Fortified Universal Composability: Taking Advantage of Simple Secure Hardware Modules

Brandon Broadnax[1], Alexander Koch[1], Jeremias Mechler[1], Tobias Müller[2], Jörn Müller-Quade[1], Matthias Nagel[1]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] FZI Research Center for Information Technology

{brandon.broadnax,alexander.koch,jeremias.mechler,joern.mueller-quade, matthias.nagel}@kit.edu,tobias.mueller@fzi.de

Abstract. We initiate the study of incorporating remotely unhackable hardware modules, such as air-gap switches and data diodes, into the field of multi-party computation. As a result, we are able to construct protocols with very strong composable security guarantees that cannot be achieved with adaptive security.

Our application of hardware modules is motivated by the fact that modules with very limited functionality can be implemented securely as fixed-function circuits and (formally) verified for correctness. They can therefore not be *hacked remotely*.

In comparison to the hardware tokens proposed by Katz at EUROCRYPT '07, our hardware modules are based on substantially weaker assumptions. Our hardware modules may be physically tampered. Hence, they cannot be passed to another (possibly malicious) party but only used and trusted by their owner. In particular, our remotely unhackable hardware modules do not constitute a setup for Universal Composability (UC).

Based on architectures with very few and very simple hardware modules, we are able to construct protocols that provide security against remote hacking if the hack occurs *after* a protocol party received its (first) input. More specifically, an adversary can neither learn nor change the inputs and outputs of a remotely hacked party in our constructions unless he has control over that party before it has received its (first) input (or controls all parties). In our constructions we assume erasing parties. However, we also show that this assumption can be substantially weakened.

Since the advantages provided by unhackable hardware modules cannot be adequately captured in existing composable security frameworks, we have conceived a new security framework based on the UC framework. We call our framework *Fortified UC*.

Keywords: universal composability, secure hardware modules

1 Introduction

In the field of multi-party computation, one distinguishes between *static* and *adaptive* corruptions. In the static setting, parties may only be corrupted prior to the start of the protocol. In the adaptive corruption model (first proposed by [CFGN96]), the adversary is able to corrupt parties throughout the protocol execution. In particular, the adversary learns all secrets of a protocol party even if a party is corrupted late in the protocol execution.

In practice, however, a protocol party could be isolated from the network and may therefore not be corruptible at any given moment during the protocol execution. For instance, a party may use unidirectional channels (data diodes) or disconnect itself via air-gap switches, making corruption through a remote hack impossible. To successfully attack, an adversary would have to hack that party before that party disconnects itself. Furthermore, a party may have additional hardware modules at its disposal that have very limited functionality (and, in particular, are not freely-programmable) and can therefore be implemented as fixed-function circuits and verified for correctness. Such hardware modules are resilient against remote hacking. They could only be corrupted if the adversary had direct physical access to them.

We therefore propose a new framework—called *"Fortified UC"*—based on the UC framework [Can01] that distinguishes between *"corruption"* and *"hacking"*. By corruption, we mean that a protocol party is under the control of the adversary before it has received its (first) input. In contrast, we speak of hacking if the adversary has gained control of a protocol party *after* that party has received its (first) input. A protocol party can only be hacked in our framework if it is currently *online*. Whether or not a party is currently online is determined by the current state of its channels, e.g., state of its air-gap switches. We call the set and structure of the hardware modules of a protocol its *architecture*.

We show that one can effectively protect against hacking. More specifically, assuming an appropriate architecture, an adversary who hacks a party cannot learn the inputs or outputs of that party, nor is he able to *change* them unless he has hacked or corrupted *all* parties. Our protocols require erasure but we also show how to considerably weaken this assumption using an appropriate architecture.

Surprisingly, we can achieve these results with only very few simple unhackable hardware modules such as an encryption device that only implements a specific public key encryption scheme. Unlike the hardware tokens proposed by [Kat07], our unhackable hardware modules can be tampered if one has direct physical access to them. They are only trusted by the party that uses them and are not passed to other parties. In particular, they cannot be used as a UC-complete setup. Using these unhackable hardware modules, we are able to protect certain parts of a protocol party such as its output interface by appropriately modularizing that party into complex hackable components as well as few simple unhackable components.

1.1 Our Contribution

We utilize realistic unhackable hardware modules that, to the best of our knowledge, have so far not been used for secure multi-party computation. Our main contributions are:

- *New composable security framework for hacking adversaries:* We propose a new security framework that, unlike previous frameworks, captures the advantages of "fortification" provided by unhackable hardware modules and isolation. As with UC security, our security notion is universally composable (cf. Theorem 2). Furthermore, our security notion is equivalent to UC security for protocols that do not use any unhackable hardware modules. In particular, UC-secure protocols can be used as building blocks for constructions in our framework (cf. Theorem 1).

- *New protocols that provide security against hacking:* Using only very few simple unhackable hardware modules, we construct protocols with very strong composable security guarantees which cannot be achieved by adaptive security. We present a construction for non-reactive functionalities (cf. Theorem 3) using only two simple unhackable hardware modules (apart from air-gap switches and data diodes) per party and a protocol for reactive functionalities (cf. Theorem 5) that uses only one additional simple unhackable hardware module. Both constructions can be proven secure in our new framework for adversaries that corrupt or hack all but one parties. We also present an augmentation of these constructions that allow simulation even in the case that all parties are corrupted or hacked (cf. Theorem 4 and Theorem 6). For our constructions we assume erasing parties. However, we later also show how this assumption can be weakened to assuming that honest parties can be reset after the protocol execution (cf. Section 6).

1.2 Related Work

Adaptive Security, first proposed in [CFGN96], captures security against adversaries that can corrupt participants at any time in the protocol. This notion has since received considerable attention by the cryptographic community, see e.g. [CLOS02; IPS08; HLP15; CPV17]. In contrast to adaptive security where an adversary learns all secrets of a corrupted party, we achieve that *hacking* a party after it received its inputs does not leak anything about them at all.

Mobile adversaries [OY91; BDLO14], a notion strictly stronger than adaptive attacks, models an adversary taking over a participant – similar in spirit to our framework as "hacks/virus attacks" – and possibly undoing the corruption at a later point in time.

Concerning the used *trusted building blocks*, we assume data diodes, which are channels which allow for communication only in one specified direction. [GIK+15] analyze the cryptographic power of unidirectional channels as a building block, whereas we use unidirectional channels as a shield against dangerous incoming data packets. [AMR14] makes use of other building blocks, such as a secure equality check hardware module to ensure the correct, UC-secure functioning of a parallel firewall setup in the case of a malicious firewall.

Tamper-proof hardware tokens, first proposed by [Kat07], are an interesting research direction for finding plausible and minimal setup assumptions for secure protocols. Along this line of research, [GIS+10] showed strong feasibility results of what can be done with these tokens. Moreover, [DMMN13] showed that UC security is possible with a constant number of untrusted and resettable hardware tokens. Furthermore, [HPV17] constructed constant-round adaptively secure protocols which allow up to N parties to be corrupted. As discussed above, we do not make use of tamper-proofness since the trusted hardware stays local to the participant.

Isolation is a general principle in IT security, with lots of research on software isolation through virtualization, see e.g. [Nem17]. In a sense, this can be seen as a software analog of an 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. ZGL18], which are however, not relevant to our work, because these isolations 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 network.

2 Fortified UC

In this section, we present our changes to the UC framework. Namely, we introduce enhanced channels, e.g. unidirectional ones that allow message flow in one direction only, together with the online-offline state for protocol parties. Additionally, we introduce a new kind of online corruption, called *"hacking"*. We also strengthen the adversary by being notified on immediate communication. In order to model our guarantees against hacking adversaries, we introduce *"fortified ideal functionalities"*.

2.1 Conventions and Notation

We denote the security parameter by n and the number of rounds of a reactive protocol by R. $\underset{\text{UC}}{\geq}$ denotes UC emulation with respect to *adaptive* corruption.

2.2 Enhanced Channels and Online-Offline State

In the UC framework, communication is possible via the `external write` instruction, which we shortly introduce. `external write` takes the sender's code and id, the receiver's code and id, the output tape as well as the message as arguments. A control function C then decides if the write is allowed or forbidden. This communication model is (intentionally) abstract and dynamic in the sense that there are e.g. no fixed, dedicated channels between protocol parties and additionally allows to capture properties such as trusted communication [Can01].

In order to reason about the online-offline state of a protocol party as well as being able to model e.g. unidirectional communication, we deviate from this concept: When the protocol architecture implies possible communication between two ITMs μ and μ', we say that there exists a channel between μ and μ'. (Formally, this means that there is an execution prefix such that the control function C

would allow an external write between μ and μ'.) Without loss of generality, we assume that every channel has a unique identifier.

Enhanced Channels In our framework, we want to capture possible security gains resulting from being isolated, e.g. by air gaps or by restrictions to the message flow. We model this by enhancing existing communication channels in the UC framework. Like those *standard channels*, *enhanced* channels can also be, e.g., between two protocol parties or between a protocol party and an ideal functionality. Furthermore, delivery can be immediate or non-immediate.

For our constructions, we propose two new kinds of channels:

1. *Data diodes* that allow communication in one direction only.
2. *Air-gap switches* that can be *connected* or *disconnected* by a protocol party that uses them. Disconnected air-gap switches allow no data transmissions at all, connected ones work as usual. For each air-gap switch, the *initial connected / disconnected state* must be specified.

As in the UC framework, `external write` calls are not carried out if the control function C forbids them. For example, messages sent in the wrong direction of a data diode are silently dropped.

Input / Output Online State The environment may, upon each activation, determine the online-offline state of each channel it uses to provide input to or receive output from protocol parties.

Online State of Parties Each main party is always explicitly connected to the *network*, i.e. connections between parties with different main parties, via a dedicated channel.

Online-offline state of parties A (sub-)party P of protocol π is *online via channel* X if

1. it can receive messages from a (sub-)party via X that either has a different main party or has the same main party as P and is online, or
2. (unless specified otherwise, as in the case of initially offline functionalities, cf. Page 6) it can receive messages from a multi-party ideal functionality $\mathcal{F}$ via X or
3. it can receive input via X from the environment or give output to the environment via X and X is not a data diode and (in both cases) the environment has *set* X *online*.

Otherwise, P is *offline via* X. If P is offline via all its channels, we say that P is *offline*. If P is online via some channel X, we say that P *is online*.

Status report Each time the adversary is activated, he gets the current online / offline states of all parties, which we call the *status*.

Initially Offline Ideal Functionalities We say that a multi-party ideal functionality is *initially offline* if a party that is connected via a standard channel or a (connected) air-gap switch to that functionality is online via the channel to that functionality as soon as it has provided input to that functionality, but offline before doing so. Multi-party ideal functionalities capture the additional security provided by disconnecting all channels except for the input port prior to receiving input.

2.3 Corruption Model

In our flavor of the UC framework, we propose a different kind of online corruption in contrast to the adaptive corruption model, called *hacking*. The adaptive corruption model allowed the adversary to corrupt a party at any point of time. In contrast, *hacking* can only happen when a party is online or able to receive messages from an unhackable party that has been *tainted* (see below).

In our framework, a (sub-)party can be either *hackable* or *unhackable* (but "corruptible").

The adversary may send $(\texttt{attack}, P)$ to a party P:

Corruption If $(\texttt{attack}, P)$ is sent *before* protocol invocation (as in the static corruption model) and P is a main party, then the environment is notified with "physical access corruption of P" and the adversary gets control over P and *all* of its sub-parties (regardless of whether they are unhackable). Also, he may choose to ignore enhanced channels of these parties.

If $(\texttt{attack}, P)$ is sent *after* protocol invocation and P is a main party which is *online*, *hackable* and has *not* received its (first) input yet, then the environment is notified with "online-initiated corruption of P" and the adversary gets control (only) over P. The adversary has to adhere to the communication restrictions implied by the enhanced channels of P.

In each case, we say that P is *corrupted* (cf. Appendix D for a motivating example).

Hacking If P is a *hackable* and *online*, the adversary gets control (only) over P and has to adhere to the communication restrictions implied by the enhanced channels of P. We then say that P is *hacked*. If P is a main party that has *already* received its (first) input, then the environment is additionally notified with "P hacked". If P is unhackable or offline, then nothing happens.

Taint with Cause In order to account for (sub-)parties that are only online via connections to *unhackable* sub-parties, we introduce the concept of *tainting with cause*. Tainting a sub-party gives no additional power to the adversary over the tainted party. However, it allows the $(\texttt{attack}, P)$ instruction to pass on as soon as P with PID pid_k can receive messages from the tainted party. Formally, *tainting with cause* means that the adversary sends an instruction $(\texttt{taint}, T)$ where T is a sequence of PIDs $(pid_1, \ldots, pid_k)$. The party with PID pid_1 must be online and unhackable. The party P with pid_k must be hackable. All other parties must be unhackable. Moreover, there must be a path from pid_1 to pid_k such that pid_i and pid_{i+1} $(i = 1, \dots, k-1)$ are online over their connection at some point and pid_{i-1} and pid_i have been already online via their connection $(i = 2, \dots, k-2)$. Starting from PID pid_1, all parties in T are automatically tainted as soon as they are able to receive messages from their predecessor in T. $(\texttt{attack}, pid_k)$ is executed as soon as P can receive messages from pid_{k-1}. The adversary may specify multiple taint with cause instructions at the same time.

2.4 Accounting for Modularization

In our constructions (see Sections 4 and 5), we heavily rely on the modularization of protocol parties as well as enhanced channels. In order to properly account for this, protocols are not only specified by their parties' code, but also by the *protocol architecture*.

In the UC framework, the adversary is not activated when immediate communication between sub-parties happens and thus is not able to adaptively hack them at these points. In our enhanced model, this is undesirable because it does not appropriately capture our notion of security. Consider, for instance, a hackable and online party sending a message (containing secret information) to an unhackable sub-party and erasing the message directly afterwards. As the message delivery is immediate, the adversary is not activated and thus is unable to intercept the message before it is erased by the sender even though the sender has been online *all the time*. We therefore give additional power to the adversary by introducing the *notify transport* mechanism, which notifies and activates the adversary under certain conditions when immediate message delivery happens.

Notify Transport Let μ, μ' have *different* PIDs and be connected via a channel with *immediate delivery*.

If μ sends a message to μ' and μ and μ' have the same main party or (unless specified otherwise) μ' is an ideal functionality, the adversary is sent a *notify transport* token consisting of μ''s PID.

Upon receiving a notify transport token, the adversary can then choose to either do nothing, or send the token containing μ''s PID to the environment. $\mathcal{Z}$ may only activate $\mathcal{A}$ again which, depending on $\mathcal{Z}$'s answer, may do either nothing or send an `attack` or `taint` instruction. Only if μ' is then under adversarial control, $\mathcal{A}$ is activated again. Otherwise, μ' is activated.

Note that, as implied by the definition, no notify transport tokens are issued for communication with the environment (e.g. when the environment gives inputs).

Interface Parties With respect to the modeling of parties, we deviate from the UC framework by allowing the main parties to invoke *interface parties*, called *input interface machines* (IIMs) and *output interface machines*, which are connected only to their main party and the environment via immediate channels. They are responsible for providing input resp. output. In the ideal execution, ideal functionalities are responsible for invoking the respective dummy parties (see Definition 2).

Protocol Architecture The *protocol architecture* specifies all communication channels (in particular their types and initial states) between (sub-)parties, functionalities, the environment as well as to the network. Note that this implies that each party has an *initial online-offline state* prior to invocation. The protocol architecture also specifies which parties are hackable and which are unhackable.

Combination of Parties As in the UC framework, we allow the (formal) combination of parties P, P' if they have the same main party, are connected via standard connections only and are both either hackable or unhackable.

As in the UC framework, we combine parties by giving them the *same PID*.

Note that, by definition, no notify transport token is given to the adversary for communication between combined parties as they have the same PID.

We will later (implicitly) combine dummy parties with their respective calling party in the constructions presented in this work.

2.5 Fortified UC emulation

We will now define security in our framework in analogy to the UC framework.

Definition 1 (##-Emulation). *Denote by* $\mathrm{Exec}_{\#\#}(\pi, \mathcal{A}, \mathcal{Z})(n, a) \in \{0,1\}$ *the output of the environment* $\mathcal{Z}$ *on input* $a \in \{0,1\}^*$ *and with security parameter* $n \in \mathbb{N}$ *when interacting with* π *and* $\mathcal{A}$ *according to the rules of the Fortified UC framework as specified in Sections 2.2 to 2.4.*

Let π *and* ϕ *be protocols.* π *is said to emulate* ϕ *in the Fortified UC framework, denoted by* $\pi \underset{\#\#}{\geq} \phi$*, if for every* PPT*-adversary* $\mathcal{A}$ *there exists a* PPT*-adversary* $\mathcal{S}$ *(the “simulator”) such that for every* PPT*-environment* $\mathcal{Z}$ *there exists negligible function* negl *such that for all* $n \in \mathbb{N}, a \in \{0,1\}^*$ *it holds that*

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

2.6 Fortified Functionalities

In contrast to adaptive corruption, where the adversary may, depending on the protocol state, change a corrupted party’s input or output, we want to weaken the implication of being hacked: Unless all N protocol parties are corrupted or hacked, the adversary does not learn a party’s input or output and may not change the input or output of a hacked party. The adversary is only given the possibility to pass on the correct output (which he does not learn) or to abort. This is modelled by “fortified functionalities” in our framework as follows (note that $\mathcal{G}$ is of the form as in Definition 7 in Appendix B):

Definition 2 (Fortified Functionality). *Let $\mathcal{G}$ be an ideal functionality with N protocol parties. Define the* fortified functionality $[\mathcal{G}]$ *of $\mathcal{G}$ as follows:*

- $[\mathcal{G}]$ *is* initially offline
- $[\mathcal{G}]$ *internally runs $\mathcal{G}$ and behaves as follows:*
  Setup: Set $c := 0$.
  Execution:
  - *When a party P receives its* first *input,* $[\mathcal{G}]$ *invokes a dummy* output party*. If $\mathcal{G}$ is reactive, a dummy* input party *is also invoked. Subsequent inputs for P must be provided via that input party (otherwise ignored).*
  - *On input* $(\texttt{attack}, P)$*: If P has* not yet *received its first input, forward* $(\texttt{corrupt}, P)$ *to $\mathcal{G}$ and increment c. Otherwise, only increment c.*
  - *If $c = N$, send all inputs to the adversary.*

  Output:
  - *If $c < N$ and $\mathcal{G}$ sends output for party P that has not been corrupted (but possibly hacked), ask the adversary whether to abort or pass on the correct output (which is not given to the adversary) via the dummy output party.*
  - *If $c = N$, the adversary may determine all parties' outputs.*

  Any other messages to or from the adversary or the protocol parties are relayed to or from $\mathcal{G}$.

Furthermore, for communication between the dummy parties and $[\mathcal{G}]$, *no* notify transport token is issued.

3 Properties of the Framework

As with UC security, our security notion is transitive and closed under general protocol composition, and the dummy adversary is complete. Furthermore, our security notion is equivalent to UC security for protocols that do not use any unhackable hardware modules (for proof sketches, see Appendix E).

Definition 3 (Emulation with Respect to the Dummy Adversary). *Define the* dummy adversary $\mathcal{D}$ *as follows:*

- *When receiving a message (sid, pid, m) from the environment, $\mathcal{D}$ sends m to the party with party identifier pid and session identifier sid.*
- *When receiving m from the party with party identifier pid and session identifier sid, $\mathcal{D}$ sends (sid, pid, m) to the environment.*
- *When receiving* `status` *from the environment, $\mathcal{D}$ sends the status to the environment.*

Let π and ϕ be protocols. π is said to emulate ϕ with respect to the dummy adversary *in the Fortified UC framework, if there exists a* PPT*-adversary $\mathcal{S}_\mathcal{D}$ such that for every* PPT*-environment $\mathcal{Z}$ there exists negligible function* negl *such that for all $n \in \mathbb{N}, a \in \{0,1\}^*$ it holds that*

$$|\mathsf{Pr}[\text{Exec}_{\#\#}(\pi, \mathcal{D}, \mathcal{Z})(n, a) = 1] - \mathsf{Pr}[\text{Exec}_{\#\#}(\phi, \mathcal{S}_\mathcal{D}, \mathcal{Z})(n, a) = 1]| \le \mathsf{negl}(n)$$

Proposition 1 (Completeness of the Dummy Adversary). *Let π and ϕ be protocols. Then, $\pi \underset{\#\#}{\geq} \phi$ if and only if π emulates ϕ with respect to the dummy adversary in the Fortified UC framework.*

Proposition 2 (Transitivity). *Let π_1, π_2, π_3 be protocols. If $\pi_1 \underset{\#\#}{\geq} \pi_2$ and $\pi_2 \underset{\#\#}{\geq} \pi_3$ then it holds that $\pi_1 \underset{\#\#}{\geq} \pi_3$.*

Definition 4 (En bloc Protocols and their Initial Fortification). *A protocol π is called* en bloc *if each protocol party has been combined with all of its subparties (i.e. they all have the same PID) and π only uses* `standard`*-connections. Also, π only calls functionalities that immediately notify the adversary upon each input and let the adversary change inputs of adaptively corrupted parties.*

Furthermore, given an en bloc protocol π, define its initial fortification $\widetilde{\pi}$ *to be identical to π except that all* `standard`*-connections between parties with* different *PIDs and between a party and an ideal functionality are replaced by* `air-gap switches`*. Also, each* `air-gap switch` *is initially* disconnected*. Upon receiving input, each party immediately connects all of its* `air-gap switches`*.*

Theorem 1 (Equivalence with UC-emulation for en bloc Protocols and their Initial Fortification). *Let π, ϕ be en bloc protocols and $\widetilde{\pi}, \widetilde{\phi}$ their initial fortification. Then,*

$$\pi \underset{\#\#}{\geq} \phi \iff \pi \underset{\mathrm{UC}}{\geq} \phi \iff \widetilde{\pi} \underset{\#\#}{\geq} \widetilde{\phi}$$

Theorem 2 (Universal Composition). *Let π be a protocol, $\mathcal{F}$ be an ideal functionality (note that $\mathcal{F}$ may be fortified) and $\rho^{\mathcal{F}}$ a protocol in the $\mathcal{F}$-hybrid model. Then it holds that*

$$\pi \underset{\#\#}{\geq} \mathcal{F} \implies \rho^{\pi} \underset{\#\#}{\geq} \rho^{\mathcal{F}}$$

4 Construction for Non-reactive Functionalities

In this section, we will construct a general MPC protocol for every fortified functionality of a *non-reactive* functionality that is secure in our framework.

The broad idea is to have the parties send *encrypted shares* of their inputs in an *offline sharing phase* where they are unhackable and subsequently use these shares to compute the desired function in an *online compute phase.*

This, however, cannot be done straightforwardly. To begin with, in the offline phase, parties are not able to retrieve the relevant public keys themselves since this would necessitate going online, making them hackable. We therefore let parties send their shares to an *unhackable encryption unit* (Enc-unit) (via a *data diode*) which retrieves the relevant public keys and sends the encrypted shares to the designated receiver's (hackable) *buffer* (note that the parties are offline an can therefore not receive messages themselves).

Furthermore, each message to be sent has to be authenticated so that the adversary cannot modify it since this would allow him to change the input of the sending party. In particular, one must prevent him from changing the messages contained in the buffers he has hacked. One could do this by assuming an "authentication unit" that signs each ciphertext. However, such an authentication unit, since it has to be online, would have to be unhackable. Since we want to use as few unhackable hardware modules as possible, we take a different approach. We let each party sign its shares and have the Enc-unit encrypt these shares together with their signatures. Note that, in general, signing-then-encrypting is not secure. Signing-then-encrypting is secure, however, if the public key encryption scheme is *non-malleable* and the digital signature scheme satisfies a property called *"length-normality"*. The latter means that the signatures of two messages of equal length are also of equal length (this prevents an adversary from learning information of the plaintexts based only on the length of their signatures). Each party sends its verification key to a (hackable) sub-party that after receiving the verification key disconnects itself from its main party and relays the verification key to a public bulletin board (via a *data diode*) together with its own PID. Once a party has sent all of its shares, it erases everything except for its own share and its verifcation key and goes online.

In the online compute phase, we must prevent the adversary from using values that are *different* from the shares that have been generated by the honest parties in the sharing phase as input to the multi-party computation. Otherwise, he would be able to change the inputs of the parties that have not been corrupted (but possibly hacked). We therefore require each input to be verified before computation. To this end, parties must input not only the shares but also the signatures of these shares (and the verification keys) into the multi-party computation where they will be checked for validity. Note that since the signing keys have been erased at the end of the offline phase, the adversary cannot generate new valid signatures for honest or hacked parties. He can also not revoke the verification key of a hacked party since this would require hacking the sub-party that registered the key, which is impossible since that party is offline.

Another problem to be taken care of is that an adversary could intercept a message in the sharing phase addressed to an honest party and *swap* that message with a ciphertext containing a share and signature received by a corrupted of hacked party. Moreover, an adversary who controls at least *two* parties knows two shares of each party along with their valid signatures and could use one of these shares *twice* in the multi-party computation. In order to prevent these attacks, we let a party sign each share *along with the PID of the designated receiver*. We also let each party include its own PID in each message it sends. This - along with non-malleability - prevents an adversary from reusing messages of honest parties for messages coming from corrupted parties (this would allow him to set the input of a corrupted party to be equal to the input of a hacked party).

Finally, we cannot simply send the result of the compute phase to a party since this party may have been hacked. Doing so would therefore allow the adversary to learn and change the output of the hacked party. Instead, we further modularize each party by introducing an *unhackable output interface machine* (OIM). To this end, we let each party i send 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. Each tuple of shares is signed along with the PID of the designated receiver. Furthemore, each party sends the random pad r_i and the MAC key k_i to its OIM (via a *data diode*). In the compute phase, the parties will then use these shares to compute the function $(y_i + r_i, \mathrm{Mac}(k_i, y_i + r_i))$, where y_i is the desired output value (of party i). A party can then send the result of the compute phase to its OIM. The OIM will then 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 and define an ideal functionality $\mathcal{F}_{\mathcal{G}}$ that implements the verification of the input values in the compute phase (i.e. checks the signatures of the shares) as well as the subsequent multi-party computation on the shares. Using Theorems 1 and 2, we will be able to realize this functionality with (existing) UC-secure protocols.

For simplicity, we assume perfect correctness for all of the following algorithms (cf. Appendix B). However, this is not necessary.

We first define the functionality $\mathcal{F}_{\mathcal{G}}$.

Construction 1
Let $\mathcal{G}$ *be a* non-reactive *ideal functionality.*
$\mathcal{F}_{\mathcal{G}}$ *proceeds as follows, running with parties* $P_1, \ldots, P_N$ *and an adversary* $\mathcal{A}$ *and parametrized with a digital signature* SIG *and a message authentication code* MAC*.*

1. *Upon receiving input* $\overline{\mathsf{vk}}_i = (\mathsf{vk}_1^{(i)}, \ldots, \mathsf{vk}_N^{(i)})$ *and* $(s_{ji}, r_{ji}, k_{ji}, \sigma_{ji})$ $(j = 1, \ldots, N)$ *from party* P_i*, store that input and send* $(\mathtt{received}, P_i)$ *to* $\mathcal{A}$*.*
2. *Once each party has sent its input, check if one party has sent* $\perp$*. If yes, output* $\perp$
3. *Else, check if* $\overline{\mathsf{vk}}_1 = \cdots = \overline{\mathsf{vk}}_N$*. If no, output* $\perp$
4. *Else, set* $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n) = (\mathsf{vk}_1^{(1)}, \ldots, \mathsf{vk}_N^{(1)})$*. For all* $i = 1, \ldots, N$*, check if* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, s_{ji}, r_{ji}, k_{ji}, \sigma_{ji}) = 1$ *for all* $j = 1, \ldots, N$*. If this does not hold, output* $\perp$
5. *Else, for each* $i = 1, \ldots, N$*, compute* $x_i = s_{i1} + s_{i2} + \cdots + s_{iN}$*,* $k_i = k_{i1} + k_{i2} + \cdots + k_{iN}$ *and* $r_i = r_{i1} + r_{i2} + \cdots + r_{iN}$*.*
6. *Internally run* $\mathcal{G}$ *on input* $(x_1, \ldots, x_N)$*. Let* $(y_1, \ldots, y_N)$ *be the output of* $\mathcal{G}$*.*
7. *For all* $i = 1, \ldots, N$*, compute* $o_i = y_i + r_i$ *and* $\theta_i \leftarrow \mathrm{Mac}(k_i, y_i + r_i)$*.*
8. *For all* $i = 1, \ldots, N$*, send a public delayed output* (o_i, θ_i) *to* P_i*.*

C *If* $\mathcal{A}$ *sends* $(\mathtt{corrupt}, P)$ *and* $\mathcal{F}_{\mathcal{G}}$ *has already received an input from party* P *then* $\mathcal{F}_{\mathcal{G}}$ *sends that input to* $\mathcal{A}$*. Otherwise* $\mathcal{F}_{\mathcal{G}}$ *sends "no input yet". Furthermore,* $\mathcal{F}_{\mathcal{G}}$ *lets* $\mathcal{A}$ *determine the input of party* P*.*

Next, we define our protocol, which is in the $(F_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}})$-hybrid model.

Construction 2 *Define the protocol* $\rho^{F_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}}$ *as follows:*
Architecture: *(cf. Fig. 1 in Appendix A) Each party has two hackable and two unhackable sub-parties. The hackable sub-parties are a* buffer *and a* registration

machine*, and the unhackable sub-parties are an* Enc-*unit and an* OIM*. Each party has an* `air-gap switch` *at its input port, an* `air-gap switch` *to its* buffer*, a* `data diode` *to its* OIM*, an* `air-gap switch` *and a* `data diode` *to its* Enc-*unit, an* `air-gap switch` *to* $\mathcal{F}_{\mathsf{reg}}$*, an* `air-gap switch` *to* $\mathcal{F}_{\mathcal{G}}$*, and an* `air-gap switch` *to the network. Furthermore, each* Enc-*unit has a* `standard`-*connection to* $\mathcal{F}_{\mathsf{krk}}$ *and a* `standard`-*connection to the network, each* buffer *has a* `standard`-*connection to the network and each* registration machine *has an* `air-gap switch` *to its main party and a* `data diode` *to* $\mathcal{F}_{\mathsf{reg}}$*. Apart from the parties' input port and the* registration machines' *connection to their main parties, all air-gap switches are* disconnected *at the beginning.*

- Offline Sharing Phase:
  Upon input x_i*, each party* P_i *does the following:*
  - Disconnect *at the input port.*
  - *Generate shares* $s_{i1} + s_{i2} + \cdots + s_{iN} = x_i$
  - *Generate* $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ *and* $(\mathsf{sgk}_i, \mathsf{vk}_i) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$
  - *Generate shares* $k_{i1} + k_{i2} + \cdots + k_{iN} = k_i$
  - *Generate a random pad* $r_i \leftarrow \{0,1\}^{p_i(n)}$ *and generate shares* $r_{i1} + r_{i2} + \cdots + r_{iN} = r_i$
  - *Send* (k_i, r_i) *to the* OIM *and the verification key* vk_i *to the* registration machine*. The* registration machine *will then* disconnect *itself from its main party and relay* vk_i *to* $\mathcal{F}_{\mathsf{reg}}$ *(using its own PID).*
  - *Create signatures* $\sigma_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, s_{ij}, r_{ij}, k_{ij})$ $(j = 1, \ldots, N)$
  - *Iteratively send* $(j, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij})$ $(j \in \{1, 2, \ldots, m\} \backslash \{i\})$ *to the* Enc-*unit (at each activation)*
  - *At first activation, the* Enc-*unit requests a key pair* $(\mathsf{pk}_i, \mathsf{sk}_i)$ *from* $\mathcal{F}_{\mathsf{krk}}$*.*
  - *Upon receiving a tuple* $(j, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij})$*, the* Enc-*unit requests the public key* pk_j *belonging to party* P_j *from* $\mathcal{F}_{\mathsf{krk}}$*. If* pk_j *does not exist yet, the* Enc-*unit sends a "request public key message" to the* Enc-*unit of* P_j*. Otherwise, it computes* $c_j^i \leftarrow \mathrm{Enc}(\mathsf{pk}_j, i, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij})$ *and sends* (i, c_j^i)[1] *to party* P_j*.*
  - *Once all shares have been sent to the* Enc-*unit,* erase *everything except for the tuple* $(s_{ii}, r_{ii}, k_{ii}, \sigma_{ii})$ *and the verification key* vk_i *(in particular, the input* x_i*, signing key* sk_i*, random pad* r_i *and MAC key* k_i *are erased).*

- Online Compute Phase:
  Once the last step in the offline sharing phase is completed, a party P_i *does the following:*
  - Connect *to the* buffer*, to the* Enc-*unit and to* $\mathcal{F}_{\mathsf{reg}}$*.*
  - *Request the secret key* sk_i *from the* Enc-*unit.*
  - *Request all verification keys* $\{\mathsf{vk}_l\}_{l \in \{1,\ldots,N\} \backslash \{i\}}$ *that were registered with the PIDs of the other parties'* registration machines *from* $\mathcal{F}_{\mathsf{reg}}$*. If not all verification keys can be retrieved yet, go into idle mode and request again at the next activation.*

[1] Sending the PID i of the sender as a prefix in the clear is not necessary but simplifies the following discussion. Note that for (i, c), we also say that c is addressed as coming from party i.

- At each activation, check if there are at least $N-1$ messages in its buffer. If no, go into idle mode and when activated again check again.
  If yes, check if one has received from each party j a set $\mathcal{M}_j = \{(j, \tilde{c})\}$ with the following property:
  There exists a tuple $(j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$ and an element $(j, c) \in \mathcal{M}_j$ such that $(*)$:
  - $*$ $\mathrm{Dec}(\mathsf{sk}_i, c) = (j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$
  - $*$ $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji}) = 1$
  - $*$ For every $(j, \tilde{c}) \in \mathcal{M}_j$ it holds that either $\mathrm{Dec}(\mathsf{sk}_i, \tilde{c}) = (j, \hat{s}_{ji}, \hat{r}_{ji}, \hat{k}_{ji}, \hat{\sigma}_{ji})$ or $(j, \tilde{c})$ is "invalid", i.e., either decrypts to $(j, \tilde{s}_{ji}, \tilde{r}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}_{ji})$ such that $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, \tilde{s}_{ji}, \tilde{r}_{ji}, \tilde{k}_{ji}, \hat{\sigma}_{ji}) = 0$, or decrypts to $(l, \tilde{s}_{ji}\tilde{r}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}_{ji})$ where $l \neq j$, or does not decrypt correctly.

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

– Online Output Phase:
Upon receiving an output from $\mathcal{F}_{\mathcal{G}}$, a party P_i does the following:
- Connect its input port.
- *If this output equals $\perp$, it sends $\perp$ to the* OIM, *which then outputs* $\perp$
- *Otherwise, let (o_i, θ_i) be the output from $\mathcal{F}_{\mathcal{G}}$. Send this tuple to the* OIM.
- *The* OIM *then checks if* $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, o_i, \theta_i) = 1$ *and outputs* $y_i = o_i + r_i$ *if this holds, and $\perp$ otherwise.*

Before stating the theorem, we define the following auxiliary experiment, which will be used in the proof.

Definition 5 (Auxiliary Experiment). *The experiment* $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$ *is defined as follows: At the beginning, the experiment generates keys* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$ *and* $(\mathsf{vk}, \mathsf{sgk}) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$. *On input* $1^n, z$ *and* pk, *the adversary* $\mathcal{A}$ *may then* non-adaptively *send queries to a signing oracle* $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$. *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* $\sigma \leftarrow \mathrm{Sig}(\mathsf{sgk}, \mathtt{prf}_2, m)$, $c^* \leftarrow \mathrm{Enc}(\mathsf{pk}, \mathtt{prf}_1, m, \sigma)$, *and sends* c^* *to* $\mathcal{A}$. *Throughout the experiment,* $\mathcal{A}$ *has access to a decryption oracle* $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$ *subject to the restriction that the queries to* $\mathrm{Dec}(\mathsf{sk}, \cdot)$ *are* non-adaptive *(i.e. parallel) and do not contain* c^*. *At the end of the experiment,* $\mathcal{A}$ *sends a tuple* (m', σ') *to the experiment. The experiment then checks if* $\mathrm{Vrfy}_{\mathrm{SIG}}(vk, m', \sigma') = 1$ *and* m' *has not been sent to* $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$ *before. If this holds, the experiment outputs* 1 *and* 0 *otherwise.*

We have the following lemma. The proof is straightforward (cf. Appendix F).

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*

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

We will use the above experiment to show that an environment $\mathcal{Z}$ cannot send "fake messages" (i, c') addressed as coming from a party i that has *not* been corrupted (but possibly hacked) such that c' was not generated by party i but (i, c') is accepted by an honest party j. Otherwise, one could build a successful adversary $\mathcal{A}$ in $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$: $\mathcal{A}$ guesses indices i, j such that $\mathcal{Z}$ sends a fake message (i, c') to party j. $\mathcal{A}$ simulates the protocol execution for $\mathcal{Z}$. For party i, $\mathcal{A}$ sends the tuples $(l, s_{il}, r_{il}, k_{il})$ for $l \neq j$ to $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$ and $(\mathtt{i}, \mathtt{j}, s_{ij}, r_{ij}, k_{ij})$ to the experiment, receiving c^*. $\mathcal{A}$ then uses (i, c^*) for (i, c^i_j) in its simulation. If $\mathcal{A}$'s guess is correct, $\mathcal{A}$ can decrypt c' using the decryption oracle $\mathcal{O}_{\mathrm{Dec}(\mathsf{sk},\cdot)}$, obtaining a message (i, m', σ'). $\mathcal{A}$ can then send (j, m', σ') to the experiment. $\mathcal{A}$ then wins because he has never sent a message of the form $(\mathtt{j}, m)$ to $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$. Note that if $\mathcal{A}$ had also sent $(j, s_{ij}, r_{ij}, k_{ij})$ to $\mathcal{O}_{\mathrm{Sig}(\mathsf{sgk},\cdot)}$, then he would not win if c' decrypts to the same plaintext as c^*, which happens if $\mathcal{Z}$ manages to break the non-malleability of PKE.

Next, we define the simulator to be used in the proof.

Definition 6 (Simulator for up to $N - 1$ Corruptions/Hacks, Non-Reactive Case). *Define the simulator* Sim *interacting with an environment* $\mathcal{Z}$ *and the ideal functionality* $[\mathcal{G}]$ *as follows:*

- Sim *generates* $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$ *for each party* i*.*
- Sim *generates* $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ *and* $(\mathsf{sgk}_i, \mathsf{vk}_i) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$ *for each party* i *that has not been corrupted.*
- Sim extracts *the inputs of the* corrupted *parties by decrypting all ciphertexts coming from* $\mathcal{Z}$ *(note that* Sim *can do this because he knows all secret keys) and looking at the inputs* $\mathcal{Z}$ *sends to* $\mathcal{F}_{\mathcal{G}}$ *for corrupted parties.* Sim *sends these inputs to* $[\mathcal{G}]$*.*
- *Each time* Sim *is activated by* $[\mathcal{G}]$ *after an* honest *party received its input,* Sim *generates* $3N$ *random strings* $s'_{ij}, r'_{ij}, k'_{ij}$*, computes* $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, s'_{ij}, r'_{ij}, k'_{ij})$ $(j = 1, \ldots, N)$ *and* $c^i_j \leftarrow \mathrm{Enc}(\mathsf{pk}_j, i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$*.* Sim *then iteratively reports* (i, c^i_j) $(j \in \{1, \ldots, N\} \setminus \{i\})$ *to* $\mathcal{Z}$ *(at each activation of party* i*).*
- *If* $\mathcal{Z}$ *requests the public key or verification key of an* honest *party,* Sim *sends the respective key to* $\mathcal{Z}$ *if that party has already received its input. If* $\mathcal{Z}$ *requests the public key or verification key or secret key of a* hacked *party,* Sim *sends the respective key to* $\mathcal{Z}$*. If* $\mathcal{Z}$ *requests the public key and secret key of a* corrupted *party, then* Sim *sends the respective key pair to* $\mathcal{Z}$ *if* $\mathcal{Z}$ *has sent a* `register`*-message addressed to* $\mathcal{F}_{\mathsf{krk}}$ *for that party before.*
- *If an honest party* j *is activated (in* Sim*'s internal simulation) and has sent all its shares and has received at least* $N - 1$ *messages (in its buffer),* Sim *checks if the following two conditions hold:*
  - *party* j *has received* all *the* (i, c^i_j) *that were sent by the* Enc*-unit of the parties that have not been corrupted (but possibly hacked).*
  - *party* j *has received from each* corrupted *party* l *a set* $\mathcal{M}_l$ *fulfilling property* $(*)$ *(see Page 14).*

If these two conditions hold, Sim *marks this party as* `genuine`*. Otherwise,* Sim *marks this party as* `fake`*. In both cases,* Sim *continues the simulation as if* $\mathcal{F}_{\mathcal{G}}$ *had received an input from party* j*.*

- *If* $\mathcal{Z}$ *sends a tuple* $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ *as the input of a* corrupted *or* hacked *party* j *to* $\mathcal{F}_{\mathcal{G}}$ *such that* $(s'_{ij}, r'_{ij}, k'_{ij}) \neq (s_{ij}, r_{ij}, k_{ij})$*, where* (s_{ij}, r_{ij}, k_{ij}) *was generated (in* Sim*'s internal simulation) by a party* i *that has* not *been corrupted (but possibly hacked), then* Sim *marks party* j *as* `fake`*. Otherwise,* Sim *verifies the signature of this input. If* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$*, then* Sim *marks this party as* `genuine`*. Otherwise,* Sim *marks this party as* `fake`*. In both cases,* Sim *continues the simulation as if* $\mathcal{F}_{\mathcal{G}}$ *had received an input from party* j*.*
- *If a party in* Sim*'s internal simulation expects an output from* $\mathcal{F}_{\mathcal{G}}$ *and* all *parties are marked as* `genuine`*, then* Sim *does the following:*
  - *For an* honest *party,* Sim *instructs* $[\mathcal{G}]$ *to send the output.*
  - *For a* hacked *party* i*,* Sim *first 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* $\mathcal{Z}$*. If* $\mathcal{Z}$ *sends a message* (m', t') *addressed to the OIM of that party, then*
    - *If* $m' \neq \tilde{y}_i$*,* Sim *instructs* $[\mathcal{G}]$ *to output* $\perp$ *to party* i
    - *If* $m' = \tilde{y}_i$*, then* Sim *verifies if* $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 1$*. If this holds, then* Sim *instructs* $[\mathcal{G}]$ *to send the output to party* i*. Otherwise,* Sim *instructs* $[\mathcal{G}]$ *to output* $\perp$ *to party* i
  - *For a* corrupted *party* i*,* Sim *first generates a random string* $\tilde{y}_i \leftarrow \{0,1\}^n$ *and sends* $(\tilde{y}_i, \mathrm{Mac}(k_i, \tilde{y}_i))$ *to* $\mathcal{Z}$*.* Sim *then lets* $\mathcal{Z}$ *determine the output.*
- *If a party in* Sim*'s internal simulation expects an output from* $\mathcal{F}_{\mathcal{G}}$ *and one of the parties is marked as* `fake`*, then* Sim *does the following:*
  - *For the* honest *parties,* Sim *instructs* $[\mathcal{G}]$ *to output* $\perp$*.*
  - *For a* hacked *party* i*,* Sim *first sends* $\perp$ *to* $\mathcal{Z}$*.* Sim *then waits for* $\mathcal{Z}$*'s response addressed to the OIM of that party and after receiving that response instructs* $[\mathcal{G}]$ *to output* $\perp$ *to party* i*.*
  - *For a* corrupted *party* i*,* Sim *first sends* $\perp$ *to* $\mathcal{Z}$ *to* $\mathcal{Z}$*.* Sim *then lets* $\mathcal{Z}$ *determine the output.*
- *Each time* $\mathcal{Z}$ *sends* `status`*,* Sim *sends the status of all simulated parties.*

We are now ready to state our theorem:

Theorem 3 (Up to $N-1$ Corruptions/Hacks, Non-Reactive Functionalities).
Let $\mathcal{G}$ *be a* non-reactive *functionality.*
Let $\mathrm{PKE} = (\mathrm{Gen}_{\mathrm{PKE}}, \mathrm{Enc}, \mathrm{Dec})$ *be a IND-pCCA-secure PKE,* $\mathrm{SIG} = (\mathrm{Gen}_{\mathrm{SIG}}, \mathrm{Sig}, \mathrm{Vrfy}_{\mathrm{SIG}})$ *an EUF-naCMA-secure and length-normal DigSig and* $\mathrm{MAC} = (\mathrm{Gen}_{\mathrm{MAC}}, \mathrm{Mac}, \mathrm{Vrfy}_{\mathrm{MAC}})$ *an EUF-1-CMA-secure MAC.*

Then it holds that $\rho^{F_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}} \underset{\#\#}{\geq} [\mathcal{G}]$ *for up to* $N-1$ *corruptions/hacks.*

Proof. By Proposition 1, it suffices to find a simulator for the dummy adversary.

The main idea of the proof is to consider a sequence of hybrids $\mathrm{H}_0, \ldots, \mathrm{H}_4$, each of which defines an ideal protocol that grants the simulator certain actions, i.e. learn/change the inputs/outputs of certain parties. Starting from an ideal protocol that gives the simulator maximal leverage (i.e. just sends all inputs to him and lets him determine each output), we will gradually reduce the simulators possibilties. The final hybrid H_4 will be the ideal protocol with functionality $[\mathcal{G}]$ and the simulator as defined in Definition 6.

Let $\mathcal{Z}$ be an environment that corrupts or hacks at most $N-1$ parties. Let $\mathrm{out}_i(\mathrm{Z})$ be the output of the environment $\mathcal{Z}$ in the hybrid H_i.

Hybrid H_0 Let H_0 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol with functionality $\mathcal{F}_0$ and the adversary Sim_0, where $\mathcal{F}_0$ and Sim_0 are defined as follows:

$\mathcal{F}_0$ is defined to be the ideal functionality that simply forwards the inputs and outputs of *all* parties to the adversary and lets the adversary determine the inputs and outputs of *all* parties. Furthermore, $\mathcal{F}_0$ is *initially offline*.

Define Sim_0 to be the ideal-model adversary that simulates the entire protocol $\rho^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}}$ for $\mathcal{Z}$. Sim_0 can do this because he is given all inputs and outputs and can change every input and determine each output.

Since all `air-gap switches` in $\rho^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}}$ are *disconnected* at the beginning, apart from the parties' input ports (and the *registration machines'* connection to their main parties), it holds that the views of $\mathcal{Z}$ in the real-model execution and in H_0 are identically distributed, hence

$$|\mathsf{Pr}[\mathrm{Exec}_{\#\#}\big(\rho^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}},\mathcal{D},\mathcal{Z}\big)=1]-\mathsf{Pr}[\mathrm{out}_0(\mathcal{Z})=1]|=0$$

Hybrid H_1 Let H_1 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol with functionality $\mathcal{F}_1$ and the adversary Sim_1, where $\mathcal{F}_1$ and Sim_1 are defined as follows:

Define $\mathcal{F}_1$ to be identical to $\mathcal{F}_0$ except that now the adversary is allowed to *determine the inputs* only of *corrupted* parties and *determine the outputs* only of *corrupted and hacked* parties (note that the adversary is still given all inputs and outputs).

Define the ideal-model adversary Sim_1 to be like Sim_0 except for the following:

- Sim_1 generates $(\mathsf{pk}_i,\mathsf{sk}_i)\leftarrow\mathrm{Gen}_{\mathrm{PKE}}(1^n)$ for each party i.
- Sim_1 generates $k_i\leftarrow\mathrm{Gen}_{\mathrm{MAC}}(1^n)$ and $(\mathsf{sgk}_i,\mathsf{vk}_i)\leftarrow\mathrm{Gen}_{\mathrm{SIG}}(1^n)$ for each party i that has not been corrupted.
- Sim_1 *extracts* the inputs of the *corrupted* parties by decrypting all ciphertexts coming from $\mathcal{Z}$ (note that Sim_1 can do this because he knows all secret keys) and looking at the inputs $\mathcal{Z}$ sends to $\mathcal{F}_{\mathcal{G}}$ for corrupted parties. Sim_1 sends these inputs to $\mathcal{F}_1$.
- If $\mathcal{Z}$ requests the public key or verification key of an *honest* party, Sim sends the respective key to $\mathcal{Z}$ if that party has already received its input. If $\mathcal{Z}$ requests the public key or verification key or secret key of a *hacked* party, Sim sends the respective key to $\mathcal{Z}$. If $\mathcal{Z}$ requests the public key and secret key of a *corrupted* party, then Sim sends the respective key pair to $\mathcal{Z}$ if $\mathcal{Z}$ has sent a `register`-message addressed to $\mathcal{F}_{\mathsf{krk}}$ for that party before.

- If an honest party j is activated (in Sim_1's internal simulation) and has sent all its shares and has received at least $N-1$ messages (in its buffer), Sim_1 checks if the following two conditions hold:
  - party j has received *all* the (i, c^i_j) that were sent by the Enc-unit of the parties that have not been corrupted (but possibly hacked).
  - party j has received from each *corrupted* party l a set $\mathcal{M}_l$ fulfilling property $(*)$.

  If these two conditions hold, Sim_1 marks this party as `genuine`. Otherwise, Sim_1 marks this party as `fake`. In both cases, Sim_1 continues the simulation as if $\mathcal{F}_{\mathcal{G}}$ had received an input from party j.
- If $\mathcal{Z}$ sends a tuple $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ as the input of a *corrupted* or *hacked* party j to $\mathcal{F}_{\mathcal{G}}$ such that $(s'_{ij}, r'_{ij}, k'_{ij}) \neq (s_{ij}, r_{ij}, k_{ij})$, where (s_{ij}, r_{ij}, k_{ij}) was generated (in Sim_1's internal simulation) by a party i that has *not* been corrupted (but possibly hacked), then Sim_1 marks party j as `fake`. Otherwise, Sim_1 verifies the signature of this input. If $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$, then Sim_1 marks this party as `genuine`. Otherwise, Sim_1 marks this party as `fake`. In both cases, Sim_1 continues the simulation as if $\mathcal{F}_{\mathcal{G}}$ had received an input from party j.
- If a party in Sim_1's internal simulation expects an output from $\mathcal{F}_{\mathcal{G}}$ and *all* parties are marked as `genuine`, then Sim_1 does the following:
  - For an *honest* party, Sim_1 instructs $\mathcal{F}_1$ to send the output.
  - For a *hacked* party i, Sim_1 first sends $(y_i + r_i, \mathrm{Mac}(k_i, y_i + r_i))$ to $\mathcal{Z}$. If $\mathcal{Z}$ responds with a tuple (m', t') such that $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 1$, then Sim_1 instructs $\mathcal{F}_1$ to output $m' + r_i$ to the hacked party i. If $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 0$, Sim_1 instructs $\mathcal{F}_1$ to output $\perp$ to party i.
  - For a *corrupted* party i, Sim_1 first generates a random string $\tilde{y}_i \leftarrow \{0,1\}^n$ and sends $(\tilde{y}_i, \mathrm{Mac}(k_i, \tilde{y}_i))$ to $\mathcal{Z}$. Sim_1 then lets $\mathcal{Z}$ determine the output.
- If a party in Sim_1's internal simulation expects an output from $\mathcal{F}_{\mathcal{G}}$ and one of the parties is marked as `fake`, then Sim_1 does the following:
  - For an *honest* party, Sim_1 instructs $\mathcal{F}_1$ to output $\perp$.
  - For a *hacked* party i, Sim_1 first sends $\perp$ to $\mathcal{Z}$. If $\mathcal{Z}$ responds with a tuple (m', t') such that $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 1$, then Sim_1 instructs $\mathcal{F}_1$ to output $m' + r_i$ to the hacked party i. If $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, m', t') = 0$, Sim_1 instructs $\mathcal{F}_1$ to output $\perp$ to party i.
  - For a *corrupted* party i, Sim_1 first sends $\perp$ to $\mathcal{Z}$ to $\mathcal{Z}$. Sim_1 then lets $\mathcal{Z}$ determine the output.
- Each time $\mathcal{Z}$ sends `status`, Sim_1 sends the status of all simulated parties.

Consider the following events:

Let $\mathbf{E}_{\mathrm{fakemess}}$ be the event that there exists an *honest* party j that fetches a tuple (i, c') in its (possibly hacked) buffer such that party i has *not* been corrupted (but possibly hacked) and $\mathrm{Dec}(\mathsf{sk}_j, c') = (i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ and $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$ but either $c' \neq c^i_j$ or c^i_j has not been generated yet by party i.

Let $\mathbf{E}_{\mathrm{fakeinp}}$ be the event that $\mathcal{Z}$ sends an input $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ for a *corrupted* or *hacked* party j to $\mathcal{F}_{\mathcal{G}}$ such that $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$ but

$(s'_{ij}, r'_{ij}, k'_{ij}) \neq (s_{ij}, r_{ij}, k_{ij})$, where (s_{ij}, r_{ij}, k_{ij}) was generated by a party i that has *not* been corrupted (but possibly hacked).

Let $\mathbf{E} = \mathbf{E}_{\text{fakemess}} \cup \mathbf{E}_{\text{fakeinp}}$. 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 j that is addressed as coming from a party i who has *not* been corrupted (but possibly hacked) decrypts to a valid message/signature pair if and only if it equals the ciphertext c^i_j sent by party i. Moreover, for each *corrupted* or *hacked* party i, since $\mathbf{E}_{\text{fakeinp}}$ does not occur, $\mathcal{Z}$ only sends inputs $(s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ to $\mathcal{F}_{\mathcal{G}}$ such that either $\text{Vrfy}_{\text{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 0$ or $\text{Vrfy}_{\text{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$ and $(s'_{ij}, r'_{ij}, k'_{ij}) = (s_{ij}, r_{ij}, k_{ij})$ was generated by party i (who has not been corrupted).

Therefore, it holds that

$$|\mathsf{Pr}[\text{out}_0(\mathcal{Z}) = 1] - \mathsf{Pr}[\text{out}_1(\mathcal{Z}) = 1]| \leq \mathsf{Pr}[\mathbf{E}] \leq \mathsf{Pr}[\mathbf{E}_{\text{fakemess}}] + \mathsf{Pr}[\mathbf{E}_{\text{fakeinp}}]$$

Claim 1: $\mathsf{Pr}[\mathbf{E}_{\text{fakemess}}]$ *is negligible.*
Consider the following adversary $\mathcal{A}$ in the auxiliary experiment $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\text{PKE},\text{SIG}}(n)$: At the beginning, $\mathcal{A}$ randomly selects a tuple $(i, j) \in \{1, \ldots, N\} \times \{1, \ldots, N\} \setminus \{i\}$. $\mathcal{A}$ 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 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 $(l, s_{il}, r_{il}, k_{il})$ for $l \neq j$ to the signing oracle $\mathcal{O}_{\text{Sig}(\mathsf{sgk},\cdot)}$, 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 $(l, s_{il}, r_{il}, k_{il}, \sigma_{il})$ $(l \notin \{i, j\})$ and sends them to the respective party in its internal simulation. Once the message (i, c^i_j) is supposed to be sent in the internal simulation, $\mathcal{A}$ sends $(\mathtt{i}, \mathtt{j}, s_{ij}, r_{ij}, k_{ij})$ to the experiment, receiving c^*. $\mathcal{A}$ then uses (i, c^*) for (i, c^i_j) in its simulation. When party j is activated and has sent all its shares and has received at least $N-1$ messages, $\mathcal{A}$ sends all ciphertexts addressed as coming from party i such that $c \neq c^*$ to the decryption oracle $\mathcal{O}_{\text{Dec}(\mathsf{sk},\cdot)}$ (if c^* has not been generated yet, $\mathcal{A}$ sends all ciphertexts addressed as coming from party i). For each message $(\tilde{i}, m, \sigma)$ he receives from the oracle $\mathcal{O}_{\text{Dec}(\mathsf{sk},\cdot)}$, $\mathcal{A}$ checks if $\text{Vrfy}_{\text{SIG}}(vk, j, m, \sigma) = 1$. If this holds for a message (i', m', σ'), then $\mathcal{A}$ sends (j, m', σ') to the experiment. If during the simulation, $\mathcal{Z}$ corrupts party i or corrupts or hacks party j or if no message $\mathcal{A}$ receives from $\mathcal{O}_{\text{Dec}(\mathsf{sk},\cdot)}$ 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 $\mathcal{O}_{\text{Dec}(\mathsf{sk},\cdot)}$ such that $c \neq c^*$ or c^* has not been generated yet and $\text{Dec}(\mathsf{sk}, c') = (i, m', \sigma')$ and $\text{Vrfy}_{\text{SIG}}(vk, j, m', \sigma') = 1$. Since $\mathcal{A}$ does not send a message of the form $(\mathtt{j}, m)$ to the signing oracle $\mathcal{O}_{\text{Sig}(\mathsf{sgk},\cdot)}$, it follows that $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\text{PKE},\text{SIG}}(n) = 1$. 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,

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

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

Claim 2: $\mathsf{Pr}[\mathbf{E}_{\mathrm{fakeinp}}]$ *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 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 $(j, s_{ij}, r_{ij}, k_{ij})$ 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 $(i, s_{ij}, r_{ij}, k_{ij}, \sigma_{ij})$ $(j = 1, \ldots, N)$ 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* or *hacked* party 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 $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_i, j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}) = 1$. If this holds, $\mathcal{A}$ sends $(j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij})$ to the experiment. If during the simulation, $\mathcal{Z}$ corrupts party i or if no message $\mathcal{A}$ checks is valid, then $\mathcal{A}$ sends $\perp$ to the experiment.

By construction, it holds that if $\mathbf{E}_{\mathrm{fakeinp}}$ occurs and $\mathcal{A}$ has correctly guessed an index i for which $\mathbf{E}_{\mathrm{fakeinp}}$ occurs, then $\mathsf{Exp}^{\text{euf-nacma}}_{\mathcal{A}(z),\mathrm{SIG}}(n) = 1$ because $(j, s'_{ij}, r'_{ij}, k'_{ij}, \sigma_{ij})$ is valid and $(j, s'_{ij}, r'_{ij}, k'_{ij}) \neq (j, s_{ij}, r_{ij}, k_{ij})$ 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 $\mathbf{E}_{\mathrm{fakeinp}}$ occurs is at least $1/N$. Hence,

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

Therefore, since $\mathsf{Pr}[\mathsf{Exp}^{\text{euf-nacma}}_{\mathcal{A}(z),\mathrm{SIG}}(n) = 1]$ is negligible by assumption and N is polynomial in n, it follows that $\mathsf{Pr}[\mathbf{E}_{\mathrm{fakeinp}}]$ 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 with functionality $\mathcal{F}_1$ (again) and the adversary Sim_2, where Sim_2 is defined as follows:

Define the ideal-model adversary Sim_2 to be like Sim_1 except for the following: For every *honest* party i, Sim_2 generates N random strings k'_{ij} and computes $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, s_{ij}, r_{ij}, k'_{ij})$ $(j = 1, \ldots, N)$, 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 $(i, \mathrm{Enc}(\mathsf{pk}_j, i, s_{ij}, r_{ij}, k'_{ij}, \sigma'_{ij}))$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ to $\mathcal{Z}$. Sim_2 still uses $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ as MAC key for the output of $\mathcal{F}_{\mathcal{G}}$ to a *hacked* party i (if that output is $\neq \perp$).

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

Define the ideal-model adversaries $\mathsf{Sim}_{2,i}$ to be like Sim_1 except for the following: For every *honest* party $l \in \{1, \ldots, i\}$, $\mathsf{Sim}_{2,i}$ generates N random strings k'_{lj}, computes $\sigma'_{lj} \leftarrow \mathrm{Sig}(\mathsf{sgk}_l, j, s_{lj}, r_{lj}, k'_{lj})$ $(j = 1, \ldots, N)$, and iteratively reports $(l, \mathrm{Enc}(\mathsf{pk}_j, l, s_{lj}, r_{lj}, k'_{lj}, \sigma'_{lj}))$ $(j \in \{1, \ldots, N\} \setminus \{l\})$ to $\mathcal{Z}$.

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 i^* is *not hacked*, i.e. if it is corrupted 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,

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

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 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 σ_{i^*j}, σ'_{i^*j} for $(j, s_{i^*j}, r_{i^*j}, k_{i^*j})$ and $(j, s_{i^*j}, r_{i^*j}, k'_{i^*j})$, respectively, and sends $(i^*, s_{i^*j}, r_{i^*j}, k_{i^*j}, \sigma_{i^*j})$, $(i^*, s_{i^*j}, r_{i^*j}, k'_{i^*j}, \sigma'_{i^*j})$ 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 $c_j^{i^*}$ and its decryption oracle to decrypt the ciphertexts in the buffer of party j that are addressed as coming from the *corrupted* parties but do not equal c^* (the ones that are equal to c^* are ignored. Note that a tuple (l, c^*) sent by a corrupted party l is always invalid since $l \neq i^*$). Note that in $\mathcal{A}$'s internal simulation, party 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 $\perp$). At the end of the experiment, $\mathcal{A}$ outputs what $\mathcal{Z}$ outputs. If during the simulation, $\mathcal{Z}$ corrupts or hacks party j or if party i^* is *not* hacked, i.e. if it is corrupted or remains honest throughout the execution, then $\mathcal{A}$ sends $\perp$ to the experiment.

Let $\mathrm{output}_b(\mathcal{A}) = 1$ denote the output of $\mathcal{A}$ in the IND-pCCA experiment when the challenge bit b is chosen. By construction, assuming party i^* is hacked, if $\mathcal{A}$ guessed an index j of an honest party 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 i^* is hacked, the probability that $\mathcal{A}$ guesses an index j of an honest party is at least $1/(N-1)$. Hence,

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

This contradicts the IND-pCCA security of PKE.

Hence, there exist a negligible function negl_2 such that

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

Hybrid H_3 Let H_3 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol with functionality $\mathcal{F}_2$ and the 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 *corrupted* parties.

Define the ideal-model adversary Sim_3 to be like Sim_2 except for the following: If $\mathcal{Z}$ receives an output from $\mathcal{F}_{\mathcal{G}}$ for a *hacked* party i then,

- If this outputs equals $\perp$, Sim_3 first sends $\perp$ to $\mathcal{Z}$. Sim_3 then waits for $\mathcal{Z}$'s response addressed to the OIM of that party and after receiving that response instructs $\mathcal{F}_2$ to output $\perp$ to party i.
- Otherwise, i.e if this output equals (m,t), if $\mathcal{Z}$ sends a response (m',t') addressed to the OIM of that party then
  - if $m'\neq m$, Sim_3 instructs the $\mathcal{F}_2$ to output $\perp$ to party i
  - if $m'=m$, then Sim_3 verifies if $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i,m',t')=1$. If this holds, then Sim_3 instructs $\mathcal{F}_2$ to send the output to party i. Otherwise, Sim_3 instructs $\mathcal{F}_2$ to output $\perp$ to party i

Let $\mathbf{E}_{\mathrm{fakeoutp}}$ be the event that $\mathcal{Z}$ sends a message (m',t') to the OIM of a *hacked* party i such that $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i,m',t')=1$ but either party i has received $\perp$ from $\mathcal{F}_{\mathcal{G}}$ or (m,t) such that $m'\neq m$, or party i has not received an output from $\mathcal{F}_{\mathcal{G}}$ yet.

It is easy to see that the following holds:

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

Therefore, it holds that

$$|\mathsf{Pr}[\mathrm{out}_2(\mathcal{Z})=1]-\mathsf{Pr}[\mathrm{out}_3(\mathcal{Z})=1]|\leq\mathsf{Pr}[\mathbf{E}_{\mathrm{fakeoutp}}]$$

Claim 3: $\mathsf{Pr}[\mathbf{E}_{\text{fakeoutp}}]$ *is negligible.*
Consider the adversary $\mathcal{A}$ against the EUF-1-CMA-security of MAC. At the beginning, $\mathcal{A}$ randomly selects an index $i \in \{1, \ldots, N\}$. $\mathcal{A}$ then simulates the hybrid H_2. Once $\mathcal{Z}$ expects the output from $\mathcal{F}_{\mathcal{G}}$ for (the *hacked*) party i, $\mathcal{A}$ 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 the OIM of party i such that $m' \neq m$, then $\mathcal{A}$ sends (m', t') to the experiment. If during the simulation, $\mathcal{Z}$ does not *hack* party i or if $\mathcal{Z}$ sends $\perp$ or a tuple (m', t') such that $m' = m$ to the OIM of party i, then $\mathcal{A}$ sends $\perp$ to the experiment.

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

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

Therefore, since $\mathsf{Pr}[\mathsf{Exp}^{\text{euf-1-cma}}_{\mathcal{A}(z),\mathrm{MAC}}(n) = 1]$ is negligible by assumption and N is polynomial in n, it follows that $\mathsf{Pr}[\mathbf{E}_{\text{fakeoutp}}]$ is also negligible.

Hence, there exist a negligible function negl_3 such that

$$|\mathsf{Pr}[\mathrm{out}_2(\mathcal{Z}) = 1] - \mathsf{Pr}[\mathrm{out}_3(\mathcal{Z}) = 1]| \leq \mathsf{negl}_3(n)$$

Hybrid H_4 Let H_4 be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol with functionality $\mathcal{F}_3$ and the 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 now the adversary is *not* given the inputs and outputs of honest and hacked parties anymore.

Define the ideal-model adversary Sim_4 to be like Sim_3 except for the following: For every *honest* party i, Sim_4 generates $3N$ random strings $s'_{ij}, r'_{ij}, k'_{ij}$, computes $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, s'_{ij}, r'_{ij}, k'_{ij})$ $(j = 1, \ldots, N)$, and iteratively reports $(i, \mathrm{Enc}(\mathsf{pk}_j, i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}))$ $(j \in \{1, \ldots, N\} \backslash \{i\})$ to $\mathcal{Z}$. If *all* parties are marked as `genuine`, then for every *corrupted* or *hacked* party i, Sim_4 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 $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$, where $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$. If one of the parties is marked as `fake`, then for every *corrupted* or *hacked* party i, Sim_4 sends $\perp$ to $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$.

Let $\mathrm{H}_{4,0}, \ldots, \mathrm{H}_{4,N}$ be the execution experiment between the environment $\mathcal{Z}$, the ideal protocol with functionality $\mathcal{F}_{3,0}, \ldots, \mathcal{F}_{3,N}$ and the adversary $\mathsf{Sim}_{4,0}, \ldots, \mathsf{Sim}_{4,N}$, respectively, where $\mathcal{F}_{3,i}$ and $\mathsf{Sim}_{4,i}$ are defined as follows:

Define $\mathcal{F}_{3,i}$ be identical to $\mathcal{F}_2$ except now the adversary is not given the inputs and outputs of the parties $l \in \{1, \ldots, i\}$ if they are honest or hacked.

Define the ideal-model adversaries $\mathsf{Sim}_{4,i}$ to be like Sim_3 except for the following: For every *honest* party $l \in \{1, \ldots, i\}$, $\mathsf{Sim}_{4,i}$ generates $3N$ random strings $s'_{lj}, r'_{lj}, k'_{lj}$, computes $\sigma'_{lj} \leftarrow \mathrm{Sig}(\mathsf{sgk}_l, j, s'_{lj}, r'_{lj}, k'_{lj})$ $(j = 1, \ldots, N)$, and iteratively reports $(l, \mathrm{Enc}(\mathsf{pk}_j, l, s'_{lj}, r'_{lj}, k'_{lj}, \sigma'_{lj}))$ $(j \in \{1, \ldots, N\} \setminus \{l\})$ to $\mathcal{Z}$. If *all* parties are marked as `genuine`, then for every *corrupted* or *hacked* party $l \in \{1, \ldots, i\}$, Sim_4 generates a random string $\tilde{y}_l \leftarrow \{0,1\}^{p_i(n)}$ and sends $(\tilde{y}_l, \mathrm{Mac}(k_l, \tilde{y}_l))$ to $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$, where $k_l \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$. If one of the parties is marked as `fake`, then for every *corrupted* or *hacked* party, $\mathsf{Sim}_{4,i}$ sends $\perp$ to $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$.

It holds that

$$\mathsf{Pr}[\mathrm{out}_{4,0}(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_3(\mathcal{Z}) = 1]$$

and

$$\mathsf{Pr}[\mathrm{out}_{4,N}(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_4(\mathcal{Z}) = 1]$$

Assume that there exists a non-negligible function ϵ such that $|\mathsf{Pr}[\mathrm{out}_3(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_4(\mathcal{Z}) = 1]| > \epsilon$. Then there exists an $i^* \in \{1, \ldots, N\}$ such that

$$|\mathsf{Pr}[\mathrm{out}_{4,i^*-1}(\mathcal{Z}) = 1] = \mathsf{Pr}[\mathrm{out}_{4,i^*}(\mathcal{Z}) = 1]| > \epsilon/N$$

One can now construct an adversary $\mathcal{A}$ against the IND-pCCA-security of PKE. The reduction is almost identical to the one in hybrid H_2.

Hence, there exist a negligible function negl_3 such that

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

Since H_4 is identical to the ideal-model experiment with functionality $[\mathcal{G}]$ and the simulator as defined in Definition 6, it follows that there exists a negligible function negl such that

$$|\mathsf{Pr}[\mathrm{Exec}_{\#\#}\left(\rho^{F_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}}, \mathcal{D}, \mathcal{Z}\right) = 1] - \mathsf{Pr}[\mathrm{Exec}_{\#\#}\left([\mathcal{G}], \mathsf{Sim}, \mathcal{Z}\right) = 1]| \leq \mathsf{negl}(n)$$

The statement follows. □

Remark 1. Note that one can also let a party check each message it receives (in its buffer) right away once it is online without having to wait for at least $N-1$ messages in the buffer. The protocol remains secure if one assumes the stronger assumption that PKE is IND-CCA-secure.

Remark 2. Using Theorems 1 and 2 (and Proposition 2 for transitivity), we can replace $\mathcal{F}_{\mathcal{G}}$ in our protocol with an appropriate adaptively UC-secure protocol, e.g. [CLOS02] (using a CRS as an additional setup). Note that the imported UC-secure protocol needs to be initially fortified (cf. Definition 4) since we require initially disconnected air-gap switches to $\mathcal{F}_{\mathcal{G}}$ in our construction.

Remark 3. Note that we do not model how to reuse machines such as the *registration machines* that stay disconnected throughout the protocol execution. In practice, one may assume, e.g., a reset button for these machines.

4.1 Up to N Corruptions/Hacks

We will now augment Construction 2 in order to obtain a protocol that is also secure if the adversary hacks *all* parties at the expense of one additional unhackabe hardware primitive called *decryption unit* (Dec-unit). In the new construction, parties do not decrypt the encrypted shares themselves but send the messages they received to the Dec-unit (cf. Fig. 3 in Appendix A).

More specifically, define the protocol $\rho_2^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}}$ to be identical to $\rho^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}}$ except that now each party additionally has an *unhackable* Dec-unit. Furthermore, each party has an `air-gap switch` to its Dec-unit and only one connection to the Enc-unit, namely a `data diode`. The Dec-unit gets the secret key from the corresponding Enc-unit in the sharing phase. In the compute phase, each party sends all the messages in its buffer to its Dec-unit for decryption. The Dec-unit only decrypts the first vector of ciphertexts it receives. Since the Dec-units 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 hacked. $\mathcal{Z}$ is unable to check if the tuples it receives were encrypted before since it does not have the secret keys. (cf. Appendix G for a more detailed description of the simulator).

The security proof is very similar to the proof of Theorem 3 and therefore omitted due to length restrictions.

Theorem 4 (Up to N Corruptions/Hacks, Non-Reactive Functionalities). *Let $\mathcal{G}$ be a* non-reactive *functionality.*
Let PKE, SIG, MAC *be as in Theorem 3.*

Then it holds that $\rho_2^{F_{\mathcal{G}},\mathcal{F}_{\mathsf{reg}},\mathcal{F}_{\mathsf{krk}}} \underset{\#\#}{\geq} [\mathcal{G}]$ for up to N corruptions/hacks.

5 Construction for Reactive Functionalities

In this section, we will construct a general MPC protocol for every fortified functionality of a *reactive* functionality that is secure in our framework. The new construction is a direct generalization of Construction 2.

For reactive functionalities, a new problem arises because a protocol party is 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 hacked. We therefore need to find a way to insert the input(s) of round $u \geq 2$ into the protocol without allowing a party to learn (or change) them.

To this end, we introduce an additional unhackable hardware module called *input interface machine* (IIM) that acts as the counterpart of the OIM for inputs. Let $\mathrm{R} \in \mathbb{N}$ be the number of rounds. In the offline sharing phase, each party i generates $2\mathrm{R}$ random pads $r_i^1, \ldots, r_i^{\mathrm{R}}, t_i^1, \ldots, t_i^{\mathrm{R}}$ and shares them as before. Also, each party pads its (first) input $\tilde{x}_i^1 = x_i^1 + t_i^1$ and computes a MAC tag of it. Then, each party sends the R random pads $r_i^1, \ldots, r_i^{\mathrm{R}}$ as well as the MAC key k_i to the OIM and the other R random pads $t_i^1, \ldots, t_i^{\mathrm{R}}$ and the MAC key k_i

to the IIM. As before, each random pad is shared with the other parties along with signatures on these shares, the PID of the designated receiver as well as the *number of the round* in which this share is to be used. Note that the latter prevents an adversary from using shares from earlier rounds.

In each online compute phase, the parties will then use their shares and their padded inputs in order to compute the desired padded output values and a MAC tag of these padded output values *along with a prefix indicating this being an output and the round number.* Verification and reconstruction of the output values is then done as before using the OIM. Note that since the prefix contains the round number, the OIM is able to reject results from *earlier* computation phases.

As before, each input to the compute phase has to be verified before the actual multi-party computation. Now, however, not only the signatures of the shares are verified but also the MAC tags of the padded inputs. In order to obtain the MAC tags for the padded inputs of round $u \geq 2$, the respective input needs to be inserted into the protocol via the IIM. The IIM pads each input it receives and computes a MAC tag of the padded input *along with a prefix indicating this being an input and the round number.* It then sends the resulting tuple to the party. This way, a party will be able to continue the computation without learning the inputs of round $u \geq 2$. Note that due to the prefix containing the round number, the adversary cannot use padded inputs of *earlier* rounds. (Also note that since the prefix indicates inputs/outputs, an adversary cannot send a padded *input* to the OIM.)

As before, we will take a modular approach and define an ideal functionality $F_{\mathcal{G}}^{\mathtt{reac}}$ that implements the verification of the input values in the compute phase as well as the multi-party computation on the shares and padded inputs.

We first define the functionality $F_{\mathcal{G}}^{\mathtt{reac}}$.

Construction 3
Let $\mathcal{G}$ be a (possibly reactive) ideal functionality.
$F_{\mathcal{G}}^{\mathtt{reac}}$ proceeds as follows, running with parties $P_1, \dots, P_N$ and an adversary $\mathcal{A}$ and parametrized with a digital signature SIG *and a message authentication code* MAC.

1. *If this is round $u = 1$, do the following:*
   - *Upon receiving input $\overline{\mathsf{vk}}_i = (\mathsf{vk}_1^{(i)}, \dots, \mathsf{vk}_N^{(i)})$, $(t_{ji}^1, r_{ji}^1, \sigma_{ji}^1, k_{ji}, \sigma'_{ji})$ $(j = 1, \dots, N)$ and $(\tilde{x}_i^1, \tau_i^1)$ from party P_i, store that input and send* $(\mathtt{received}, P_i)$ *to $\mathcal{A}$.*
   - *Once each party has sent its input, check if one party has sent $\perp$. If yes, output $\perp$*
   - *Else, check if $\overline{\mathsf{vk}}_1 = \cdots = \overline{\mathsf{vk}}_N$. If no, output $\perp$*
   - *Else, set $(\mathsf{vk}_1, \dots, \mathsf{vk}_n) = (\mathsf{vk}_1^{(1)}, \dots, \mathsf{vk}_N^{(1)})$. For all $i = 1, \dots, N$, check if* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, k_{ji}, \sigma'_{ji}) = 1$ *for all $j = 1, \dots, N$,. If no, output $\perp$*
   - *Else, for all $i = 1, \dots, N$, compute and store $k_i = k_{i1} + k_{i2} + \cdots + k_{iN}$.*
   - *Continue with Step 3*
2. *Else, if this is round $u > 1$, do the following:*
   - *Upon receiving input $(t_{ji}^u, r_{ji}^u, \sigma_{ji}^u)$ $(j = 1, \dots, N)$ and $(\tilde{x}_i^u, \tau_i^u)$, store that input and send* $(\mathtt{received}, P_i)$ *to $\mathcal{A}$.*

- *Once each party has sent its input, continue with Step 3*

3. *For all $i = 1, \dots, N$, check if* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, u, i, t^u_{ji}, r^u_{ji}, \sigma^u_{ji}) = 1$ *for all* $j = 1, \dots, N$ *and if* $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, \texttt{Inp Round u}, \tilde{x}^u_i, \tau^u_i) = 1$. *If this does not hold, output* $\perp$
4. *Else, for each* $i = 1, \dots, N$, *compute* $r^u_i = r^u_{i1} + r^u_{i2} + \cdots + r^u_{iN}$ *and* $t^u_i = t^u_{i1} + t^u_{i2} + \cdots + t^u_{iN}$ *and* $x^u_i = \tilde{x}^u_i + t^u_i$.
5. *Internally run* $\mathcal{G}$ *on input* $(x^u_i, \dots, x^u_N)$. *Let* $(y^u_1, \dots, y^u_N)$ *be the output of* $\mathcal{G}$.
6. *For all* $i = 1, \dots, N$, *compute* $o^u_i = y^u_i + r^u_i$ *and* $\theta^u_i \leftarrow \mathrm{Mac}(k_i, \texttt{Outp Round u}, y^u_i + r^u_i)$.
7. *For all* $i = 1, \dots, N$, *send a public delayed output* (o^u_i, θ^u_i) *to* P_i.

C *If* $\mathcal{A}$ *sends* $(\texttt{corrupt}, P)$ *and* $\mathcal{F}_{\mathcal{G}}$ *has already received an input from party* P *then* $\mathcal{F}_{\mathcal{G}}$ *sends that input to* $\mathcal{A}$. *Otherwise* $\mathcal{F}_{\mathcal{G}}$ *sends "no input yet". Furthermore,* $\mathcal{F}_{\mathcal{G}}$ *lets* $\mathcal{A}$ *determine the input of party* P.

Next, we define our protocol for reactive functionalities, which is in the $(F^{reac}_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}})$-hybrid model.

Construction 4 *Define the protocol* $\rho_3^{F_{\mathcal{G}}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}}$ *as follows:*
Architecture: *(cf. Fig. 3 in Appendix A) Each party has two hackable and three unhackable sub-parties. The hackable sub-parties are a* buffer *and a* registration machine*, and the unhackable sub-parties are an* Enc*-unit, an* OIM *and an* IIM*. Each party has an* `air-gap switch` *to its* buffer*, a* `data diode` *to its* OIM*, an* `air-gap switch` *and a* `data diode` *to its* Enc*-unit, an* `air-gap switch` *to* $\mathcal{F}_{\mathsf{reg}}$*, an* `air-gap switch` *to* $F^{\mathsf{reac}}_{\mathcal{G}}$*, a* `standard`*-connection to its* IIM *and an* `air-gap switch` *to the network. Furthermore, each* Enc*-unit has a* `standard`*-connection to* $\mathcal{F}_{\mathsf{krk}}$ *and a* `standard`*-connection to the network, each* buffer *has a* `standard`*-connection to the network, each* IIM *has an* `air-gap switch` *at its input port and each* registration machine *has an* `air-gap switch` *to its main party and a* `data diode` *to* $\mathcal{F}_{\mathsf{reg}}$*. Apart from the parties' input ports and the* registration machines' *connection to their main parties, all air-gap switches are* disconnected *at the beginning.*

- Offline Sharing Phase:
  Upon input x^1_i, *each party* P_i *does the following:*
  - Disconnect *at the input port.*
  - *Generate random pads* $t^1_i, t^2_i, \dots, t^{\mathrm{R}}_i \leftarrow \{0,1\}^n$ *and* $r^1_i, r^2_i, \dots, r^{\mathrm{R}}_i \leftarrow \{0,1\}^{p_i(n)}$
  - *Generate shares* $t^u_{i1} + t^u_{i2} + \cdots + t^u_{iN} = t^u_i$ $(u = 1, \dots, \mathrm{R})$ *and* $r^u_{i1} + r^u_{i2} + \cdots + r^u_{iN} = r^u_i$ $(u = 1, \dots, \mathrm{R})$.
  - *Generate* $k_i \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ *and* $(\mathsf{sgk}_i, \mathsf{vk}_i) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$
  - *Send* (k_i, r^u_i) $(u = 1, \dots, \mathrm{R})$ *to the* OIM *and* (k_i, t^u_i) $(u = 1, \dots, \mathrm{R})$ *to the* IIM.
  - *Send the verification key* vk_i *to the* registration machine*. The* registration machine *will then* disconnect *itself from its main party and relay* vk_i *to* $\mathcal{F}_{\mathsf{reg}}$ *(using its own PID).*
  - *Create signatures* $\sigma^u_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, u, j, t^u_{ij}, r^u_{ij})$ *and* $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, k_{ij})$ $(j = 1, \dots, N; u = 1, \dots, \mathrm{R})$.

- *Compute* $\tilde{x}_i^1 = x_i^1 + t_i^1$ *and* $\tau_i^1 \leftarrow \mathrm{Mac}(k_i, \texttt{Inp Round 1}, \tilde{x}_i^1)$
- *Let* $\bar{t}_{ij} = (t_{ij}^1, t_{ij}^2, \ldots, t_{ij}^{\mathrm{R}})$, $\overline{r}_{ij} = (r_{ij}^1, r_{ij}^2, \ldots, r_{ij}^{\mathrm{R}})$ *and* $\overline{\sigma}_{ij} = (\sigma_{ij}^1, \sigma_{ij}^2, \ldots, \sigma_{ij}^{\mathrm{R}})$. *Iteratively send* $(j, \bar{t}_{ij}, \overline{r}_{ij}, \overline{\sigma}_{ij}, k_{ij}, \sigma'_{ij})$ $(j \in \{1, \ldots, \mathrm{R}\} \setminus \{i\})$ *to the* Enc-*unit (at each activation)*
- *At first activation, the* Enc-*unit requests a key pair* $(\mathsf{pk}_i, \mathsf{sk}_i)$ *from* $\mathcal{F}_{\mathsf{krk}}$.
- *Upon receiving a tuple* $(j, \bar{t}_{ij}, \overline{r}_{ij}, \overline{\sigma}_{ij}, k_{ij}, \sigma'_{ij})$ $(j \in \{1, \ldots, N\} \setminus \{i\})$, *the* Enc-*unit requests the public key* pk_j *belonging to party* P_j *from* $\mathcal{F}_{\mathsf{krk}}$. *If* pk_j *does not exist yet, the* Enc-*unit sends a "request public key message" to the* Enc-*unit of* P_j. *Otherwise, it computes* $c_j^i \leftarrow \mathrm{Enc}(\mathsf{pk}_j, i, \bar{t}_{ij}, \overline{r}_{ij}, \overline{\sigma}_{ij}, k_{ij}, \sigma'_{ij})$ *and sends* (i, c_j^i) *to party* P_j.
- *Once all shares have been sent to the* Enc-*unit,* erase *everything except for the tuple* $(\bar{t}_{ii}, \overline{r}_{ii}, \overline{\sigma}_{ii}, k_{ii}, \sigma'_{ii})$ *and* $(\tilde{x}_i^1, \tau_i^1)$ *and the verification key* vk_i *(in particular, the input* x_i, *signing key* sk_i, *random pads* t_i^u, r_i^u *and MAC key* k_i *are erased).*

– First Online Compute Phase:
Once the last step in the offline sharing phase is completed, a party P_i *does the following:*

- Connect *to the* buffer, Enc-*unit and* $\mathcal{F}_{\mathsf{reg}}$.
- *Request the secret key* sk_i *from the* Enc-*unit.*
- *Request all verification keys* $\{\mathsf{vk}_l\}_{l \in \{1,\ldots,N\} \setminus \{i\}}$ *that were registered with the PIDs of the other parties'* registration machines *from* $\mathcal{F}_{\mathsf{reg}}$. *If not all verification keys can be retrieved yet, go into idle mode and request again at the next activation.*
- *At each activation, check if there are at least* $N-1$ *messages in its buffer. If no, go into idle mode and when activated again check again.*
  If yes, check if one has received from each party j *a set* $\mathcal{M}_j = \{(j, \tilde{c})\}$ *with the following property:*
  There exists a tuple $(\hat{\bar{t}}_{ji}, \hat{\overline{r}}_{ji}, \hat{\overline{\sigma}}_{ji}, \hat{k}_{ji}, \hat{\sigma}'_{ji})$, *where* $\hat{\bar{t}}_{ji} = (\hat{t}_{ji}^1, \hat{t}_{ji}^2, \ldots, \hat{t}_{ji}^{\mathrm{R}})$, $\hat{\overline{r}}_{ji} = (\hat{r}_{ji}^1, \hat{r}_{ji}^2, \ldots, \hat{r}_{ji}^{\mathrm{R}})$ *and* $\hat{\overline{\sigma}}_{ji} = (\hat{\sigma^1}_{ji}, \hat{\sigma}_{ji}^2, \ldots, \hat{\sigma}_{ji}^{\mathrm{R}})$, *and an element* $(j, c) \in \mathcal{M}_j$ *such that*
  - $\mathrm{Dec}(\mathsf{sk}_i, c) = (j, \hat{\bar{t}}_{ji}, \hat{\overline{r}}_{ji}, \hat{\overline{\sigma}}_{ji}, \hat{k}_{ji}, \hat{\sigma}'_{ji})$
  - $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, u, i, \hat{t}_{ji}^u, \hat{r}_{ji}^u, \hat{\sigma}_{ji}^u) = 1$ $(u = 1, \ldots, \mathrm{R})$ *and* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, \hat{k}_{ji}, \hat{\sigma}'_{ji}) = 1$
  - *For every* $(j, \tilde{c}) \in \mathcal{M}_j$ *it holds that either* $\mathrm{Dec}(\mathsf{sk}_i, \tilde{c}) = (j, \hat{\bar{t}}_{ji}, \hat{\overline{r}}_{ji}, \hat{\overline{\sigma}}_{ji}, \hat{k}_{ji}, \hat{\sigma}'_{ji})$ *or* $(j, \tilde{c})$ *is "invalid", i.e., either decrypts to* $(j, \tilde{\bar{t}}_{ji}, \tilde{\overline{r}}_{ji}, \tilde{\overline{\sigma}}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}'_{ji})$ *such that either* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, u, i, \tilde{t}_{ji}^u, \tilde{r}_{ji}^u, \tilde{\sigma}_{ji}^u) = 0$ *for some* u *or* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}_j, i, \tilde{k}_{ji}, \tilde{\sigma}'_{ji}) = 0$, *or decrypts to* $(l, \tilde{\bar{t}}_{ji}, \tilde{\overline{r}}_{ji}, \tilde{\overline{\sigma}}_{ji}, \tilde{k}_{ji}, \tilde{\sigma}'_{ji})$ *where* $l \neq j$, *or* $\tilde{c}$ *does not decrypt correctly.*

  If this does not hold, send $\perp$ *to* $\mathcal{F}_{\mathcal{G}}$.
  Else, send all verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ *as well as all* $(\hat{t}_{ji}^1, \hat{r}_{ji}^1, \hat{\sigma}_{ji}^1, \hat{k}_{ji}, \hat{\sigma}'_{ji})$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ *and the own shares* $(t_{ii}^1, r_{ii}^1, \sigma_{ii}^1, k_{ii}, \sigma'_{ii})$ *and* $(\tilde{x}_i^1, \tau_i^1)$ *to* $F_{\mathcal{G}}^{\texttt{reac}}$.

  - *Instruct the* IIM *to* connect *its input port.*

- Subsequent Online Compute Phases:
  Upon receiving an input x_i^u in round u, each IIM *does the following:*
  - *Compute* $\tilde{x}_i^u = x_i^u + t_i^u$ *and* $\tau_i^u \leftarrow \mathrm{Mac}(k_i, \texttt{Inp Round u}, \tilde{x}_i^u)$ *and send* $(\tilde{x}_i^u, \tau_i^u)$ *to the party* P_i.
  - *Party* P_i *then sends* $(\hat{t}_{ji}^u, \hat{r}_{ji}^u, \hat{\sigma}_{ji}^u)$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ *and the own share* $(t_{ii}^u, r_{ii}^u, \sigma_{ii}^u)$ *and* $(\tilde{x}_i^u, \tau_i^u)$ *to* $F_{\mathcal{G}}^{\texttt{reac}}$.

- Online Output Phases:
  Upon receiving an output from $F_{\mathcal{G}}^{\texttt{reac}}$ *in round u, a party P_i does the following:*
  - *If this output equals* $\perp$*, it sends* $\perp$ *to the* OIM*, which then outputs* $\perp$
  - *Otherwise, let* (o_i^u, θ_i^u) *be the output from* $F_{\mathcal{G}}^{\texttt{reac}}$*. Send this tuple to the* OIM.
  - *The* OIM *then checks if* $\mathrm{Vrfy}_{\mathrm{MAC}}(k_i, \texttt{Outp Round u}, o_i^u, \theta_i^u) = 1$ *and outputs* $y_i^u = o_i^u + r_i^u$ *if this holds, and* $\perp$ *otherwise.*

We are now ready to state our theorem for reactive functionalities. The proof is similar to the proof of Theorem 3 and therefore omitted due to length restrictions.

Theorem 5 (Up to $N-1$ Corruptions/Hacks, Reactive Functionalities).
Let $\mathcal{G}$ be a (possibly reactive) functionality.
Let $\mathrm{PKE} = (\mathrm{Gen}_{\mathrm{PKE}}, \mathrm{Enc}, \mathrm{Dec})$ *be a IND-pCCA-secure PKE,* $\mathrm{SIG} = (\mathrm{Gen}_{\mathrm{SIG}}, \mathrm{Sig}, \mathrm{Vrfy}_{\mathrm{SIG}})$ *an EUF-naCMA-secure and length-normal DigSig and* $\mathrm{MAC} = (\mathrm{Gen}_{\mathrm{MAC}}, \mathrm{Mac}, \mathrm{Vrfy}_{\mathrm{MAC}})$ *an EUF-CMA-secure MAC.*

Then it holds that $\rho_3^{F_{\mathcal{G}}^{reac}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}} \underset{\#\#}{\geq} [\mathcal{G}]$ *for up to $N-1$ corruptions/hacks.*

5.1 Up to N Corruptions/Hacks

As in Section 4.1, we can augment Construction 4 in order to obtain a protocol $\rho_4^{F_{\mathcal{G}}^{reac}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}}$ that is also secure if the adversary hacks *all* parties at the expense of the additional unhackabe hardware module Dec-unit (cf. Fig. 4 in Appendix A). We again omit the proof due to length restrictions.

Theorem 6 (Up to N Corruptions/Hacks, Reactive Functionalities).
Let $\mathcal{G}$ be a (possibly reactive) functionality.
Let PKE, SIG, MAC *be as in Theorem 5.*

Then it holds that $\rho_4^{F_{\mathcal{G}}^{reac}, \mathcal{F}_{\mathsf{reg}}, \mathcal{F}_{\mathsf{krk}}} \underset{\#\#}{\geq} [\mathcal{G}]$ *for up to N corruptions/hacks.*

6 Weakening the Assumption on Erasure

We can also obtain the results in Theorems 3 to 6 with only a very weak notion of erasure. To this end, we split each main party into two *hackable* parts S and T that are connected via a `data diode`. At the beginning, S takes the (first) input and carries out the offline sharing phase. Once the sharing phase is over, S sends its own shares (and for reactive functionalities also its padded input) together with their signatures (and possibly MAC tags) to T. From then on, T carries out all further computations. S is never activated again and remains offline throughout the protocol execution. After the protocol execution, S has to be "destroyed" or at least reset to its initial state in order to erase its secret inputs. This assumption is weaker than the selective erasure we require in Theorems 3 to 6. Moreover, it is in line with what is implicitly assumed in large parts of the MPC literature, e.g. in the UC framework, where the Turing machines holding secrets cease to exist after protocol executions.

7 Conclusion

We have proposed a new framework that captures the advantages provided by unhackable hardware modules and isolation. Using few simple unhackable hardware modules, we constructed protocols for securily realizing any fortified functionality in our framework.

References

[AMR14] D. Achenbach, J. Müller-Quade, and J. Rill. "Universally Composable Firewall Architectures Using Trusted Hardware". In: *Balkan-CryptSec 2014*. LNCS 9024. Springer, 2014, pp. 57–74.

[BDH+17] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. "Concurrently Composable Security with Shielded Super-Polynomial Simulators". In: *EUROCRYPT 2017*. LNCS 10210. 2017, pp. 351–381.

[BDLO14] J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky. "How to withstand mobile virus attacks, revisited". In: *PODC 2014*. ACM, 2014, pp. 293–302.

[Can01] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *FOCS 2001*. IEEE. 2001, pp. 136–145.

[CFGN96] R. Canetti, U. Feige, O. Goldreich, and M. Naor. "Adaptively Secure Multi-Party Computation". In: *STOC 1996*. 1996, pp. 639–648.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. "Universally composable two-party and multi-party secure computation". In: *STOC 2002*. ACM, 2002, pp. 494–503.

[CPV17] R. Canetti, O. Poburinnaya, and M. Venkitasubramaniam. "Equivocating Yao: constant-round adaptively secure multiparty computation in the plain model". In: *STOC 2017*. ACM, 2017, pp. 497–509.

[DMMN13] N. Döttling, T. Mie, J. Müller-Quade, and T. Nilges. "Implementing Resettable UC-Functionalities with Untrusted Tamper-Proof Hardware-Tokens". In: *TCC 2013*. LNCS 7785. Springer, 2013, pp. 642–661.

[GIK$^+$15] S. Garg, Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. "Cryptography with One-Way Communication". In: *CRYPTO 2015*. LNCS 9216. Springer, 2015, pp. 191–208.

[GIS$^+$10] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. "Founding Cryptography on Tamper-Proof Hardware Tokens". In: *TCC 2010*. LNCS 5978. Springer, 2010, pp. 308–326.

[HLP15] C. Hazay, Y. Lindell, and A. Patra. "Adaptively Secure Computation with Partial Erasures". In: *PODC 2015*. ACM, 2015, pp. 291–300.

[HPV17] C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam. "Constant Round Adaptively Secure Protocols in the Tamper-Proof Hardware Model". In: *PKC 2017*. LNCS 10175. Springer, 2017, pp. 428–460.

[IPS08] Y. Ishai, M. Prabhakaran, and A. Sahai. "Founding Cryptography on Oblivious Transfer - Efficiently". In: *CRYPTO 2008*. LNCS 5157. Springer, 2008, pp. 572–591.

[Kat07] J. Katz. "Universally Composable Multi-party Computation Using Tamper-Proof Hardware". In: *EUROCRYPT 2007*. LNCS. Springer, 2007, pp. 115–128.

[Nem17] H. Nemati. "Secure System Virtualization: End-to-End Verification of Memory Isolation". PhD thesis. Royal Institute of Technology, Stockholm, Sweden, 2017. URL: http://nbn-resolving.de/urn:nbn:se:kth:diva-213030.

[OY91] R. Ostrovsky and M. Yung. "How to Withstand Mobile Virus Attacks (Extended Abstract)". In: *PODC 1991*. ACM, 1991, pp. 51–59.

[ZGL18] E. Zheng, P. Gates-Idem, and M. Lavin. "Building a virtually air-gapped secure environment in AWS: with principles of devops security program and secure software delivery". In: *Hot Topics in the Science of Security, HoTSoS 2018*. ACM, 2018, 11:1–11:8.

Appendix

A Graphical Depicture of Architectures

This section contains graphical depictions of the architectures of the protocols in Sections 4 and 5. Main parties are represented by circles, sub-parties and ideal functionalities by boxes. Boxes with bold lines denote that the sub-party is *unhackable*. Standard channels are denoted by lines, data diodes and air-gap switches by their usual symbols. Dashed lines denote standard connections to other parties that are currently not shown. Downward connections from the main party and possibly from the OIMs or IIMs are to the environment (or the calling protocol).

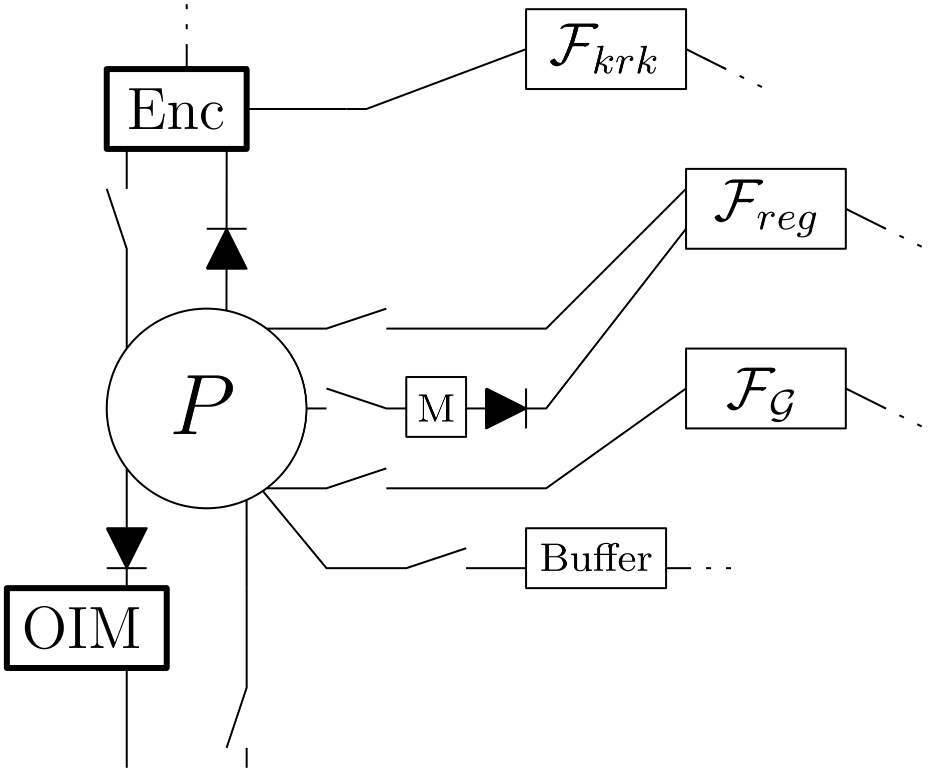

Fig. 1. Architecture for non-reactive functionalities and up to $N-1$ corruption / hacks.

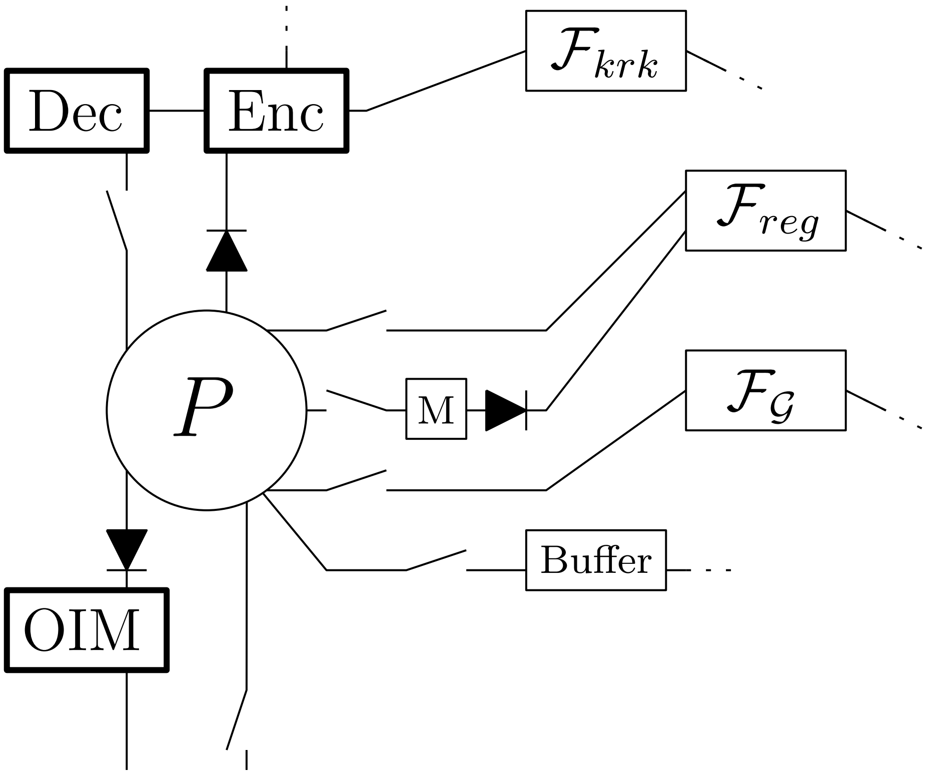

Fig. 2. Architecture for non-reactive functionalities and up to N corruption / hacks.

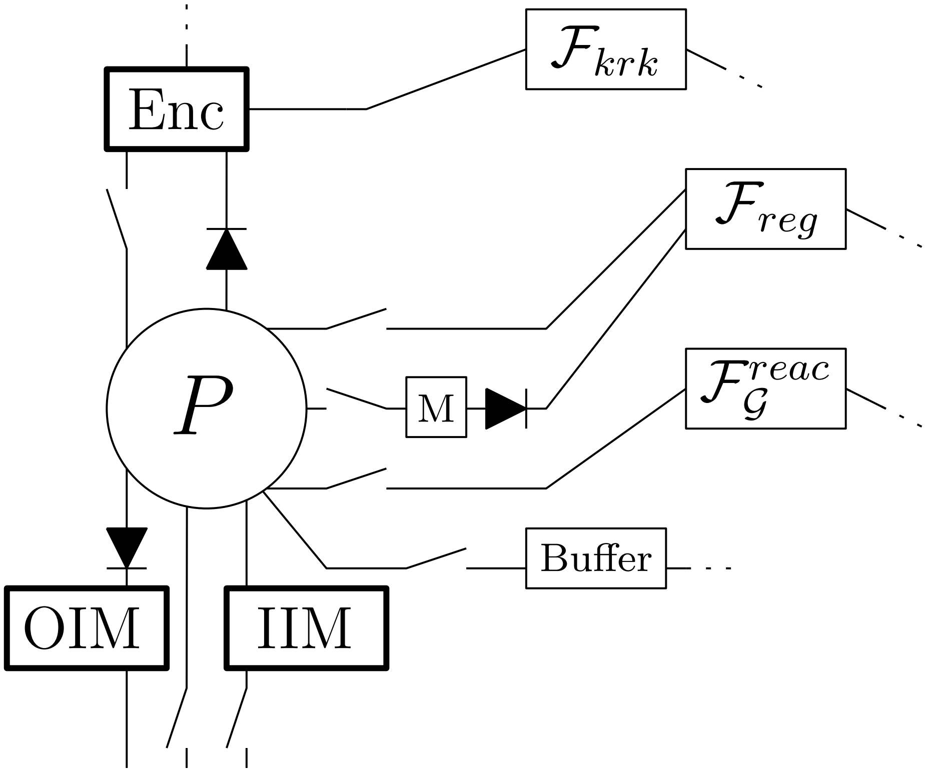

Fig. 3. Architecture for reactive functionalities and up to $N - 1$ corruption / hacks.

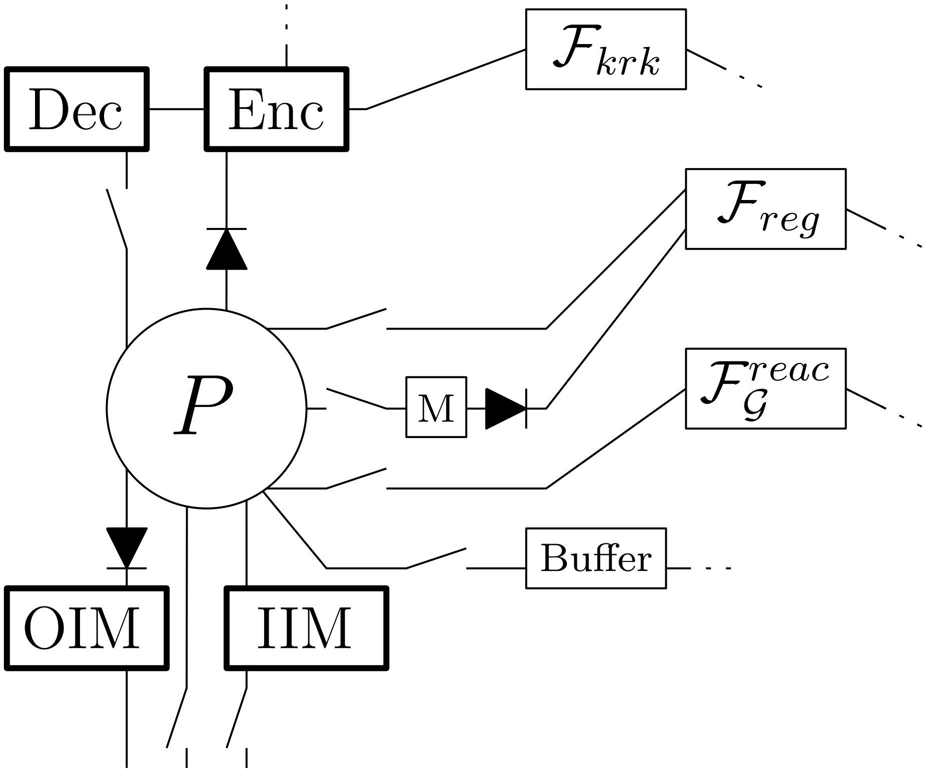

Fig. 4. Architecture for reactive functionalities and up to N corruption / hacks.

B Definitions

B.1 Ideal Functionalities

In our constructions (Sections 4 and 5), we make use of several ideal functionalities which we introduce in the following.

Definition 7 (Ideal Functionality $\mathcal{F}_{\mathsf{SFE}}^{f}$). *$\mathcal{F}_{\mathsf{SFE}}$ proceeds as follows, given a list of functions $f = f_1, \ldots, f_R$, $f_i : (\{0,1\}^* \cup \{\bot\})^N \times U \times S \to (\{0,1\}^*)^N$ $(i = 1, \ldots, R)$. At the first activation, verify that $sid = (\mathcal{P}, sid')$ where $\mathcal{P}$ is an ordered set of N identities; else halt. Denote the identities $P_1, \ldots, P_N$. Also, initialize variables $x_1^i, \ldots, x_N^i, y_1^i, \ldots, y_N^i$ $(i = 1, \ldots, R)$, state to a default value $\bot$. Set $c_j = 0$ $(j = 1, \ldots, N)$, $c = 0$. Next:*

1. *Upon receiving input $(\mathtt{Input}, sid, v)$ from some party $P_i \in \mathcal{P}$, set $x_i^{c_i} = v$ and send a message $(\mathtt{Input}, sid, P_i)$ to the adversary. Increment c_i.*
2. *Upon receiving input $(\mathtt{Output}, sid, v)$ from some party $P_i \in \mathcal{P}$, do:*

(a) *If x_i^c has been set for all parties P_i that are currently uncorrupted, and $y_1^c, \dots, y_n^c$ have not been yet set, then choose $r^c \leftarrow U$ and set $(y_1^c, \dots, y_n^c, state') = f_c(x_1^c, \dots, x_n^c, r^c, state)$. Let the adversary determine the output of corrupted parties.*

(b) *Generate a private delayed output y_i^c to P_i.*

(c) *If all $P_i \in \mathcal{P}$ have received output for round c, increment c and set $state = state'$.*

Definition 8 (Ideal Functionality $\mathcal{F}_{\mathsf{reg}}$).
$\mathcal{F}_{\mathsf{reg}}$ proceeds as follows:

- *Report: Upon receiving a message* $(\texttt{register}, \mathrm{vk})$ *from party P, send* $(\texttt{registered}, P, \mathrm{vk})$ *to the adversary; upon receiving ok from the adversary, record the pair* (P, vk). *Otherwise, ignore the message.*
- *Retrieve: Upon receiving a message* $(\texttt{retrieve}, P_i)$ *from some party P_j (or the adversary $\mathcal{S}$), generate a public delayed output* $(\texttt{retrieve}, P_i, \mathrm{vk})$ *to P_j, where $v = \perp$ if no record* (P, vk) *exists.*

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

Definition 9 (Ideal Functionality $\mathcal{F}_{\mathsf{krk}}$).
$\mathcal{F}_{\mathsf{krk}}$ proceeds as follows, given a (deterministic) key generation function $\mathrm{Gen}_{\mathrm{PKE}}$ *(with security parameter n), running with parties $P_1, \dots, P_N$ and an adversary $\mathcal{S}$:*

- *Registration: When receiving a message* $(\texttt{register}, sid)$ *from party P_i that has not previously registered, compute* $(\mathrm{pk}_i, \mathrm{sk}_i) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$ *and record the tuple* $(P_i, \mathrm{pk}_i, \mathrm{sk}_i)$.
- *Retrieval: When receiving a message* $(\texttt{retrieve}, sid, P_i)$ *from party P_j (where $j \neq i$), if there is a previously recorded tuple of the form* $(P_i, \mathrm{pk}_i, \mathrm{sk}_i)$, *then generate a public delayed output* (i, P_i, pk_i) *to P_j. Otherwise generate a public delayed output* $(i, P_i, \perp)$ *to P_j. When receiving a message* $(\texttt{retrieve}, i, P_i)$ *from party P_i, if there is a previously recorded tuple of the form* $(P_i, \mathrm{pk}_i, \mathrm{sk}_i)$, *then generate a private delayed output* $(i, P_i, \mathrm{pk}_i, \mathrm{sk}_i)$ *to P_i. Otherwise, generate a private delayed output* $(i, P_i, \perp)$ *to P_i.*

B.2 Cryptographic Primitives

In the following, we define the cryptographic primitives used in this paper along with their required security properties.

Public-Key Encryption Schemes

Definition 10 (Public-Key Encryption Scheme).
Let $\mathcal{M} \subseteq \{0,1\}^{p(n)}$ be the message space. A public-key encryption scheme $\mathrm{PKE} = (\mathrm{Gen}_{\mathrm{PKE}}, \mathrm{Enc}, \mathrm{Dec})$ *consists of three probabilistic polynomial-time algorithms such that:*

1. *The* key-generation algorithm $\mathrm{Gen}_{\mathrm{PKE}}$ *takes as input* 1^n *and outputs a tuple* $(\mathrm{pk}, \mathrm{sk})$*. We call* pk *the* public key *and* sk *the* private key *or* secret key*.*
2. *The* encryption algorithm Enc *takes as input a public key* pk *and a message* $m \in \mathcal{M}$ *and outputs a ciphertext* c*.*
3. *The* decryption algorithm Dec *takes as input a private key* sk *and a ciphertext* c *and outputs a message* $m \in \mathcal{M}$ *or a special symbol* $\perp$ *denoting failure.*

We call PKE perfectly correct *if* $\mathrm{Dec}(\mathrm{sk}, \mathrm{Enc}(\mathrm{pk}, m)) = m$ *for any* $m \in \mathcal{M}$ *and for all* $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$*.*

Definition 11 (Indistinguishability Under Parallel Chosen Ciphertext Attack). *We call a public-key encryption scheme* PKE *IND-pCCA-secure if for every* PPT*-adversary* $\mathcal{A}$ *and all* $z \in \{0,1\}^*$ *there exists a negligible function* negl *such that*

$$\left| \Pr[\mathsf{Exp}^{IND-pCCA}_{\mathcal{A}(z),\mathrm{PKE}}(n) = 1] - \frac{1}{2} \right| \leq \mathsf{negl}(n)$$

The experiment $\mathsf{Exp}^{IND-pCCA}_{\mathcal{A}(z),\mathrm{PKE}}(n)$ *is defined as follows: At the beginning, the experiment generates keys* $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{Gen}_{\mathrm{PKE}}(1^n)$*. On input* 1^n*,* z *and* pk*, the adversary* $\mathcal{A}$ *chooses two messages* m_0^*, m_1^* *of equal length and sends them to the experiment. The experiment then chooses a bit* b *uniformly random from* $\{0,1\}$ *and encrypts* $c^* \leftarrow \mathrm{Enc}(\mathrm{pk}, m_b)$*. On input* 1^n*,* z*,* c^* *and* pk*, the adversary may now choose an arbitrary number of ciphertexts (not containing* c^**) and do a single query to an oracle* $\mathcal{O}_{\mathrm{Dec}(\mathrm{sk},\cdot)}$ *sending these ciphertexts to decrypt them all in parallel. Afterwards,* $\mathcal{A}$ *chooses a bit* $b' \in \{0,1\}$*. If* $b = b'$*, the experiment outputs* 1*, otherwise* 0*.*

Message Authentication Codes

Definition 12 (Message Authentication Code). *A* message authentication code $\mathrm{MAC} = (\mathrm{Gen}_{\mathrm{MAC}}, \mathrm{Mac}, \mathrm{Vrfy}_{\mathrm{MAC}})$ *consists of three probabilistic polynomial-time algorithms such that:*

1. *The* key-generation algorithm $\mathrm{Gen}_{\mathrm{MAC}}$ *takes as input* 1^n *and outputs a key* k*. We call* k *the* MAC key*.*
2. *The* tag-generation algorithm Mac *takes as input a MAC key* k *and a message* m *and outputs a* MAC tag t*.*
3. *The* verification algorithm $\mathrm{Vrfy}_{\mathrm{MAC}}$ *takes as input a MAC key* k*, a message* m *and a presumptive MAC tag* t *and outputs a bit* $b \in \{0,1\}$*, with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid*.*

It is required that for every MAC key $k \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$ *and every* $m \in \{0,1\}^*$*, it holds that* $\mathrm{Vrfy}_{\mathrm{MAC}}(k, m, \mathrm{Mac}(k, m)) = 1$ *(correctness).*

Definition 13 (Existential Unforgeability under One Chosen Message Attack for MACs). *We call a message authentication code* MAC *EUF-1-CMA-secure if for every* PPT*-adversary* $\mathcal{A}$ *and all* $z \in \{0,1\}^*$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Exp}^{EUF-1-CMA}_{\mathcal{A}(z),\mathrm{MAC}}(n) = 1] \leq \mathsf{negl}(n)$$

The experiment $\mathsf{Exp}^{EUF-1-CMA}_{\mathcal{A}(z),\mathrm{MAC}}(n)$ *is defined as follows: At the beginning, the experiment generates a key* $k \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$*. On input* 1^n*, the adversary* $\mathcal{A}$ *may send a single query* m' *to an oracle* $\mathcal{O}_{\mathrm{Mac}(k,\cdot)}$*. Afterwards,* $\mathcal{A}$ *outputs a tuple* (m^*, t^*)*. If* $\mathrm{Vrfy}_{\mathrm{MAC}}(k, m^*, t^*) = 1$ *and* $m^* \neq m'$*, the experiment outputs 1, else 0.*

Definition 14 (Existential Unforgeability under Chosen Message Attack for MACs). *We call a message authentication code* MAC *EUF-CMA-secure if for every* PPT*-adversary* $\mathcal{A}$ *and all* $z \in \{0,1\}^*$ *there exists a negligible function* negl *such that*

$$\mathsf{Pr}[\mathsf{Exp}^{EUF-CMA}_{\mathcal{A}(z),\mathrm{MAC}}(n) = 1] \leq \mathsf{negl}(n)$$

The experiment $\mathsf{Exp}^{EUF-CMA}_{\mathcal{A}(z),\mathrm{MAC}}(n)$ *is defined as follows: At the beginning, the experiment generates a key* $k \leftarrow \mathrm{Gen}_{\mathrm{MAC}}(1^n)$*. On input* 1^n*, the adversary* $\mathcal{A}$ *may send queries to an oracle* $\mathcal{O}_{\mathrm{Mac}(k,\cdot)}$*. Let* $\mathcal{Q}$ *be the set of all queries. Eventually,* $\mathcal{A}$ *outputs a tuple* (m^*, t^*)*. If* $\mathrm{Vrfy}_{\mathrm{MAC}}(k, m^*, t^*) = 1$ *and* $m^* \notin \mathcal{Q}$*, the experiment outputs 1, else 0.*

Signature Schemes

Definition 15 (Signature Scheme). *A* signature scheme $\mathrm{SIG} = (\mathrm{Gen}_{\mathrm{SIG}}, \mathrm{Sig}, \mathrm{Vrfy}_{\mathrm{SIG}})$ *consists of three probabilistic polynomial-time algorithms such that:*

1. *The* key-generation algorithm $\mathrm{Gen}_{\mathrm{SIG}}$ *takes as input* 1^n *and outputs a tuple* $(\mathsf{vk}, \mathsf{sgk})$*. We call* vk *the (public)* verification key *and* sgk *the (private)* signing key *or* signature key*.*
2. *The* signature-generation algorithm Sig *takes as input a signing key* sgk *and a message* m *and outputs a* signature σ*.*
3. *The* verification algorithm $\mathrm{Vrfy}_{\mathrm{SIG}}$ *takes as input a verification key* vk*, a message* m *and a presumptive signature* σ *and outputs a bit* $b \in \{0,1\}$*, with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid*.*

It is required that for every key pair $(\mathsf{vk}, \mathsf{sgk}) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$ *and every* $m \in \{0,1\}^*$*, it holds that* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathsf{vk}, m, \mathrm{Sig}(\mathsf{sgk}, m)) = 1$ *(correctness).*

We say that SIG *is* length-normal *if for all* $m, m' \in \{0,1\}^*$ *such that* $|m| = |m'|$*,* $(\mathsf{vk}, \mathsf{sgk}) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n), \sigma \leftarrow \mathrm{Sig}(\mathsf{sgk}, m), \sigma' \leftarrow \mathrm{Sig}(\mathsf{sgk}, m')$*, it holds that* $|\sigma| = |\sigma'|$*.*

Definition 16 (Existential Unforgeability under Non-Adaptive Chosen Message Attack for Signature Schemes). *We call* SIG *EUF-naCMA-secure if for every* PPT*-adversary* $\mathcal{A}$ *and all* $z \in \{0,1\}^*$ *there exists a negligible function* negl *such that*

$$\mathsf{Pr}[\mathsf{Exp}^{EUF-naCMA}_{\mathcal{A}(z),\mathrm{SIG}}(n) = 1] \leq \mathsf{negl}(n)$$

The experiment $\mathsf{Exp}^{EUF-naCMA}_{\mathcal{A}(z),\mathrm{SIG}}(n)$ *is defined as follows: At the beginning, the experiment generates keys* $(\mathrm{vk}, \mathrm{sgk}) \leftarrow \mathrm{Gen}_{\mathrm{SIG}}(1^n)$*. On input* 1^n*, the adversary* $\mathcal{A}$ *may send queries to a signing oracle* $\mathcal{O}_{\mathsf{Sig}(\mathrm{sgk},\cdot)}$*. Let* $\mathcal{Q}$ *be the set* *of all queries. Afterwards on input* 1^n *and* vk*,* $\mathcal{A}$ *outputs a tuple* (m^*, σ^*)*. If* $\mathrm{Vrfy}_{\mathrm{SIG}}(\mathrm{vk}, m^*, \sigma^*) = 1$ *and* $m^* \notin \mathcal{Q}$*, the experiment outputs 1, else 0.*

C Summary: Corruption Rules

main party P	hackable	(initial) state of P	notify to environment
Corruption			
prior to protocol invocation	*	*	“physical access corruption of P”
Impact: The adversary gets control over P and *all* of its sub-parties regardless of whether they are unhackable. Also, he may choose to ignore enhanced channels of these parties.			
after protocol invocation and prior to input	hackable	online	“online-initiated corruption of P”
Impact: The adversary gets control over P only. The adversary has to adhere to the communication restrictions implied by the enhanced channels of P.			
Hack			
input received	hackable	online	“P hacked” if P is a main party
Impact: The adversary gets control (only) over P. The adversary has to adhere to the communication restrictions implied by the enhanced channels of P.			

Table 1. Corruption Rules

D A Simple Motivating Example for the Corruption Model

Consider a protocol π that consists of two *hackable* parties P_1 and P_2 (who e.g. jointly compute some function) that are connected via a `standard`-channel. Consider another protocol ϕ that is identical to π except that P_1 is connected to P_2 via an `air-gap switch`, which is initially *disconnected* but connected as soon as P_1 receives its input.

Intuitively, π should not emulate ϕ since P_1 has an “open” connection to P_2 in π from the beginning, but not in ϕ. This can indeed be shown in our framework. Consider an environment $\mathcal{Z}$ that invokes P_1 and sets the online/offline-state of P_1’s input port *offline*. Furthermore, consider an adversary $\mathcal{A}$ interacting with π that sends $(\mathtt{attack}, P_1)$ *before* P_1 receives its input but after P_1 is invoked. Since P_1 is hackable and online, $\mathcal{Z}$ will be notified with “online-initiated corruption of P_1”. However, this cannot be simulated in ϕ since P_1 is offline before it receives its input if $\mathcal{Z}$ sets the online/offline-state of P_1’s input port offline.

Conversely, one should be able to argue that ϕ emulates π since ϕ is intuitively more secure than π. This is also possible in our framework, since the simulator interacting with π is able to suppress the `attack` instruction to P_1.

E Proofs of the Properties of the Framework

In this section, we prove various properties of our framework.

Proposition 1 (Completeness of the Dummy Adversary). *Let π and ϕ be protocols. Then, π FUC-emulates ϕ if and only if π FUC-emulates ϕ with respect to the dummy adversary.*

Proof (Sketch). The proof is almost identical to the proof in the UC framework (cf. [Can01]). The only difference is that the environment $\mathcal{Z}_\mathcal{D}$, which internally runs a copy of a given adversary $\mathcal{A}$ and environment $\mathcal{Z}$, forwards the current status (current online/offline state of all parties) of all parties to $\mathcal{A}$ each time $\mathcal{A}$ is activated in $\mathcal{Z}_\mathcal{D}$'s internal simulation. Note that $\mathcal{Z}_\mathcal{D}$ can obtain the status by sending `status` to the dummy adversary $\mathcal{D}$. □

Proposition 2 (Transitivity). *Let π_1, π_2, π_3 be protocols. If $\pi_1 \underset{\#\#}{\geq} \pi_2$ and $\pi_2 \underset{\#\#}{\geq} \pi_3$ then it holds that $\pi_1 \underset{\#\#}{\geq} \pi_3$.*

Proof (Sketch). The proof follows from the same argument as in the UC framework [Can01]. □

Theorem 1 (Equivalence with UC-emulation for en bloc Protocols and their Initial Fortification). *Let π, ϕ be en bloc protocols and $\widetilde{\pi}, \widetilde{\phi}$ their initial fortification. Then,*

$$\pi \underset{\#\#}{\geq} \phi \iff \pi \underset{\mathrm{UC}}{\geq} \phi \iff \widetilde{\pi} \underset{\#\#}{\geq} \widetilde{\phi}$$

Proof (Sketch). These statements follow from the fact that for en bloc protocols and their initial fortifications UC environments can easily simulate environments in our framework and vice versa. This is because in an en bloc protocol or its initial fortification a notify transport is only triggered if a protocol party sends a message to an ideal functionality and each ideal functionality called by a protocol party immediatey notifies the adversary and lets him change the inputs of hacked parties. Also, the online/offline state of a party in an en bloc protocol or its initial fortification can be trivially derived. □

Theorem 2 (Universal Composition). *Let π be a protocol, $\mathcal{F}$ be an ideal functionality (note that $\mathcal{F}$ may be fortified) and $\rho^\mathcal{F}$ a protocol in the $\mathcal{F}$-hybrid model. Then it holds that*

$$\pi \underset{\#\#}{\geq} \mathcal{F} \implies \rho^\pi \underset{\#\#}{\geq} \rho^\mathcal{F}$$

Proof (Sketch). The proof is almost identical to the proof in the UC framework (cf. [Can01]). The main difference is that the environment $\mathcal{Z}_\rho$, which interacts with π and internally runs the protocol ρ and a given environment $\mathcal{Z}$ (and all but one of the instances of π that are called by ρ), determines the online/offline state of each input port to a party in π according to the online/offline state of the respective calling party in ρ in its internal simulation. This way, the online/offline state of the parties in π when interacting with $\mathcal{Z}_\rho$ are the same as in the interaction between ρ^π and $\mathcal{Z}$. Also, if $\mathcal{Z}$ sends an `attack`-instruction in $\mathcal{Z}_\rho$'s internal simulation to a party in π, $\mathcal{Z}_\rho$ forwards that `attack`-instruction to the respective party (if the protocol ρ has already been invoked in its internal simulation but π has not been invoked yet, then $\mathcal{Z}_\rho$ first invokes the respective party before sending the `attack`-instruction) and reports the correct notification to $\mathcal{Z}$ if the party to which $\mathcal{Z}$ has sent the `attack`-instruction is (combined with) a main party in ρ^π. □

F Proof of Lemma 1

In this section, we proof Lemma 1.

Lemma 1 (Restatement). *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*

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

Proof (Sketch). Assume there exists an adversary $\mathcal{A}$ that wins in the experiment $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$ with non-negligible probability. Since PKE is IND-pCCA-secure, one can replace c^* by $c' \leftarrow \mathrm{Enc}(\mathsf{pk}, 0^L)$, 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 $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$ for $\mathcal{A}$ using his signing oracle and c' for c^*. Once $\mathcal{A}$ sends a tuple (m, σ) to the experiment $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$, $\mathcal{A}'$ 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 $\mathsf{Exp}^{aux}_{\mathcal{A}(z),\mathrm{PKE},\mathrm{SIG}}(n)$. □

G Simulator for up to N Corruptions/Hacks, Non-Reactive Case

In the following, we give a detailed description of the simulator for up to N corruptions/hacks (non-reactive case) (cf. Section 4.1).

The simulator Sim' for the case of up to N corruptions/hacks is identical to the simulator for up to $N-1$ in Definition 6, except for the following: Once *all* parties have been *hacked*, Sim', who learns the inputs and outputs of all parties from $[\mathcal{G}]$ in this case, reports plaintext tuples to $\mathcal{Z}$ in such a way that the shares they contain are consistent with the parties' inputs and outputs. Note that $\mathcal{Z}$ cannot check if the tuples it receives from Sim' were encrypted before since it does not have the secret keys.

More specifically, for every *honest* party i, Sim' generates $3N$ random strings $s'_{ij}, r'_{ij}, k'_{ij}$, computes $\sigma'_{ij} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, j, s'_{ij}, r'_{ij}, k'_{ij})$ $(j = 1, \ldots, N)$, and reports $(i, \mathrm{Enc}(\mathsf{pk}_j, i, s'_{ij}, r'_{ij}, k'_{ij}, \sigma'_{ij}))$ $(j \in \{1, \ldots, N\} \setminus \{i\})$ to $\mathcal{Z}$. Furthermore, for each party $i = 1, \ldots, N$, Sim' generates random strings $\tilde{y}_i \leftarrow \{0,1\}^n$.

Once the last party, denoted by PID l^*, has been *hacked*, Sim' computes for each i the shares $\tilde{s}_{il^*} = x_i + \sum_{j \in \{1,\ldots,N\} \setminus \{l^*\}} s'_{ij}$, and $\tilde{k}_{il^*} = k_i + \sum_{j \in \{1,\ldots,N\} \setminus \{l^*\}} k'_{ij}$ and $\tilde{r}_{il^*} = \tilde{y}_i + y_i + \sum_{j \in \{1,\ldots,N\} \setminus \{l^*\}} r'_{ij}$. Sim' then computes $\tilde{\sigma}_{il^*} \leftarrow \mathrm{Sig}(\mathsf{sgk}_i, l^*, \tilde{s}_{il^*}, \tilde{r}_{il^*}, \tilde{k}_{il^*})$ and reports the tuples $(i, \tilde{s}_{il^*}, \tilde{r}_{il^*}, \tilde{k}_{il^*}, \tilde{\sigma}_{il^*})$ $(i = 1, \ldots, N)$. When $\mathcal{Z}$ sends a vector of ciphertexts to the Dec-unit of party l^*, then Sim' checks for each c' contained in that vector if $c' = c^i_{l^*}$ for some i. For each c' for which this holds, Sim' returns the corresponding $(i, \tilde{s}_{il^*}, \tilde{r}_{il^*}, \tilde{k}_{il^*}, \tilde{\sigma}_{il^*})$. Otherwise, Sim' returns $\mathrm{Dec}(\mathsf{sk}^*_l, c')$. When $\mathcal{Z}$ sends a vector of ciphertexts to the Dec-unit of a party $i \neq l^*$, Sim' decrypts each ciphertext contained in that vector using sk_i.

If *all* parties are marked as `genuine`, then for every *corrupted* or *hacked* party i, Sim' sends $(\tilde{y}_i, \mathrm{MAC}(k_i, \tilde{y}_i))$ to $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$. If one of the parties is marked as `fake`, then for every *corrupted* or *hacked* party i, Sim' sends $\perp$ to $\mathcal{Z}$ as output from $\mathcal{F}_{\mathcal{G}}$.

H A Short Introduction into UC Security

In the following, we give a brief overview of the UC framework. The following is taken from [BDH+17].

The essential idea is to define security by means of the indistinguishability between an experiment in which the task at hand is carried out by dummy parties with the help of an ideal incorruptible entity and an experiment in which the parties must conduct the task themselves. In contrast to previous attempts to define security by simulation the indistinguishability must not only hold after the protocol execution has completed, but the distinguisher—called the environment $\mathcal{Z}$—takes part in the experiment, orchestrates all adversarial attacks, supplies the inputs to the parties running the challenge protocol and can observe the parties' output as well as communication during the whole protocol execution.

The basic model of computation The basic model of computation consists of a set of (possibly polynomial many) instances (ITIs) of interactive Turing machines (ITMs). An interactive Turing machine (ITM) is the description of a Turing machine with additional tapes, namely the identity tape, tapes for subroutine input and output as well as tapes for incoming and outgoing network messages. The tangible instantiation of an ITM—the ITI—is identified by the content of its identity tape which consists of an session and a party identifier (SID/PID). The order of activation of the ITIs is completely asynchronous and message-driven. An ITI gets activated if either subroutine input or an incoming message is written onto its respective tape. If the ITI provides subroutine output or writes an outgoing message, the activation of the ITI completes and the ITI to whom the message has been delivered to gets activated next. Each experiment comprises two special ITIs the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$ (in the real experiment) or the simulator Sim (in the ideal experiment). The environment is the ITI that is initially activated. If during the execution any ITI completes its activation without giving any output, the environment is activated again as a fall-back. If the environment $\mathcal{Z}$ conducts a subroutine output, the whole experiments stops. The output of the experiment is the output of $\mathcal{Z}$.

The adversary The adversary $\mathcal{A}$ has the following capabilities. If any ITI writes an outgoing message the message is not directly delivered to the incoming tape of designated receiver but the adversary is responsible for all message transfers. To this end every message is implicitly copied to the incoming message tape of the adversary. The adversary can process the message arbitrarily. The adversary may decide to deliver to message (by writing the message on its own outgoing tape), the adversary may postpone or completely suppress the message, inject new messages or alter messages in any way including the recipient and/or alleged sender. This modeling reflects the idea of an unreliable and untrusted network. Please note twofold: (a) Only incoming/outgoing messages are under the control of the adversary, subroutine input/output between ITIs is immediate and trustworthy as long as the ITIs are *uncorrupted*. (b) As the sequence of activations is message-driven the adversary also controls the scheduling and order of execution. Moreover the adversary can *corrupt* an ITI. In this case the adversary learns the complete entire state of the corrupted ITI and takes over its execution. This means whenever the corrupted ITI would have been activated (even due to subroutine input) the adversary gets activated with the same input.

The real experiment In the real experiment for a challenge protocol π, denoted by $\text{Exec}(\pi, \mathcal{A}, \mathcal{Z})$, the environment $\mathcal{Z}$ is activated first. After the invocation of the adversary $\mathcal{A}$ the environment $\mathcal{Z}$ requests the creation of the challenge protocol. The main parties of π become subroutines of the environment and the environment freely choses their input and the SID of the challenge protocol. The experiment is executed as outlined above.

The ideal experiment In the ideal experiment, denoted by $\text{Exec}(\mathcal{F}, \mathcal{S}, \mathcal{Z})$, the challenge protocol is silently replaced by an instance of $\mathcal{F}$ together with dummy parties. The dummy parties obtain a common session identifier (SID) and individual party identifiers (PIDs) from the environment as if they were the actual main parties of the protocol π in the real experiment, however they merely forward the subroutine input/output between the instance of the functionality $\mathcal{F}$ and the environment. The ideal functionality $\mathcal{F}$ is simultaneously a subroutine for each dummy party, holds the same SID but no PID, and conducts the prescribed task without the necessity to exchange any network messages. Moreover, in the ideal experiment the adversary is replaced by a simulator Sim that mimics the adversarial behavior to the environment as if this was the real experiment with real parties carrying out the real protocol with real π-messages.

Definition of Security A protocol π is said to UC-realize an ideal functionality $\mathcal{F}$, denoted by $\pi \underset{\text{UC}}{\geq} \mathcal{F}$, iff

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z} : \text{Exec}(\pi, \mathcal{A}, \mathcal{Z}) \overset{\text{c}}{\equiv} \text{Exec}(\mathcal{F}, \mathcal{S}, \mathcal{Z}) \tag{1}$$

holds, where the randomness on the left and on the right is taken over the initial input of $\mathcal{Z}$ and all random tapes of all PPT machines.