where $\mathsf{Sim}_{2,i}$ is defined as follows:

Define the simulators $\mathsf{Sim}_{2,i}$ to be like Sim_1 except for the following: In item 8, each time $\mathsf{Sim}_{2,i}$ is activated by $\mathcal{F}_1$ after an *honest* party $P_l \in \{P_1, \ldots, P_i\}$ received its input, $\mathsf{Sim}_{2,i}$ generates N random strings k'_{lj}, computes $\sigma'_{lj} \leftarrow \mathrm{Sig}(\mathsf{sgk}_l, (P_j, s_{lj}, r_{lj}, k'_{lj}))$ $(j = 1, \ldots, N)$, and iteratively reports $(P_l, \mathrm{Enc}(\mathsf{pk}_j, (P_l, s_{lj}, r_{lj}, k'_{lj}, \sigma'_{lj})))$ $(j \in \{1, \ldots, N\} \setminus \{l\})$ to $\mathcal{Z}$. If a party $P_l \in \{P_1, \ldots, P_i\}$ is corrupted after having received input, $\mathsf{Sim}_{2,i}$ sends $(s_{ll}, r_{ll}, k'_{ll}, \sigma'_{ll}, \mathsf{vk}_l, \mathsf{sk}_l)$ to $\mathcal{Z}$ in item 10.

It holds that

$$\mathsf{Pr}[\mathrm{out}_{2,0}(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_1(\mathcal{Z}) = 1]$$

and

$$\mathsf{Pr}[\mathrm{out}_{2,N}(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_2(\mathcal{Z}) = 1].$$

Assume that there exists a non-negligible function ϵ such that $|\mathsf{Pr}[\mathrm{out}_1(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_2(\mathcal{Z}) = 1]| > \epsilon$. Then there exists an $i^* \in \{1, \ldots, N\}$ such that

$$|\mathsf{Pr}[\mathrm{out}_{2,i^*-1}(\mathcal{Z}) = 1] - \mathsf{Pr}[\mathrm{out}_{2,i^*}(\mathcal{Z}) = 1]| > \epsilon/N.$$

Moreover, if party P_{i^*} is not corrupted after input, i.e. if it is corrupted before input or remains honest throughout the execution, then the views of $\mathcal{Z}$ in H_{2,i^*-1} and H_{2,i^*} are identically distributed. Therefore,

$$\begin{aligned}\epsilon/N <& |\mathsf{Pr}[\mathrm{out}_{2,i^*-1}(\mathcal{Z}) = 1] - \mathsf{Pr}[\mathrm{out}_{2,i^*}(\mathcal{Z}) = 1]| \\ =& |\mathsf{Pr}[\mathrm{out}_{2,i^*-1}(\mathcal{Z}) = 1 \\ &\qquad \wedge P_{i^*} \textbf{ corrupted after input}] \\ &- \mathsf{Pr}[\mathrm{out}_{2,i^*}(\mathcal{Z}) = 1 \\ &\qquad \wedge P_{i^*} \textbf{ corrupted after input}]|\end{aligned}$$

Consider the following adversary $\mathcal{A}$ against the IND-pCCA security of PKE: At the beginning, $\mathcal{A}$ randomly selects an index $j \in \{1, \ldots, N\} \setminus \{i^*\}$. $\mathcal{A}$ then simulates the experiment H_{2,i^*-1}. When $\mathcal{Z}$ gives the party P_{i^*} its input x_{i^*}, $\mathcal{A}$ generates shares s_{i^*l}, r_{i^*l}, k_{i^*l} of the input x_{i^*}, of a random pad r_{i^*} and of a MAC key k_{i^*} just like in H_{2,i^*-1}. $\mathcal{A}$ additionally generates random strings k'_{i^*l} $(l \in \{1, \ldots, N\})$. $\mathcal{A}$ then generates signatures $\sigma_{i^*j}, \sigma'_{i^*j}$ for $(P_j, s_{i^*j}, r_{i^*j}, k_{i^*j})$ and $(P_j, s_{i^*j}, r_{i^*j}, k'_{i^*j})$, respectively, and sends $(P_{i^*}, s_{i^*j}, r_{i^*j}, k_{i^*j}, \sigma_{i^*j})$, $(P_{i^*}, s_{i^*j}, r_{i^*j}, k'_{i^*j}, \sigma'_{i^*j})$ to the experiment, receiving a ciphertext c^*. Note that $\mathcal{A}$'s challenge messages are allowed, i.e. have the same length, because SIG is length-normal. $\mathcal{A}$ then continues simulating the experiment H_{2,i^*-1} using c^* as c_{i^*j} and his decryption oracle to decrypt the ciphertexts in the buffer of P_j that are addressed as coming from the parties corrupted before input but do not equal c^* (the ones that are equal to c^* are ignored. Note that a tuple (P_l, c^*) sent by a party P_l corrupted before input is always invalid since $P_l \neq P_{i^*}$). Note that in $\mathcal{A}$'s internal simulation, party P_{i^*} receives the correct value from $\mathcal{F}_{\mathcal{G}}$ (i.e. $(y_{i^*} + r_{i^*}, \mathrm{Mac}(k_{i^*}, y_{i^*} + r_{i^*}))$ or $\bot$). At the end of the experiment, $\mathcal{A}$ outputs what $\mathcal{Z}$ outputs. If during the simulation, $\mathcal{Z}$ corrupts P_j (before or after input) or if P_{i^*} is *not* corrupted after input, $\mathcal{A}$ sends $\bot$ to the experiment.

Let $\mathrm{output}_b(\mathcal{A})$ denote the output of $\mathcal{A}$ in the IND-pCCA experiment when the challenge bit b is chosen. By construction, assuming party P_{i^*} is corrupted after input, if $\mathcal{A}$ guessed an index j such that party P_j remains honest then it holds that if the challenge bit is 0 the view of $\mathcal{Z}$ in $\mathcal{A}$'s internal simulation is distributed as in the experiment H_{2,i^*-1} and if the challenge bit is 1 the view of $\mathcal{Z}$ in $\mathcal{A}$'s internal simulation is distributed as in the experiment H_{2,i^*}. Moreover, assuming party P_{i^*} is corrupted after input, the probability that $\mathcal{A}$ guesses an index j such that party P_j remains honest is at least $1/(N-1)$. Hence,

$$\begin{aligned}&|\Pr[\mathrm{output}_0(\mathcal{A})=1]-\Pr[\mathrm{output}_1(\mathcal{A})=1]|\\ =&|\Pr[\mathrm{out}_{2,i^*-1}(\mathcal{Z})=1\wedge P_{i^*}\ \textbf{corrupted after input}\\ &\quad\wedge\textbf{Guess correct}]\\ &-\Pr[\mathrm{out}_{2,i^*}(\mathcal{Z})=1\wedge P_{i^*}\ \textbf{corrupted after input}\\ &\quad\wedge\textbf{Guess correct}]|\\ >&\ \epsilon/(N\cdot(N-1)).\end{aligned}$$

This contradicts the IND-pCCA security of PKE. Hence, there exist a negligible function negl_2 such that

$$|\Pr[\mathrm{out}_1(\mathcal{Z})=1]-\Pr[\mathrm{out}_2(\mathcal{Z})=1]|\le\mathsf{negl}_2(n).$$

Hybrid H_3

Let H_3 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol $\mathtt{AG}(\mathcal{F}_2)$ and the ideal-model adversary Sim_3, where $\mathcal{F}_2$ and Sim_3 are defined as follows:

Let $\mathcal{F}_2$ be identical to $\mathcal{F}_1$ except that now the adversary is allowed to determine the outputs only of the dummy OIMs of the parties *corrupted before input.*

Define Sim_3 to be like Sim_2 except that item 19 is identical to the same step of the simulator in Definition 5. Let $\mathbf{E}_{\mathrm{fakeout}}$ be the event that $\mathcal{Z}$ sends a message (m',t') to OIM of a party P_i corrupted after input such that $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i,m',t')=1$ but either P_i has received $\perp$ from $\mathcal{F}_{\mathcal{G}}$ or a tuple (m,t) such that $m'\neq m$ or P_i has not received an output from $\mathcal{F}_{\mathcal{G}}$ yet. It is easy to see that the following holds:

$$\Pr[\mathrm{out}_2(\mathcal{Z})=1\wedge\neg\mathbf{E}_{\mathrm{fakeout}}]=\Pr[\mathrm{out}_3(\mathcal{Z})=1\wedge\neg\mathbf{E}_{\mathrm{fakeout}}]$$

Therefore, it holds that

$$|\Pr[\mathrm{out}_2(\mathcal{Z})=1]-\Pr[\mathrm{out}_3(\mathcal{Z})=1]|\le\Pr[\mathbf{E}_{\mathrm{fakeout}}].$$

Claim 3: $\Pr[\mathbf{E}_{\mathrm{fakeout}}]$ is negligible

Consider the adversary $\mathcal{A}$ against the EUF-1-CMA-security of MAC. At the beginning, $\mathcal{A}$ randomly selects an index $i\in\{1,\dots,N\}$. $\mathcal{A}$ then simulates the hybrid H_2. Once $\mathcal{Z}$ expects the output from $\mathcal{F}_{\mathcal{G}}$ for party P_i (if P_i is corrupted after input), $\mathcal{A}$ computes the (padded) result m for this party. If $m=\perp$, $\mathcal{A}$ sends $\perp$ to $\mathcal{Z}$. Otherwise, $\mathcal{A}$ sends m to the MAC oracle $\mathcal{O}_{\mathrm{Mac}(k,\cdot)}$, receiving a tag t. $\mathcal{A}$ then sends (m,t) to $\mathcal{Z}$. If $\mathcal{Z}$ sends a tuple (m',t') to OIM of P_i such that $m'\neq m$, then $\mathcal{A}$ sends (m',t') to the experiment. If during the simulation, P_i is *not* corrupted after input or if $\mathcal{Z}$ sends $\perp$ or a tuple (m',t') such that $m'=m$ to OIM of P_i, then $\mathcal{A}$ sends $\perp$ to the experiment.

By construction, it holds that if $\mathbf{E}_{\mathrm{fakeout}}$ occurs and $\mathcal{A}$ correctly guessed an index for which $\mathbf{E}_{\mathrm{fakeout}}$ occurs, then $\mathsf{Exp}^{\mathsf{euf\text{-}1\text{-}cma}}_{\mathrm{MAC},\mathcal{A}(z)}(n)=1$ because (m',t') is valid and $m'\neq m$ has not been sent to $\mathcal{O}_{\mathrm{Mac}(k,\cdot)}$. Moreover, the probability that $\mathcal{A}$ correctly guesses an index for which $\mathbf{E}_{\mathrm{fakeout}}$ occurs is at least $1/N$. Hence,

$$\Pr[\mathsf{Exp}^{\mathsf{EUF\text{-}1\text{-}CMA}}_{\mathrm{MAC},\mathcal{A}(z)}(n)=1]\ge\Pr[\mathbf{E}_{\mathrm{fakeout}}]/N.$$

Therefore, since $\Pr[\mathsf{Exp}^{\mathsf{euf\text{-}1\text{-}cma}}_{\mathrm{MAC},\mathcal{A}(z)}(n)=1]$ is negligible because MAC is EUF-1-CMA-secure by assumption and N is polynomial in n, it follows that $\Pr[\mathbf{E}_{\mathrm{fakeout}}]$ is also negligible. Hence, there exist a negligible function negl_3 such that $|\Pr[\mathrm{out}_2(\mathcal{Z})=1]-\Pr[\mathrm{out}_3(\mathcal{Z})=1]|\le\mathsf{negl}_3(n)$.

Hybrid H_4

Let H_4 be the execution experiment between $\mathcal{Z}$, the ideal protocol $\mathtt{AG}(\mathcal{F}_3)$ and the ideal-model adversary Sim_4, where $\mathcal{F}_3$ and Sim_4 are defined as follows: Let $\mathcal{F}_3$ be identical to $\mathcal{F}_2$ except that the adversary is *not* given the inputs and outputs of honest parties anymore. The adversary is only given the inputs and outputs of parties corrupted after input when all parties are corrupted.

Define the adversary Sim_4 to be like Sim_3 except that items 8, 10 and 18 are identical to the same steps of the simulator in Definition 5.

Using an argument that is almost identical to the one in hybrid H_2 one can show that that there exists a negligible function negl_3 such that

$$|\Pr[\mathrm{out}_3(\mathcal{Z})=1]-\Pr[\mathrm{out}_4(\mathcal{Z})=1]|\le\mathsf{negl}_3(n)$$

Since H_4 is identical an execution between $\mathcal{Z}$, the ideal protocol $\mathtt{AG}([\mathcal{G}])$ and the simulator as defined in Definition 5, it follows that there exists a negligible function negl such that

$$\begin{aligned}&|\Pr[\mathrm{Exec}_{\mathrm{UC\#}}\big(\Pi_{\mathcal{G}},\mathcal{D},\mathcal{Z}\big)=1]-\\ &\qquad\Pr[\mathrm{Exec}_{\mathrm{UC\#}}\big(\mathtt{AG}([\mathcal{G}]),\mathsf{Sim},\mathcal{Z}\big)=1]|\le\mathsf{negl}(n).\end{aligned}$$

The statement follows. □

C A Short Introduction to the UC Framework

In the following, we give a brief overview of the UC framework. The following is adapted from [10]. For a detailed introduction see [11].

In the UC framework, security is defined by the indistinguishability of two experiments: the *ideal experiment* and the *real experiment*. In the ideal experiment, the task at hand is carried out by dummy parties with the help of an ideal incorruptible entity—called the ideal functionality $\mathcal{F}$. In the real experiment, the parties execute a protocol π in order to solve the prescribed task themselves. A protocol π is said to be a (secure) *realization* of $\mathcal{F}$ if no PPT machine $\mathcal{Z}$, called the *environment*, can distinguish between these two experiments. In contrast to previous simulation-based notions, indistinguishability must not only hold after the protocol execution has completed, but even if the environment $\mathcal{Z}$—acting as the *interactive* distinguisher—takes part in the experiment, orchestrates all adversarial attacks, gives input to the parties running the challenge protocol, receives the parties' output and observes the communication during the whole protocol execution.

The Basic Model of Computation

The basic model of computation consists of a set of (a polynomial number of) instances (ITIs) of interactive Turing machines (ITMs). An ITM is the description of a Turing machine with an additional identity tape, three externally writable input tapes (namely for input, subroutine output[9] and incoming messages) and an outgoing message tape. The latter is jointly used to provide input to any of the three input tapes of another ITM. The tangible instantiation of an ITM—the ITI—is identified by the content of its identity tape. The order of activation of the ITIs is completely asynchronous and message-driven. An ITI gets activated if input, subroutine output or an incoming message is written onto its respective tape. If the ITI writes onto its outgoing message tape and calls the special `external write` instruction, the activation of this ITI completes. The message must explicitly designate the identity and input tape of the receiving ITI. Each experiment comprises two special ITIs: The environment $\mathcal{Z}$, and the adversary $\mathcal{A}$ (in the real experiment) or the simulator $\mathcal{S}$ (in the ideal experiment). The environment $\mathcal{Z}$ is the ITI that is initially activated. If any ITI completes its activation without giving any output, the environment is activated again as a fallback. If the environment $\mathcal{Z}$ provides subroutine output, the whole experiments stops. The output of the experiment is the output of $\mathcal{Z}$. Without loss of generality, we assume that $\mathcal{Z}$ outputs a single bit only.

The Control Function and Message Delivery

If an ITI writes a message onto its outgoing message tape and calls `external write`, a *control function* decides if the operation is allowed[10]. If so, the experiment proceed as follows: If the receiver is uncorrupted and the designated input tape is either *input* or *subroutine output*, the message is copied to the respective tape of receiver. Else (meaning if the message is intended to be sent to an *incoming message* tape or the receiver is corrupted) the message is delivered to the respective tape of the adversary. This captures the natural intuition that input and subroutine output normally occurs within the same physical party and thus should be authenticated, immediate, confidential and of integrity. In contrast, external communication is only possible through an unreliable network under adversarial control.

UC Framework Conventions

In the UC framework, many important aspects are unspecified. For example, it leaves open which ITI is allowed to invoke what kind of new ITIs. The conventions stated in the following are probably the mostly used ones and quite natural.

Each party is identified by its party identifier (PID) *pid* which is unique to the party and is the UC equivalent of the physical identity of this party. A party runs a protocol π by means of an ITI which is called the *main party* of this instance of π. An ITI can invoke subsidiary ITIs to execute sub-protocols. A subsidiary and its parent use their *input/subroutine output* tape to communicate with each other. The set of ITIs taking part in the same protocol but for different parties communicate through their *incoming message* tapes. An instance

[9] Beware: Despite its name, this tape is actually an input tape as it receives subroutine output.

[10] N.b.: The control function is another ITI that exists "outside" of the experiment and checks which combination of sender ID, receiver ID and message tape are feasible. For example, only subsidiary ITIs are typically allowed to provide subroutine output to their main ITI. For details see [11].

of a protocol is identified by its session identifier (SID) sid. All ITIs taking part in the same protocol instance share the same SID. A specific ITI is identified by its ID $\mathsf{id} = (pid, sid)$.

The (Dummy) Adversary

The adversary $\mathcal{A}$ is instructed by $\mathcal{Z}$ and represents $\mathcal{Z}$'s interface to the network. To this end, all messages from any party to a party that has a different main party and that are intended to be written to an *incoming message* tape are copied to the adversary. The adversary can process the message arbitrarily. The adversary may decide to deliver the message (by writing the message on its own outgoing message tape), postpone or completely suppress the message, inject new messages or alter messages in any way including the recipient and/or alleged sender.

$\mathcal{Z}$ may also instruct $\mathcal{A}$ to corrupt a party. In this case, $\mathcal{A}$ takes over the position of the corrupted party, reports its internal state to $\mathcal{Z}$ and from then on may arbitrarily deviate from the protocol in the name of the corrupted party as requested by $\mathcal{Z}$. This means whenever the corrupted ITI would have been activated (even due to subroutine output), the adversary gets activated with the same input.

A special case for the adversary is the so-called *dummy adversary* that reports all received messages to the environment and delivers all messages coming from the environment. It can be shown that the dummy adversary is *complete*, i.e. that if a simulator for the dummy adversary exists, then there also exists a simulator for any other adversary.

Ideal Functionalities and the Ideal Protocol

An ideal functionality $\mathcal{F}$ is a special type of ITM whose instantiations (ITIs) bear a SID but no PID. Hence, it is an exception to the aforementioned identification scheme. Input to and subroutine output from $\mathcal{F}$ is performed through dummy parties. Dummy parties merely forward their input to the input tape of $\mathcal{F}$ and subroutine output from $\mathcal{F}$ to their own outgoing message tape. They share the same SID as $\mathcal{F}$, but additionally have individual party identifiers (PIDs) as if they were the actual main parties of a (real) protocol. The ideal functionality $\mathcal{F}$ is simultaneously a subroutine for each dummy party and conducts the prescribed task. IDEAL($\mathcal{F}$) is called the *(ideal) protocol* for $\mathcal{F}$ and denotes the set of $\mathcal{F}$ together with its dummy parties.

The UC Experiment

Let π be a protocol, $\mathcal{Z}$ an environment and $\mathcal{A}$ an adversary. The UC experiment, denoted by $\mathrm{Exec}_{\pi,\mathcal{A},\mathcal{Z}}(n,a)$, initially activates the environment $\mathcal{Z}$ with security parameter 1^n and input $a \in \{0,1\}^*$. The first ITI that is invoked by $\mathcal{Z}$ is the adversary $\mathcal{A}$. All other parties invoked by $\mathcal{Z}$ are set to be main parties of the challenge protocol π. $\mathcal{Z}$ freely chooses their input, their PIDs and the SID of the challenge protocol. The experiment is executed as outlined above.

Definition of Security

Let π, ϕ be protocols. π *emulates* ϕ *in the UC framework*, denoted by $\pi \geq \phi$, if for every PPT adversary $\mathcal{A}$ there is a PPT adversary $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$ there is a negligible function negl such that for all $n \in \mathbb{N}, a \in \{0,1\}^*$ it holds that

$$|\mathsf{Pr}[\mathrm{Exec}\big(\pi,\mathcal{A},\mathcal{Z}\big)(n,a) = 1] - \\ \mathsf{Pr}[\mathrm{Exec}\big(\phi,\mathcal{S},\mathcal{Z}\big)(n,a) = 1]| \leq \mathsf{negl}(n),$$

where $\mathrm{Exec}\big(\pi,\mathcal{A},\mathcal{Z}\big)(n,a)$ denotes the random variable for the environment $\mathcal{Z}$'s output in the UC execution experiment with protocol π and adversary $\mathcal{A}$ on input a and security parameter n.

The simulator $\mathcal{S}$ mimics the adversarial behavior to the environment as if this were the real experiment with real parties carrying out the real protocol with real π-messages. Moreover, $\mathcal{S}$ must come up with a convincing internal state upon corrupted parties, consistent with the simulated protocol execution up to this point (dummy parties do not have an internal state).

Protocol Composition

UC security is closed under protocol composition: Let π, ϕ, ρ be protocols. Then,

$$\pi \geq_{\mathrm{UC}} \phi \implies \rho^{\pi} \geq_{\mathrm{UC}} \rho^{\phi}$$

C.1 Bulletin Board Functionality

In our constructions, we make use of the ideal functionality $\mathcal{F}_{\mathsf{reg}}$ that models a public bulletin board.

Definition 6 ($\mathcal{F}_{\mathsf{reg}}$)**.** $\mathcal{F}_{\mathsf{reg}}$ proceeds as follows:

- Report: Upon receiving a message $(\texttt{register}, sid, v)$ from party P, send $(\texttt{registered}, sid, P, v)$ to the adversary; upon confirmation, record the pair (P, v). Otherwise, ignore the message.

- Retrieve: Upon receiving a message $(\texttt{retrieve}, \mathit{sid}, P_i)$ from some party P_j (or the adversary $\mathcal{A}$), generate a public delayed output $(\texttt{retrieve}, \mathit{sid}, P_i, v)$ to P_j, where $v = \perp$ if no record (P, v) exists.

Note that in contrast to the usual definition, we allow key revocation in $\mathcal{F}_{\mathsf{reg}}$.