Universally Composable Subversion-Resilient Cryptography

Suvradip Chakraborty[1⋆], Bernardo Magri[2⋆⋆], Jesper Buus Nielsen[3], and Daniele Venturi[4⋆⋆⋆]

[1] ETH Zurich
[2] The University of Manchester
[3] Aarhus University
[4] Sapienza University of Rome

Abstract Subversion attacks undermine security of cryptographic protocols by replacing a legitimate honest party's implementation with one that leaks information in an undetectable manner. An important limitation of all currently known techniques for designing cryptographic protocols with security against subversion attacks is that they do not automatically guarantee security in the realistic setting where a protocol session may run concurrently with other protocols.

We remedy this situation by providing a foundation of *reverse firewalls* (Mironov and Stephens-Davidowitz, EUROCRYPT'15) in the *universal composability* (UC) framework (Canetti, FOCS'01 and J. ACM'20). More in details, our contributions are threefold:

- We generalize the UC framework to the setting where each party consists of a core (which has secret inputs and is in charge of generating protocol messages) and a firewall (which has no secrets and sanitizes the outgoing/incoming communication from/to the core). Both the core and the firewall can be subject to different flavors of corruption, modeling different kinds of subversion attacks. For instance, we capture the setting where a subverted core looks like the honest core to any efficient test, yet it may leak secret information via covert channels (which we call *specious subversion*).
- We show how to sanitize UC commitments and UC coin tossing against specious subversion, under the DDH assumption.
- We show how to sanitize the classical GMW compiler (Goldreich, Micali and Wigderson, STOC 1987) for turning MPC with security in the presence of semi-honest adversaries into MPC with security in the presence of malicious adversaries. This yields a completeness theorem for maliciously secure MPC in the presence of specious subversion.

Additionally, all our sanitized protocols are *transparent*, in the sense that communicating with a sanitized core looks indistinguishable from communicating with an honest core. Thanks to the composition theorem, our methodology allows, for the first time, to design subversion-resilient protocols by sanitizing different sub-components in a modular way.

⋆ Work done while at IST Austria; supported in part by ERC grant 724307.
⋆⋆ Work done while at Aarhus University.
⋆⋆⋆ Supported by the grant SPECTRA from Sapienza University of Rome.

1 Introduction

Cryptographic schemes are typically analyzed under the assumption that the machines run by honest parties are fully trusted. Unfortunately, in real life, there are a number of situations in which this assumption turns out to be false. In this work, we are concerned with one of these situations, where the adversary is allowed to subvert the implementation of honest parties in a stealthy way. By stealthy, we mean that the outputs produced by a subverted machine still look like honestly computed outputs, yet, the adversary can use such outputs to completely break security. Prominent examples include backdoored implementations [16,15,18] and algorithm-substitution (or *kleptographic*) attacks [22,23,3,2,4]. The standardization of the pseudorandom number generator Dual_EC_DRBG, as exposed by Snowden, is a real-world instantiation of the former, while Trojan horses, as in the case of the Chinese hack chip attack, are real-world instantiations of the latter.

1.1 Subversion-Resilient Cryptography

Motivated by these situations, starting from the late 90s, cryptographers put considerable effort into building cryptographic primitives and protocols that retain some form of security in the presence of *subversion attacks*.

Yet, after nearly 30 years of research, all currently known techniques to obtain subversion resilience share the limitation of only implying *standalone* security, *i.e.* they only guarantee security of a protocol in isolation, but all bets are off when such a protocol is used in a larger context in the presence of subversion attacks. This shortcoming makes the design of subversion-resilient cryptographic protocols somewhat cumbersome and highly non-modular. For instance, Ateniese, Magri, and Venturi [1] show how to build subversion-resilient signatures, which in turn were used by Dodis, Mironov and Stephens-Davidowitz [17] to obtain subversion-resilient key agreement protocols, and by Chakraborty, Dziembowski and Nielsen [11] to obtain subversion-resilient broadcast; however, the security analysis in both [17] and [11] reproves security of the construction in [1] from scratch. These examples bring the fundamental question:

Can we obtain subversion resistance in a composable *security framework?*

A positive answer to the above question would dramatically simplify the design of subversion-resilient protocols, in that one could try to first obtain security under subversion attacks for simpler primitives, and then compose such primitives in an arbitrary way to obtain protocols for more complex tasks, in a modular way.

1.2 Our Contributions

In this work, we give a positive answer to the above question using so-called *cryptographic reverse firewalls*, as introduced by Mironov and Stephens-Davidowitz [21]. Intuitively, a reverse firewall is an external party that sits between an honest

party and the network, and whose task is to sanitize the incoming/outgoing communication of the party it is attached to, in order to annhilate subliminal channels generated via subversion attacks. The main challenge is to obtain sanitation while maintaining the correctness of the underlying protocol, and in a setting where other parties may be completely under control of the subverter itself.

While previous work showed how to build reverse firewalls for different cryptographic protocols in *standalone* security frameworks we provide a foundation of reverse firewalls in the framework of universal composability (UC) of Canetti [7,6]. More in details, our contributions are threefold:

- We generalize the UC framework to the setting where each party consists of a core (which has secret inputs and is in charge of generating protocol messages) and a firewall (which has no secrets and sanitizes the outgoing/incoming communication from/to the core). Both the core and the firewall can be subject to different flavors of corruption, modeling different kinds of subversion attacks. For instance, we capture the setting where a subverted core looks like the honest core to any efficient test, yet it may leak secret information via covert channels (which we call *specious subversion*).
- We show how to sanitize UC commitments and UC coin tossing against specious subversion, under the decisional Diffie-Hellman (DDH) assumption in the common reference string (CRS) model. Our sanitized commitment protocol is non-interactive, and requires 2λ group elements in order to commit to a λ-bit string; the CRS is made of 3 group elements.
- We show how to sanitize the classical compiler by Goldreich, Micali and Wigderson (GMW) [20] for turning multiparty computation (MPC) with security against semi-honest adversaries into MPC with security against malicious adversaries. This yields a completeness theorem for maliciously secure MPC in the presence of specious subversion.

Additionally, all our sanitized protocols are *transparent*, in the sense that communicating with a sanitized core looks indistinguishable from communicating with an honest core. Thanks to the composition theorem, our methodology allows, for the first time, to design subversion-resilient protocols by sanitizing different sub-components in a modular way.

1.3 Technical Overview

Below, we provide an overview of the techniques we use in order to achieve our results, starting with the notion of subversion-resilient UC security, and then explaining the main ideas behind our reverse firewalls constructions.

Subversion-resilient UC Security At a high level we model each logical party P_i of a protocol Π as consisting of two distinct parties of the UC framework, one called the core C_i and one called the firewall F_i. These parties can be independently corrupted. For instance, the core can be subverted and the firewall honest, or

the core could be honest and the firewall corrupted. The ideal functionalities $\mathcal{F}$ implemented by such a protocol will also recognize two UC parties per virtual party and can let their behavior depend on the corruption pattern. For instance, $\mathcal{F}$ could specify that if C_i is subverted and F_i honest, then it behaves as if P_i is honest on $\mathcal{F}$. Or it could say that if C_i is honest and F_i corrupt, then it behaves as if P_i is honest but might abort on $\mathcal{F}$. This is a reasonable choice as a corrupt firewall can always cut C_i off from the network and force an abort. We then simply ask that Π UC-realizes $\mathcal{F}$. By asking that Π UC-realizes $\mathcal{F}$ we exactly capture that if the core is subverted and the firewall is honest, this has the same effect as P_i being honest. See Table 1 for all possible corruption combinations for C_i and F_i at a glance, and how they translate into corruptions for P_i in an ideal execution with functionality $\mathcal{F}$.

Unfortunately, it turns out that for certain functionalities it is just impossible to achieve security in the presence of *arbitrary* subversion attacks. For instance, a subverted prover in a zero-knowledge proof could simply output an honestly computed proof or the all-zero string depending on the first bit of the witness. Since the firewall would not know a valid witness, these kind of subversion attacks cannot be sanitized. For this reason, following previous work [21,17,19,11], we focus on classes of subversion attacks for which a subverted core looks like an honest core to any efficient test, yet it may signal private information to the subverter via subliminal channels. We call such corruptions *specious*. We note that testing reasonably models a scenario in which the core has been built by an untrusted manufacturer who wants to stay covert, and where the user tests it against a given specification before using it in the wild.

By defining subversion resilience in a black-box way, via the standard notion of UC implementation, we also get composition almost for free via the UC composition theorem. One complication arises to facilitate modular composition of protocols. When doing a modular construction of a subversion-resilient protocol, both the core and the firewall will be built by modules. For instance, the core could be built from a core for a commitment scheme and the core for an outer protocol using the commitment scheme. Each of these cores will come with their own firewall: one sanitizes the core of the commitment scheme; the other sanitizes the core of the outer protocol. The overall firewall is composed of these two firewalls. It turns out that it is convenient that these two firewalls can coordinate, as it might be that some of the commitments sent need to have the message randomized, while others might only have their randomness refreshed. The latter can be facilitated by giving the firewall of the commitment scheme a sanitation interface where it can be instructed by the outer firewall to do the right sanitation. Note that the protocol implementing the commitment ideal functionality now additionally needs to implement this sanitation interface.

We refer the reader to Section 2 for a formal description of our model. Note that another natural model would have been to have P_i split into three parts (or tiers), C_i, U_i, and F_i, where: (i) U_i is a user program which gets inputs and sends messages on the network; (ii) C_i is a core holding cryptographic keys and implementations of, *e.g.*, signing and encryption algorithms; and (iii) F_i is a

firewall used by U_i to sanitize messages to and from C_i in order to avoid covert channels. The above better models a setting where we are only worried that some part of the computer might be subverted. The generalisation to this case is straightforward given the methodology we present for the case with no user program U_i. Since we only look at subversions which are indistinguishable from honest implementations, having the "unsubvertable" U_i appears to give no extra power. We therefore opted for the simpler model for clarity. Further discussion on the three-tier model can be found the full version [12].

Strong sanitation. The main challenge when analyzing subversion security of a protocol in our framework is that, besides maliciously corrupting a subset of the parties, the adversary can, *e.g.*, further speciously corrupt the honest parties. To overcome this challenge, we introduce a simple property of reverse firewalls which we refer to as *strong sanitation*. Intuitively, this property says that no environment, capable of doing specious corruptions of an honest core in the real world, can distinguish an execution of the protocol with one where an honest core is replaced with a so-called *incorruptible* core (that simply behaves honestly in case of specious corruption). The latter, of course, requires that the firewall of the honest core is honest.

We then prove a general lemma saying that, whenever a firewall has strong sanitation, it is enough to prove security in our model without dealing with specious corruptions of honest parties. This lemma significantly simplifies the security analysis of protocols in our model.

Commitments In Section 3, we show how to obtain subversion-resilient UC commitments. First, we specify a sanitizable string commitment functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$. This functionality is basically identical to the standard functionality for UC commitments [8], except that the firewall is allowed to sanitize the value s that the core commits to, using a blinding factor r; the effect of this sanitation is that, when the core opens the commitment, the ideal functionality reveals $\hat{s} = s \oplus r$. Note that this is the sanitation allowed by the sanitation interface. An implementation will further have to sanitise the randomness of outgoing commitments to avoid covert channels.

Second, we construct a protocol $\widehat{\Pi}_{\mathsf{sCOM}}$ that UC realizes $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ in the presence of subversion attacks. Our construction borrows ideas from a recent work by Canetti, Sarkar and Wang [10], who showed how to construct efficient non-interactive UC commitments with adaptive security. The protocol, which is in the CRS model and relies on the standard DDH assumption, roughly works as follows. The CRS is a tuple of the form (g, h, T_1, T_2), such that $T_1 = g^x$ and $T_2 = h^{x'}$ for $x \neq x'$ (*i.e.*, a non-DH tuple). In order to commit to a single bit b, the core of the committer encodes b as a value $u \in \{-1, 1\}$ and outputs $B = g^\alpha \cdot T_1^u$ and $H = h^\alpha \cdot T_2^u$, where α is the randomness. The firewall sanitizes a pair (B, H) by outputting $\widehat{B} = B^{-1} \cdot g^\beta$ and $\widehat{H} = H^{-1} \cdot h^\beta$, where β is chosen randomly; note that, upon receiving an opening (b, α) from the core, the firewall can adjust it by returning $(1 - b, -\alpha + \beta)$. Alternatively, the firewall can choose to leave the bit b

unchanged and only refresh the randomness of the commitment; this is achieved by letting $\widehat{B} = B \cdot g^{\beta}$ and $\widehat{H} = H \cdot h^{\beta}$; in this case, the opening is adjusted to $(b, \alpha + \beta)$. In the security proof, we distinguish two cases:

- In case the committer is maliciously corrupt, the simulator sets the CRS as in the real world but additionally knows the discrete log t of h to the base g. Such a trapdoor allows the simulator to extract the bit b corresponding to the malicious committer by checking whether $H/T_2 = (B/T_1)^t$ (in which case $b = 1$) or $H \cdot T_2 = (B \cdot T_1)^t$ (in which case $b = 0$). If none of the conditions hold, no opening exists.
- In case the committer is honest, the simulator sets the CRS as a DH-tuple. Namely, now $T_1 = g^x$ and $T_2 = h^x$ for some x known to the simulator. The latter allows the simulator to fake the commitment as $B = g^{\alpha}$ and $H = h^{\alpha}$, and later adjust the opening to any given $u \in \{-1, 1\}$ (and thus $b \in \{0, 1\}$) by letting $\alpha' = \alpha - u \cdot x$.

The above ideas essentially allow to build a simulator for the case of two parties, where one is maliciously corrupt and the other one has an honest core and a semi-honest firewall. These ideas can be generalized to n parties (where up to $n - 1$ parties are maliciously corrupt, while the remaining party has an honest core and a semi-honest firewall) using an independent CRS for each pair of parties. Finally, we show that the firewall in our protocol is strongly sanitizing and thus all possible corruption cases reduce to the previous case. In particular, strong sanitation holds true because a specious core must produce a pair (B, H) of the form $B = g^{\alpha} \cdot T_1^{\tilde{u}}$ and $H = h^{\alpha} \cdot T_2^{\tilde{u}}$ for some $\tilde{u} \in \{-1, 1\}$ (and thus $\tilde{b} \in \{0, 1\}$), as otherwise a tester could distinguish it from an honest core by asking it to open the commitment; given such a well-formed commitment, the firewall perfectly refreshes its randomness (and eventually blinds the message).

As we show in Section 3, the above construction can be extended to the case where the input to the commitment is a λ-bit string by committing to each bit individually; the same CRS can be reused across all of the commitments.

Coin Tossing Next, in Section 4, we show a simple protocol that UC realizes the standard coin tossing functionality $\mathcal{F}_{\mathsf{TOSS}}$ in the presence of subversion attacks. Recall that the ideal functionality $\mathcal{F}_{\mathsf{TOSS}}$ samples a uniformly random string $s \in \{0, 1\}^{\lambda}$ and sends it to the adversary, which can then decide which honest party gets s (*i.e.*, the coin toss output).

Our construction is a slight variant of the classical coin tossing protocol by Blum [5]; the protocol is in the $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$-hybrid model, and roughly works as follows. The core of each party commits to a random string $s_i \in \{0, 1\}^{\lambda}$ through the ideal functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$. Then, the firewall of the coin toss instructs the firewall of the commitment to blind s_i using a random blinding factor $r_i \in \{0, 1\}^{\lambda}$ which is revealed to the core. At this point, each (willing) party opens the commitment, which translates into $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ revealing $\hat{s}_j = s_j \oplus r_j$, and each party finally outputs $s = s_i \oplus r_i \oplus \bigoplus_{j \neq i} \hat{s}_j$.

In the security proof, the simulator can fake the string s_i of an honest party so that it matches the output of the coin tossing s (received from $\mathcal{F}_{\mathsf{TOSS}}$), the strings s_j received from the adversary (on behalf of a malicious core), and the blinding factor r_i received from the adversary (on behalf of a semi-honest firewall). This essentially allows to build a simulator for the case where up to $n-1$ parties are maliciously corrupt, while the remaining party has an honest core and a semi-honest firewall. Finally, we show that the firewall in our protocol is strongly sanitizing and thus all possible corruption cases reduce to the previous case. Strong sanitation here holds because any string s_i chosen by a specious core is mapped to a uniformly random string $\hat{s}_i$ via the sanitation interface of the functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$.

Completeness Theorem Finally, in Section 5, we show how to sanitize the GMW compiler, which yields a completeness theorem for UC subversion-resilient MPC. Recall that in the classical GMW compiler one starts with an MPC protocol Π tolerating $t < n$ semi-honest corruptions and transforms it into an MPC protocol tolerating t malicious corruptions as follows. First, the players run an augmented coin-tossing protocol, where each party receives a uniformly distributed string (to be used as its random tape) and the other parties receive a commitment to that string. Second, each party commits to its own input and proves in zero knowledge that every step of the protocol Π is executed correctly and consistently with the random tape and input each party is committed to.

As observed by Canetti, Lindell, Ostrovsky and Sahai [9], the above compilation strategy cannot immediately be translated in the UC setting, as the receiver of a UC commitment obtains no information about the value that was committed to. Hence, the parties cannot prove in zero knowledge statements relative to their input/randomness commitment. This issue is resolved by introducing a commit-and-prove ideal functionality, which essentially allows each party to commit to a witness and later prove arbitrary NP statements relative to the committed witness.

In order to sanitize the GMW compiler in the presence of subversion attacks, we follow a similar approach. Namely, we first introduce a sanitazable commit-and-prove functionality $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$. This functionality is very similar in spirit to the standard commit-and-prove functionality, except that the firewall can decide to blind the witness that the core commits to. In the full version [12], we show how to realize the sanitizable commit-and-prove functionality in the CRS model from the DDH assumption, using re-randomizable non-interactive zero-knowledge arguments for all of NP [13]. In fact, there we exhibit a much more general construction that can be instantiated from any so-called *malleable mixed commitment*, a new notion that we introduce and that serves as a suitable abstraction of our DDH-based construction from Section 3.

In the actual protocol, we use both the coin tossing functionality $\mathcal{F}_{\mathsf{TOSS}}$ and the sanitizable commit-and-prove functionality $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ to determine the random tape of each party as follows. Each core commits to a random string s_i via $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$; the corresponding firewall blinds s_i with a random r_i that is revealed to the core.

Thus, the players use $\mathcal{F}_{\mathsf{TOSS}}$ to generate public randomness s_i^* that can be used to derive the random tape of party P_i as $s_i^* \oplus (s_i \oplus r_i)$. Moreover each core commits to its own input x_i, which however is not blinded by the firewall. The above allows each party, during the protocol execution, to prove via $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ that each message has been computed correctly and consistently with the committed input and randomness derived from the public random string s_i^* received from $\mathcal{F}_{\mathsf{TOSS}}$.

The security analysis follows closely the one in [9], except that in our case we show that any adversary corrupting up to t parties maliciously, and the firewall of the remaining honest parties semi-honestly, can be reduced to a semi-honest adversary attacking Π. Since we additionally show that our firewall is strongly sanitizing, which essentially comes from the ideal sanitation interface offered by $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, all possible corruption cases reduce to the previous case.

2 A UC Model of Reverse Firewalls

In this section we propose a foundation of reverse firewalls in the UC model [7]. We use the UC framework for concreteness as it is the *de facto* standard. However, we keep the description high level and do not depend on very particular details of the framework. Similar formalizations could be given in other frameworks defining security via comparison to ideal functionalities, as long as these ideal functionalities are corruption aware: they know which parties are corrupted and their behavior can depend on it.

2.1 Quick and Dirty Recap of UC

A protocol Π consists of code for each of the parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The parties can in turn make calls to ideal functionalities $\mathcal{G}$. More precisely, the code of the program is a single machine. As part of its input, it gets a party identifier pid which tells the code which party it should be running the code for. This allows more flexibility for dynamic sets of parties. Below, we will only consider programs with a fixed number of parties. We are therefore tacitly identifying n parties identifiers $\mathsf{pid}_1, \ldots, \mathsf{pid}_n$ with the n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$, *i.e.*, $\mathsf{P}_i = \mathsf{pid}_i$. We prefer the notation P_i for purely idiomatic reasons.

A party P_i can call an ideal functionality. To do so it will specify which $\mathcal{G}$ to call (technically it writes down the code of $\mathcal{G}$ and a session identifier sid distinguishing different calls), along with an input x. Then, $(\mathsf{sid}, \mathsf{pid}, x)$ is given to $\mathcal{G}$. If $\mathcal{G}$ does not exists, then it is created from its code.

There is an adversary $\mathcal{A}$ which attacks the protocol. It can corrupt parties via special corruption commands. How parties react to these corruptions is flexible; the parties can in principle be programmed to react in any efficient way. As an example, in response to input `active-corrupt`, we might say that the party in the future will output all its inputs to the adversary, and that it will let the adversary specify what messages the party should send. The adversary can also control ideal functionalities, if the ideal functionalities expose an interface for

that. It might for instance be allowed to influence at what time messages are delivered on an ideal functionality of point-to-point message transmission.

There is also an environment $\mathcal{E}$ which gives inputs to the parties and sees their outputs. The environment can talk freely to the adversary. A real world execution $\textsc{Exec}_{\Pi,\mathcal{A},\mathcal{E}}$ is driven by the environment which can activate parties or ideal functionalities. The parties and ideal functionalities can also activate each other. The details of activation are not essential here, and can be found in [7].

The protocol Π is meant to implement an ideal functionality $\mathcal{F}$. This is formulated by considering a run of $\mathcal{F}$ with dummy parties which just forward messages between $\mathcal{E}$ and $\mathcal{F}$. In addition, there is an adversary $\mathcal{S}$, called the simulator, which can interact with $\mathcal{F}$ on the adversarial interface, and which can interact freely with $\mathcal{E}$ as an adversary can. The simulation is the process $\textsc{Exec}_{\mathcal{F},\mathcal{S},\mathcal{E}}$, where we do not specify the dummy protocol but use $\mathcal{F}$ for the dummy protocol composed with $\mathcal{F}$. We say that Π UC-realizes $\mathcal{F}$ if there exists an efficient simulator which makes the simulation look like the real world execution to any efficient environment:

$$\exists \mathcal{S} \forall \mathcal{E} : \textsc{Exec}_{\Pi,\mathcal{A},\mathcal{E}} \approx \textsc{Exec}_{\mathcal{F},\mathcal{S},\mathcal{E}},$$

where $\mathcal{A}$ is the dummy adversary (that simply acts as a proxy for the environment), and where the quantifications are over poly-time interactive Turing machines.

Consider a protocol Π that realizes an ideal functionality $\mathcal{F}$ in a setting where parties can communicate as usual, and additionally make calls to an unbounded number of copies of some other ideal functionality $\mathcal{G}$. (This model is called the $\mathcal{G}$-hybrid model.) Furthermore, let Γ be a protocol that UC-realizes $\mathcal{G}$ as sketched above, and let $\Pi^{\mathcal{G}\to\Gamma}$ be the composed protocol that is identical to Π, with the exception that each interaction with the ideal functionality $\mathcal{G}$ is replaced with a call to (or an activation of) an appropriate instance of the protocol Γ. Similarly, any output produced by the protocol Γ is treated as a value provided by the functionality $\mathcal{G}$. The composition theorem states that in such a case, Π and $\Pi^{\mathcal{G}\to\Gamma}$ have essentially the same input/output behavior. Namely, Γ behaves just like the ideal functionality $\mathcal{G}$ even when composed with an arbitrary protocol Π. A special case of this theorem states that if Π UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, then $\Pi^{\mathcal{G}\to\Gamma}$ UC-realizes $\mathcal{F}$.

2.2 Modeling Reverse Firewalls

To model reverse firewalls, we will model each party P_i as two separate parties in the UC model: the core C_i and the firewall F_i. To be able to get composability for our framework via UC composition, we model them as separate parties each with their own party identifier $(\mathsf{pid}, \mathsf{F})$ and $(\mathsf{pid}, \mathsf{C})$. We use pid to denote the two of them together. Below we write, for simplicity, P_i to denote the full party, C_i to denote the core, and F_i to denote the firewall. Being two separate parties, the core and the firewall cannot talk directly. It will be up to the ideal functionality $\mathcal{G}$ used for communication to pass communication with the core through the corresponding firewall before acting on the communication. It might be that

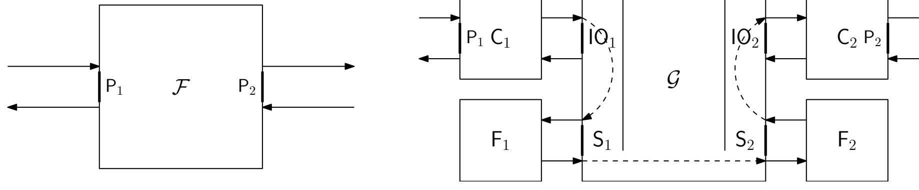

Figure 1. Implementing a normal functionality $\mathcal{F}$ using a sanitizable hybrid functionality $\mathcal{G}$ and a sanitizing protocol $\Pi = (\mathsf{C}, \mathsf{F})$. Cores and firewalls talk to sanitizable functionalities directly. Cores can additionally talk to the environment to exchange inputs and outputs. Firewalls only talk to ideal functionalities. We think of ideal functionalities as sanitizing the communication with the core via the firewall. This is illustrated in the figure by information from the core going to the firewall, and information to the core coming via the firewall. There is no formal requirement to what extent this happens; it is up to the ideal functionality to decide what type of sanitation is possible, if any.

when $\mathcal{G}$ gets a message from C_i it will output this message to F_i and allow F_i to change the message, possibly under some restrictions. We say that F_i sanitizes the communication, and we call the interface connecting F_i for $\mathcal{G}$ the sanitation interface of $\mathcal{G}$. We call such an ideal functionality a "sanitizable" ideal functionality.

Consider a party $(\mathsf{C}_i, \mathsf{F}_i)$ with core C_i and firewall F_i connected to a sanitizing ideal functionality $\mathcal{G}$. The idea is that the firewall gets to sanitize all communication of the core C_i. The UC model seemingly allows a loophole, as the core could make a call to some other ideal functionality $\mathcal{H}$ instead of talking to $\mathcal{G}$. As we discuss later, this behavior is ruled out if C_i is specious, so we will not explicitly disallow it. If our model is later extended to allow stronger (non-specious) types of subversion, then one would probably have to explicitly forbid C_i to use this loophole.

When using a sanitizable ideal functionality, it is convenient to be able to distinguish the interface of the ideal functionality from the parties using the interface. We call the interface of $\mathcal{G}$ to which the core of P_i is connected the input-output interface, IO. We call the party connected to it C_i. We call the interface of $\mathcal{G}$ to which the firewall of P_i is connected the sanitation interface, S. We call the party connected to it F_i. This is illustrated in Fig. 1.

2.3 Specious Corruptions

A major motivation for studying subversion resilience is to construct firewalls which ensure that security is preserved even if the core is subverted. In this section, we describe and discuss how we model subversion in the UC framework.

In a nutshell, we let the adversary replace the code of the core. Clearly, if the core is arbitrarily corrupted, it is impossible to guarantee any security. We therefore have to put restrictions on the code used to subvert the core. One can

consider different types of subversions. In this work, we will consider a particularly "benign" subversion, where the subverted core looks indistinguishable from the honest core to any efficient test. This is a particularly strong version of what has been called "functionality preservation" in other works [21,17,19,11]. As there are slightly diverting uses of this term we will coin a new one to avoid confusion.

The central idea behind our notion is that we consider corruptions where a core C_i has been replaced by another implementation $\widetilde{\mathsf{C}}_i$ which cannot be distinguished from C_i by black-box access to $\widetilde{\mathsf{C}}_i$ or C_i. We use the term *specious* for such corruptions, as they superficially appear to be honest, but might not be.

More in details, we define specious corruptions via testing. Imagine a test T which is given non-rewinding black-box access to either C_i or $\widetilde{\mathsf{C}}_i$, and that tries to guess which one it interacted with. We say that a subversion is specious if it survives all efficient tests. This is a very strong notion. One way to motivate this notion could be that $\widetilde{\mathsf{C}}_i$ might be built by an untrusted entity, but the buyer of $\widetilde{\mathsf{C}}_i$ can test it up against a specification. If the untrusted entity wants to be sure to remain covert, it would have to do a subversion that survives all tests. We assume that the test does not have access to the random choices made by $\widetilde{\mathsf{C}}_i$. This makes the model applicable also to the case where $\widetilde{\mathsf{C}}_i$ is a blackbox or uses an internal physical process to make random choices. We will allow the entity doing the subversion to have some auxiliary information about the subversion and its use of randomness. This will, for instance, allow the subversion to communicate with the subverter in a way that cannot be detected by any test (*e.g.*, using a secret message acting as a trigger).

For a machine T and an interactive machine $\widetilde{\mathsf{C}}$, we use $\mathsf{T}^{\widetilde{\mathsf{C}}}$ to denote that T has non-rewinding black-box access to $\widetilde{\mathsf{C}}$. If during the run of $\mathsf{T}^{\widetilde{\mathsf{C}}}$ the machine $\widetilde{\mathsf{C}}$ requests a random bit, then a uniformly random bit is sampled and given to $\widetilde{\mathsf{C}}$. Such randomness is not shown to T. We define the following game for an efficiently sampleable distribution D and a test T.

- Sample $(\widetilde{\mathsf{C}}, a) \leftarrow \mathsf{D}$, where a is an auxiliary string.
- Sample a uniformly random bit $b \in \{0, 1\}$:
  - If $b = 0$, then run $\mathsf{T}^{\widetilde{\mathsf{C}}}$ to get a guess $g \in \{0, 1\}$.
  - If $b = 1$, then run T^{C} to get a guess $g \in \{0, 1\}$.
- Output $c = b \oplus g$.

Let $\textsc{Test}_{\mathsf{D},\mathsf{T}}$ denote the probability that $c = 0$, *i.e.*, the probability that the guess at b is correct.

Definition 1 (Specious subversion). *We say that* D *is computationally specious if for all PPT tests* T *it holds that* $\textsc{Test}_{\mathsf{D},\mathsf{T}} - 1/2$ *is negligible.*

We return to the discussion of the loophole for specious cores of creating other ideal functionalities $\mathcal{H}$ that are not sanitizing. Note that if a core creates an ideal functionality that it is not supposed to contact, then this can be seen by testing. Therefore, such a core is not considered specious. Hence, the notion of specious closes the loophole.

The notion of specious is strong, as it requires that no test T can detect the subversion. At first glance it might even look too strong, as it essentially implies that the subversion is correct. However, as we show next, a specious subversion can still signal to the outside in an undetectable manner. To formalize this notion, we define the following game for an efficiently sampleable distribution D, an adversary A and a decoder Z.

- Sample $(\widetilde{\mathsf{C}}, a) \leftarrow \mathsf{D}$, where a is an auxiliary string.
- Sample a uniformly random bit $b \in \{0,1\}$:
  - If $b = 0$, then run $\mathsf{A}^{\widetilde{\mathsf{C}}}$ to get a signal $s \in \{0,1\}^*$.
  - If $b = 1$, then run A^{C} to get a signal $s \in \{0,1\}^*$.
- Run $\mathsf{Z}(a, s)$ to get a guess $g \in \{0,1\}$.
- Output $c = b \oplus g$.

Let $\textsc{Signal}_{\mathsf{D},\mathsf{A},\mathsf{Z}}$ denote the probability that $c = 0$, *i.e.*, the probability that the guess at b is correct.

Definition 2 (Signaling). *We say that* D *is computationally signalling if there exists a PPT adversary* A *and a PPT decoder* Z *such that* $\textsc{Signal}_{\mathsf{D},\mathsf{A},\mathsf{Z}} - 1/2$ *is non-negligible.*

Lemma 1. *There exist a machine* C*, and an efficiently sampleable distribution* D*, such that* D *is both computationally specious and signaling.*

Proof (Proof sketch). Consider a machine C that when queried outputs a fresh uniformly random $y \in \{0,1\}^\lambda$. Let $\Phi = \{\phi_\kappa : \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda\}_{\kappa \in \{0,1\}^\lambda}$ be a family of pseudorandom permutations. Consider the subversion $\widetilde{\mathsf{C}}$ of C that hardcodes a key $\kappa \in \{0,1\}^\lambda$ and: (i) when initialised samples a uniformly random counter $x \in \{0,1\}^\lambda$; (ii) when queried, it returns $\phi_\kappa(x)$ and increments x. Moreover, let D be the distribution that picks $\kappa \in \{0,1\}^\lambda$ at random and outputs $(\widetilde{\mathsf{C}}, a = \kappa)$.

Note that the distribution D is specious, as the key κ is sampled at random *after* T has been quantified. In particular, the outputs of ϕ_κ are indistinguishable from random to T. The distribution D is also clearly signaling, as it can be seen by taking the adversary A that queries its target oracle twice and sends the outputs y_1 and y_2 as a signal to the decoder. The decoder Z, given $a = \kappa$, computes $x_i = \phi_\kappa^{-1}(y_i)$ (for $i = 1, 2$) and outputs 0 if and only if $x_2 = x_1 + 1$.

We can also define what it means for a set of subversions to be specious.

Definition 3 (Specious subversions). *Given an efficiently sampleable distribution* D *with outputs of the form* $(\widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m, a) \leftarrow \mathsf{D}$*, we let* D_i *be the distribution sampling* $(\widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m, a) \leftarrow \mathsf{D}$ *and then outputting* $(\widetilde{\mathsf{C}}_i, (i, a))$*. We say that* D *is specious if each* D_i *is specious.*

We now define the notion of a specious corruption. In this paper, we assume that all specious corruptions are static.

Definition 4 (Specious corruption). *We say that a party accepts specious corruptions if, whenever it gets input* $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *from the adversary, it replaces its code by* $\widetilde{\mathsf{C}}$. *If the input* $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *is not the first one received by the party, then it ignores it. We say that an environment* $\mathcal{E}$ *prepares specious corruptions if it operates as follows. First, it writes* $(\textsc{Specious}, \mathsf{D})$ *on a special tape, where* D *is specious. Then, it samples* $(\widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m, a) \leftarrow \mathsf{D}$ *and writes this on the special tape too. Finally, it inputs* $(\textsc{Specious}, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m)$ *to the adversary. The above has to be done on the first activation, before any other communication with protocols or the adversary. We call this a specious environment.*

In case of emulation with respect to the dummy adversary, we further require that if the environment instructs the dummy adversary to input $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *to a party, then* $\widetilde{\mathsf{C}}$ *is from the list in* $(\textsc{Specious}, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m)$. *We say that an adversary interacting with a specious environment does specious corruptions if whenever the adversary inputs* $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *to a party, then* $\widetilde{\mathsf{C}}$ *is from the list* $(\textsc{Specious}, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m)$ *received from the specious environment. We call such an adversary specious. In particular, an adversary which never inputs* $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *to any party is specious. We also call an environment specious if it does not write* $(\textsc{Specious}, \mathsf{D})$ *on a special tape as the first thing, but in this case we require that it does not input anything of the form* $(\textsc{Specious}, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_m)$ *to the adversary, and that it never instructs the dummy adversary to input* $(\textsc{Specious}, \widetilde{\mathsf{C}})$ *to any party.*

In addition we require that specious environments and adversaries only do static corruptions and that all corruptions are of the form.

- *Core* Malicious *and firewall* Malicious.
- *Core* Honest *and firewall* SemiHonest.
- *Core* Specious *and firewall* Honest.
- *Core* Honest *and firewall* Malicious.

We assume that all cores accept specious corruptions, and no other parties accept specious corruptions.

We add a few comments to the definition. First, let us explain why we only require security for the above four corruption patterns. Of all the corruption patterns shown in Table 1 giving rise to a Malicious party, the one with core Malicious and firewall Malicious gives the adversary strictly more power than any of the other ones, so we only ask for simulation of that case. Similarly, of the 3 corruption patterns giving rise to an Honest party, the ones with the core Honest and Specious and the firewall SemiHonest and Honest respectively are different, as neither gives powers to the adversary which are a subset of the other, so we ask for simulation of both. The remaining case of Honest core and Honest firewall we can drop, as it is a special case of the Honest core and SemiHonest firewall. The only corruption pattern giving rise to an Isolate party is when the core is Honest and the firewall is Malicious; we therefore ask to simulate this case too.

Core C	Firewall F	Party P in $\mathcal{F}$
Honest	Honest	Honest
Honest	SemiHonest	Honest
Specious	Honest	Honest
Honest	Malicious	Isolate
Specious	SemiHonest	Malicious
Specious	Malicious	Malicious
Malicious	Honest	Malicious
Malicious	SemiHonest	Malicious
Malicious	Malicious	Malicious

Table 1. Corruption patterns for cores and firewalls in our model, and their translation in the ideal world. The highlighted rows are the cases that one needs to consider when proving security using our framework.

Second, note that it might look odd that we ask the environment to sample the subversion $\widetilde{\mathsf{C}}_i$. Could we not just ask that, when it inputs (Specious, $\widetilde{\mathsf{C}}_i$) to a core, then $\widetilde{\mathsf{C}}_i$ is specious? It turns out that this would give a trivial notion of specious corruption. Recall that in the notion of specious, we quantify over all tests. If we first fix $\widetilde{\mathsf{C}}$, and then quantify over all tests when defining that it is specious, then the universal quantifier could be used to guess random values shared between $\widetilde{\mathsf{C}}$ and the adversary, like the key κ used in Lemma 1 (demonstrating that a specious subversion can still be signaling). Therefore, a single $\widetilde{\mathsf{C}}$ specious subversion cannot be signalling. Hence, asking for a specific subversion to be specious would make the notion of specious corruption trivial. By instead asking that a distribution D is specious, we can allow $\widetilde{\mathsf{C}}$ and the adversary to sample joint randomness (like a secret key κ) after the test T has already been quantified. Namely, recall that in the test game we first fix a T, and only then do we sample D. This allows specious corruptions which can still signal to the adversary, as demonstrated above. The reason why we ask the environment to sample D and not the adversary has to do with UC composition, which we return to later.

2.4 Sanitizing Protocols Implementing Regular Ideal Functionalities

For illustration, we first describe how to implement a regular ideal functionality given a sanitizing ideal functionality. Later, we cover the case of implementing a sanitizing ideal functionality given a sanitizing ideal functionality, see Fig. 1.

Consider a sanitizing protocol Π, using a sanitizable ideal functionality $\mathcal{G}$, that implements a regular ideal functionality $\mathcal{F}$ with n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. By regular, we mean that $\mathcal{F}$ itself does not have a sanitation interface. Note that it makes perfect sense for a sanitizing protocol Π, using a sanitizable ideal functionality $\mathcal{G}$, to implement a regular ideal functionality. The firewall is an aspect of the implementation Π and the sanitizable hybrid ideal functionality $\mathcal{G}$. In particular, this aspect could be completely hidden by the implementation of Π. However, typically the behavior when the firewall is honest and corrupted is not the same.

A corrupted firewall can isolate the core by not doing its job. We therefore call a party P_i where C_i is honest and F_i is corrupt an "isolated" party. We insist that if C_i is specious and F_i is honest, then it is as if P_i is honest. Hence, $\mathcal{F}$ should behave as if P_i is honest. We would therefore like the behavior of $\mathcal{F}$ to depend only on whether P_i is honest, isolated, or corrupt. To add some structure to this, we introduce the notion of a wrapped ideal functionality and a wrapper.

A wrapped ideal functionality $\mathcal{F}$ should only talk to parties P_i. The wrapper Wrap will talk to a core C_i and a firewall F_i. The wrapper runs $\mathcal{F}$ internally, and we write $\mathsf{Wrap}(\mathcal{F})$. The inputs to and from C_i on $\mathsf{Wrap}(\mathcal{F})$ are forwarded to the interface for P_i on $\mathcal{F}$. The only job of Wrap is to introduce the same parties as in the protocol and translate corruptions of C_i and F_i into corruptions on P_i. We say that parties P_i in an ideal execution with $\mathcal{F}$ can be HONEST, MALICIOUS or ISOLATE. The wrapped ideal functionality $\mathsf{Wrap}(\mathcal{F})$ translates corruptions using the following *standard corruption translation table.*

Honest: If C_i is HONEST and F_i HONEST, let P_i be HONEST on $\mathcal{F}$.
Malicious: If C_i is MALICIOUS, corrupt P_i as MALICIOUS on $\mathcal{F}$.
Isolated: If C_i is HONEST and F_i is MALICIOUS, corrupt P_i as ISOLATE on $\mathcal{F}$.
Sanitation: If C_i is SPECIOUS and F_i is HONEST, let P_i be HONEST on $\mathcal{F}$.
No Secrets: If C_i is HONEST and F_i is SEMIHONEST, let P_i be HONEST on $\mathcal{F}$.

We discuss the five cases next. The **Honest** and **Malicious** cases are straightforward; if both the core and the firewall are honest, then treat P_i as an honest party on $\mathcal{F}$. Similarly, if the core is malicious, then treat P_i as a malicious party on $\mathcal{F}$. The **Isolated** case corresponds to the situation where the core is honest and the firewall is corrupted, and thus the firewall is isolating the core from the network. This will typically correspond to a corrupted party. However, in some cases, some partial security might be obtainable, like the inputs of the core being kept secret. We therefore allow an ISOLATE corruption as an explicit type of corruption. The standard behavior of $\mathcal{F}$ on an ISOLATE corruption is to do a MALICIOUS corruption of P_i in $\mathcal{F}$.

The **Sanitation** case essentially says that the job of the firewall is to turn a specious core into an honest core. This, in particular, means that the firewall should remove any signaling. We add the **No Secrets** case to avoid trivial solutions where the firewall is keeping, *e.g.*, secret keys used in the protocol. We want secret keys to reside in the core, and that firewalls only sanitize communication of the core. We also do not want that the core just hands the inputs to the firewall and lets it run the protocol. A simple way to model this is to require that the protocol should tolerate a semi-honest corruption of the firewall when the core is honest. We do not require that we can tolerate a specious core and a semi-honest firewall. Removing signaling from a core will typically require randomizing some of the communication. For this, the firewall needs to be able to make secret random choices. Note that, with this modeling, a core and a firewall can be seen as a two-party implementation of the honest party, where one can tolerate either a specious corruption of the core or a semi-honest corruption of the firewall.

Definition 5 (Wrapped subversion-resilient UC security). *Let $\mathcal{F}$ be an ideal functionality for n parties $\mathsf{P}_1, \dots, \mathsf{P}_n$. Let Π be a sanitizing protocol with*

n cores $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and n firewalls $\mathsf{F}_1, \ldots, \mathsf{F}_n$. Let $\mathcal{G}$ be a sanitizable ideal functionality which can be used by Π as in Fig. 1. We say that Π wsrUC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if Π UC-realizes $\mathsf{Wrap}(\mathcal{F})$ in the $\mathcal{G}$-hybrid model with the restriction that we only quantify over specious environments and specious adversaries.

The typical behavior of a sanitizing ideal functionality is that, when it receives a message from the core, it will output the received message to the firewall, or output some partial information about the message to the firewall. Later, it will receive some new message or sanitation instruction from the firewall. Given this, it constructs the actual information to pass to the core functionality of $\mathcal{G}$. This might later end up at a firewall of another party, and after sanitation end up at the core of that party. The latter is illustrated in Fig. 1, and an example is given below. Note that this is not a formal requirement, but just a description of idiomatic use of sanitation to give an intuition on the use of the model.

To illustrate the use of sanitizable ideal functionalities, we specify an ideal functionality $\mathcal{F}_{\mathsf{SAT}}$ for sanitizable authenticated communication. The communication between cores goes via the firewall which might change the messages. Note that firewalls can be sure which other firewall they talk to, but corrupted firewalls can lie to their local core about who sent a message. In fact, they can pretend a message arrived out of the blue. We also equip $\mathcal{F}_{\mathsf{SAT}}$ with the possibility for distributing setup, as this is needed in some of our protocols. We assume a setup generator Setup which samples the setup and gives each party their corresponding value. The firewalls also get a value. This, *e.g.*, allows to assume that the firewalls know a CRS. Since we do not want firewalls to keep secrets, we leak their setup values to the adversary. This would not be a problem if the setup values is a CRS.

Functionality $\mathcal{F}_{\mathsf{SAT}}$

- Initially sample $((v_1, w_1), \ldots, (v_n, w_n)) \leftarrow \mathsf{Setup}()$ and output v_i on IO_i and w_i on S_i. Leak w_i to the adversary.
- On input $(\textsc{Send}, a, \mathsf{P}_j)$ on IO_i, output $(\textsc{Send}, a, \mathsf{P}_j)$ on S_i. To keep the description simple we assume honest parties sends the same a at most once. Adding fresh message identifiers can be used for this in an implementation.
- On input $(\textsc{Send}, b, \mathsf{P}_k)$ on S_i, leak $(\textsc{Send}, \mathsf{P}_i, b, \mathsf{P}_k)$ to the adversary and store $(\textsc{Send}, \mathsf{P}_i, b, \mathsf{P}_k)$.
- On input $(\textsc{Deliver}, (\textsc{Send}, \mathsf{P}_i, b, \mathsf{P}_k))$ from the adversary, where $(\textsc{Send}, \mathsf{P}_i, b, \mathsf{P}_k)$ is stored, delete this tuple and output $(\textsc{Receive}, \mathsf{P}_i, b)$ on S_k.
- On input $(\textsc{Receive}, \mathsf{P}_m, c)$ on S_k, output $(\textsc{Receive}, \mathsf{P}_m, c)$ on IO_i.

Remark 1 (on $\mathcal{F}_{\mathsf{SAT}}$). We note that all protocols in this work, even if not explicitly stated, are described in the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model. Moreover, whenever we say that the core sends a message to the firewall (or vice-versa) we actually mean that they communicate using $\mathcal{F}_{\mathsf{SAT}}$.

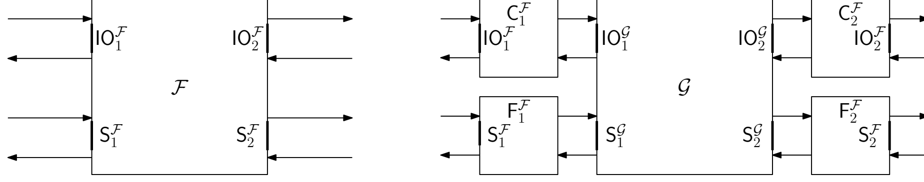

Figure 2. Implementing $\mathcal{F}$ via protocol $\Pi = (\mathsf{C}^{\mathcal{F}}, \mathsf{F}^{\mathcal{F}})$ using $\mathcal{G}$.

2.5 General Case

We now turn our attention to implementing sanitizable ideal functionalities. When a protocol Π implements a sanitizable ideal functionality, we call Π a sanitizable protocol. Notice the crucial difference between being a *sanitizable* protocol and a *sanitizing* protocol. A sanitizable protocol Π *implements* the sanitization interface S_i of $\mathcal{F}$. Whereas a sanitizing protocol Π would have a firewall *using* the sanitization interface S_i of $\mathcal{G}$.

When implementing a sanitizable ideal functionality $\mathcal{F}$, the protocol should implement the sanitation interface $\mathsf{S}^{\mathcal{F}}$ for F. This means that the protocol will be of the form $\Pi = (\mathsf{IO}, \mathsf{S})$ where $\mathsf{IO} = (\mathsf{IO}_1, \ldots, \mathsf{IO}_n)$ and $\mathsf{S} = (\mathsf{S}_1, \ldots, \mathsf{S}_n)$. Notice that C_i and F_i formally are separate parties, so they cannot talk directly.

It is natural that it is the firewall of the implementation $\Pi = (\mathsf{IO}, \mathsf{S})$ which handles this. The firewall has access to the sanitation interface of $\mathcal{G}$, which it can use to sanitize Π. This means that F gets what could look like a double role now. First, it sanitizes Π using $\mathsf{S}^{\mathcal{G}}$. Second, it has to implement the sanitation interface $\mathsf{S}^{\mathcal{F}}$ of Π (matching that of $\mathcal{F}$). Note, however, that this is in fact the same job. The sanitation interface $\mathsf{S}^{\mathcal{F}}$ of Π is used to specify how Π should be sanitized. It is natural that $\mathsf{F}^{\mathcal{F}}$ needs to knows this specification. It then uses $\mathsf{S}^{\mathcal{G}}$ to implement the desired sanitation. This is illustrated in Fig. 2.

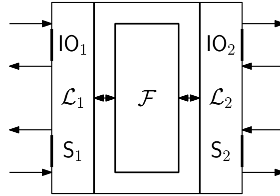

Figure 3. The wrapper $\mathsf{Wrap}(\mathcal{F}, \mathcal{L}_1, \ldots, \mathcal{L}_n)$).

When defining security of a protocol implementing a sanitizable ideal functionality, we do not need to use a wrapper as when implementing a normal ideal functionality, as $\mathcal{F}$ already has the same parties as in the protocol. It is however

still convenient to use a wrapper to add some structure to how we specify a sanitizable ideal functionality. We assume a central part which does the actual computation, and outer parts which sanitize the inputs from P_i before they are passed to the central part.

Definition 6 (Well-formed sanitizing ideal functionality). *A well-formed sanitizing ideal functionality consists of an ideal functionality $\mathcal{F}$, called the central part, with an interface P_i for each party. The interface P_i can be* Honest*,* Malicious*, or* Isolate*. There are also n outer parts $\mathcal{L}_1, \ldots, \mathcal{L}_n$ where $\mathcal{L}_i$ has an interface IO_i for the core and S_i for the firewall. The outer part $\mathcal{L}_i$ can only talk to the central part on P_i and the outer parts cannot communicate with each other. The interface IO_i can be* Honest*,* Malicious*, or* Specious*. The interface S_i can be* Honest*,* Malicious*, or* SemiHonest*. The corruption of $\mathcal{F}.\mathsf{IO}_i$ is computed from that of $\mathcal{L}_i.\mathsf{IO}_i$ and $\mathcal{L}_i.\mathsf{S}_i$ using the standard corruption translation table.*

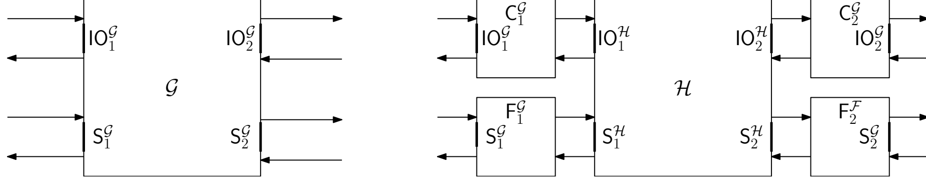

Figure 4. Implementing $\mathcal{G}$ via protocol $\Gamma = (\mathsf{C}^{\mathcal{G}}, \mathsf{F}^{\mathcal{G}})$ using $\mathcal{H}$.

Definition 7 (Subversion-resilient UC security). *Let $\mathcal{F}$ be an ideal functionality for n cores $\mathsf{C}_1^{\mathcal{F}}, \ldots, \mathsf{C}_n^{\mathcal{F}}$ and n firewalls $\mathsf{F}_1^{\mathcal{F}}, \ldots, \mathsf{F}_n^{\mathcal{F}}$, and let Π be a sanitizing protocol with n cores $\mathsf{C}_1^{\mathcal{F}}, \ldots, \mathsf{C}_n^{\mathcal{F}}$ and n firewalls $\mathsf{F}_1^{\mathcal{F}}, \ldots, \mathsf{F}_n^{\mathcal{F}}$. Let $\mathcal{G}$ be a sanitizable ideal functionality which can be used by Π as in Fig. 2. We say that Π srUC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if $\mathcal{F}$ can be written as a well-formed sanitizing ideal functionality, and Π UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with the restriction that we only quantify over specious environments and specious adversaries.*

2.6 Composition

We now address composition. In Fig. 2, we illustrate implementing $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Similarly, in Fig. 4, we implement $\mathcal{G}$ given $\mathcal{H}$. In Fig. 5, we illustrate the effect of composition. We can let $\mathsf{C}_i = \mathsf{C}_i^{\mathcal{F}} \circ \mathsf{C}_i^{\mathcal{G}}$ and $\mathsf{F}_i = \mathsf{F}_i^{\mathcal{F}} \circ \mathsf{F}_i^{\mathcal{G}}$. Then, we again have a sanitizing protocol $\Pi^{\mathcal{G} \to \Gamma} = (\mathsf{C}, \mathsf{F})$. For composition to work, we need that specious corruptions respect the composition of a core.

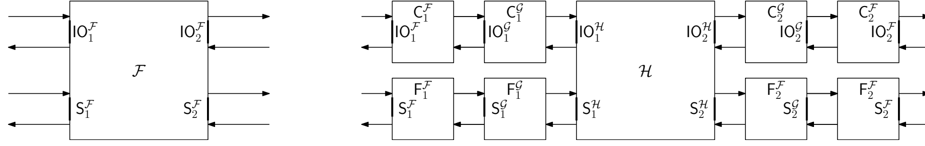

Figure 5. Implementing $\mathcal{F}$ via protocol $\Pi^{\mathcal{G}\rightarrow\Gamma}$ using $\mathcal{H}$.

Definition 8 (Specious corruption of a composed core). *We say that an adversary does a specious corruption of a composed core $\mathsf{C}_i = \mathsf{C}_i^{\mathcal{F}} \circ \mathsf{C}_i^{\mathcal{G}}$ if it inputs* $(\text{Specious}, \widetilde{\mathsf{C}}_i^{\mathcal{F}}, \widetilde{\mathsf{C}}_i^{\mathcal{G}})$*, where both $\mathsf{C}_i^{\mathcal{F}}$ and $\mathsf{C}_i^{\mathcal{G}}$ are specious. In response $\mathsf{C}_i^{\mathcal{F}}$ replaces its code with $\widetilde{\mathsf{C}}_i^{\mathcal{F}}$, and $\mathsf{C}_i^{\mathcal{G}}$ replaces its code with $\widetilde{\mathsf{C}}_i^{\mathcal{G}}$.*

Note that one could imagine a specious corruption of a composed core $\widetilde{\mathsf{C}}_i$ which could not be written as the composition of specious subversions $\widetilde{\mathsf{C}}_i^{\mathcal{F}}$ and $\widetilde{\mathsf{C}}_i^{\mathcal{G}}$.

Theorem 1 (srUC Composition). *Let $\mathcal{F}$ and $\mathcal{G}$ be ideal functionalities, and let Π and Γ be protocols. Assume that all are subroutine respecting and subroutine exposing as defined in [6]. If Π srUC-realizes $\mathcal{F}$, and Γ srUC-realizes $\mathcal{G}$, then $\Pi^{\mathcal{G}\rightarrow\Gamma}$ srUC-realizes $\mathcal{F}$.*

The proof of Theorem 1 appears in the full version [12].

Note that if, *e.g.*, $\mathcal{G}$ in the composition is well-formed and therefore wrapped, then it is the wrapped functionality which is considered at all places. Therefore, in Fig. 4 the ideal functionality $\mathcal{G}$ being implemented will be the wrapped ideal functionality, and in Fig. 2 the hybrid ideal functionality $\mathcal{G}$ being used would again be the wrapped one. There is no notion of "opening up the wrapping" during composition. If $\mathcal{F}$ is a regular ideal functionality then $\mathsf{Wrap}(\mathcal{F})$ can be written as a well-formed sanitizing ideal functionality. Therefore wsrUC security relative to $\mathcal{F}$ implies srUC security relative to $\mathsf{Wrap}(\mathcal{F})$. During composition it would be $\mathsf{Wrap}(\mathcal{F})$ which is used as a hybrid functionality. This is basically the same as having $\mathcal{F}$ under the standard corruption translation.

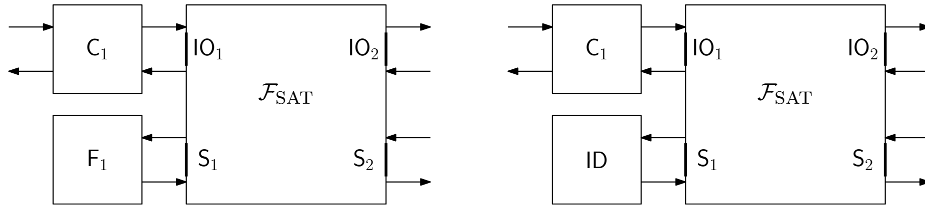

Figure 6. A core with its matching firewall or with the identity firewall.

2.7 Computational Transparency

A central notion in the study of reverse firewalls is the notion of transparency. The firewall is only supposed to modify the behavior of a subverted core. If the firewall is attached to an honest core, it must not change the behavior of the core. We define transparency in line with [21], namely, an honest core without a firewall attached should be indistinguishable from an honest core with a firewall attached.

Notice that this does not make sense if the party is implementing a sanitizable ideal functionality, like in Fig. 2. Without a firewall $\mathsf{F}_1^{\mathcal{F}}$, no entity would implement the interface $\mathsf{S}_1^{\mathcal{F}}$, which would make a core without a firewall trivially distinguishishable from a core with a firewall. Presumably, the interface $\mathsf{S}_1^{\mathcal{F}}$ is present because different inputs on this interface will give different behaviors. We therefore only define transparency of firewalls implementing a regular ideal functionality, as in Fig. 1. Note also that if $\mathcal{G}$ in Fig. 1 has a complex interaction with F_i, then an execution without F_i might not make sense. Therefore, we additionally only consider transparency in the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model. In this model we can let F_i be an *identity firewall* which does not modify the communication. This has the desired notion of no firewall being present.

Definition 9 (Transparency). *Let* $(\mathsf{C}_i, \mathsf{F}_i)$ *be a party for the* $\mathcal{F}_{\mathsf{SAT}}$*-hybrid model. Let* Π_i *be the protocol for the* $\mathcal{F}_{\mathsf{SAT}}$*-hybrid model where party number* i *is* $(\mathsf{C}_i, \mathsf{F}_i)$*, and all other parties are dummy parties. Let* ID *be the firewall which always outputs any message it receives as input. Let* Π_i' *be the protocol for the* $\mathcal{F}_{\mathsf{SAT}}$*-hybrid model where party number* i *is* $(\mathsf{C}_i, \mathsf{ID})$*, and all other parties are dummy parties. These two protocols are illustrated in Fig. 6. We say that* F_i *is* computationally transparent *if, for all poly-time environments* $\mathcal{E}$ *which do not corrupt* C_i *or* F_i/ID*, it holds that* $\textsc{Exec}_{\mathcal{E},\Pi_i,\mathcal{A}} \approx \textsc{Exec}_{\mathcal{E},\Pi_i',\mathcal{A}}$*, where* $\mathcal{A}$ *is the dummy adversary.*

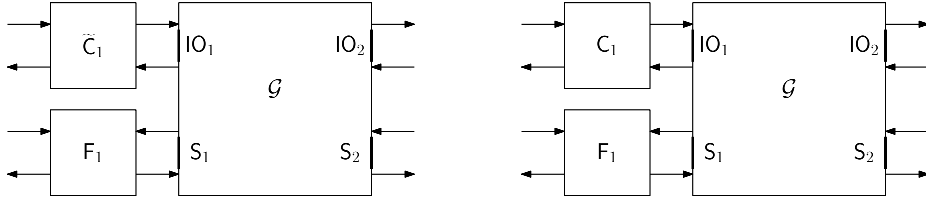

Figure 7. An honest core with its matching firewall or a specious core with the same firewall.

2.8 Strong Sanitation

Another central notion in the study of reverse firewalls is the notion that we call sanitation. Namely, if you hide a specious core behind a firewall, then it looks

like an honest core behind a firewall. So far, we have defined this implicitly by saying that a specious corruption of a core plus an honest firewall should be simulatable by having access to an honest party on the ideal functionality being implemented. This actually does not imply that the network cannot distinguish between a specious core or an honest core behind the firewall. It only says that the effect of a specious core behind a firewall are not dire enough that you cannot simulate given an honest party in the ideal world.

In this section, we give a game-based definition of sanitation capturing the stronger notion that, behind a firewall, a specious core looks like an honest core. Recall that a core C_i is capable of receiving a specious corruption $(\textsc{Specious}, \widetilde{\mathsf{C}})$ from the environment, in which case it replaces its code by $\widetilde{\mathsf{C}}$. For such a core, let $\widehat{\mathsf{C}}$ be the *incorruptible core* which when it receives a specious corruption $(\textsc{Specious}, \widetilde{\mathsf{C}})$ will ignore it and keep running the code of C.

Definition 10 (Strong sanitation). *Let $(\mathsf{C}_i, \mathsf{F}_i)$ by a party for the $\mathcal{G}$-hybrid model. Let $\widehat{\mathsf{C}}_i$ be the corresponding incorruptible core. Let Π_i be the protocol for the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model where party number i is $(\mathsf{C}_i, \mathsf{F}_i)$, and all other parties are dummy parties. Let Π_i' be the protocol for the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model where party number i is $(\widehat{\mathsf{C}}_i, \mathsf{F}_i)$, and all other parties are dummy parties. Note that if the environment does a $(\textsc{Specious}, \widetilde{\mathsf{C}})$ corruption of core number i, then in Π_i core i will run $\widetilde{\mathsf{C}}$, whereas in Π_i' it will run C_i. These two outcomes are illustrated in Fig. 7. We say that F_i is* strongly sanitising *if, for all poly-time environments $\mathcal{E}$ which do not corrupt F_i, but which are allowed a specious corruption of the core, it holds that $\textsc{Exec}_{\mathcal{E},\Pi_i,\mathcal{A}} \approx \textsc{Exec}_{\mathcal{E},\Pi_i',\mathcal{A}}$, where $\mathcal{A}$ is the dummy adversary.*

It is easy to see that the definition is equivalent to requiring that, for all poly-time environments $\mathcal{E}$ which do not corrupt C_i or F_i/ID, it holds that $\textsc{Exec}_{\mathcal{E},\Pi_i,\mathcal{A}} \approx \textsc{Exec}_{\mathcal{E},\Pi_i',\mathcal{A}}$, where $\mathcal{A}$ is the dummy adversary.

Lemma 2. *Consider a protocol Π where for all parties $(\mathsf{C}_i, \mathsf{F}_i)$ it holds that F_i has strong sanitation. Then it is enough to prove security for these cases:*

- *Core* Malicious *and firewall* Malicious*.*
- *Core* Honest *and firewall* SemiHonest*.*
- *Core* Honest *and firewall* Malicious*.*

If in addition we assume the standard corruption behavior for Isolate*, it is enough to prove the cases:*

- *Core* Malicious *and firewall* Malicious*.*
- *Core* Honest *and firewall* SemiHonest*.*

If in addition the protocol Π is for the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model and has computational transparency, then it is enough to prove the case:

- *Core* Malicious *and firewall* Malicious*.*
- *Core* Honest *and firewall* Honest*.*

Proof. We prove the first claim. Note that relative to Definition 7 we removed the case with the core SPECIOUS and the firewall HONEST. We show that this reduces to the case with core HONEST and the firewall HONEST. First replace each C_i by $\widehat{\mathsf{C}}_i$. This cannot be noticed due to strong sanitation. Then notice that we can replace an environment $\mathcal{E}$ doing specious corruption by $\mathcal{E}'$ which just internally do not pass on $(\text{SPECIOUS}, \widetilde{\mathsf{C}})$ to the core. Namely, it does not matter if $\widehat{\mathsf{C}}_i$ ignores the commands or we let $\mathcal{E}'$ do it. Then, we can replace $\widehat{\mathsf{C}}_i$ by C_i as there are no commands to ignore. So it is enough to prove security for the core HONEST and the firewall HONEST. This case follows from the case with the core SPECIOUS and the firewall HONEST as being honest is a special case of being specious.

The second claim follows from the fact that under standard corruption behavior for ISOLATE the party P_i on the ideal functionality is MALICIOUS when the firewall is MALICIOUS. So the simulator has the same power when simulating an honest core and malicious firewall as when simulating a malicious core and a malicious firewall. Then note that being an honest core is a special case of being a malicious core.

In the last claim, we have to prove that assuming computational transparency one does not have to prove the case with the core HONEST and the firewall SEMIHONEST. One can instead prove the case with the core HONEST and the firewall HONEST. To see this note that, by definition of transparency, we can replace the firewall with the identity firewall ID. For this firewall, an HONEST corruption is as powerful as a SEMIHONEST corruption. This is because the only effect of a semi-honest corruption of ID is to leak the internal value w_i from the setup and the communication sent via ID. The ideal functionality $\mathcal{F}_{\mathsf{SAT}}$ already leaks that information when ID is honest.

3 String Commitment

In this section, we show how to build UC string commitments with security in the presence of subversion attacks. In particular, after introducing the sanitizable commitment functionality, we exhibit a non-interactive commitment (with an associated reverse firewall) that UC realizes this functionality in the CRS model, under the DDH assumption.

3.1 Sanitizable Commitment Functionality

The sanitazable commitment functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$, which is depicted below, is an extension of the standard functionality for UC commitments [8]. Roughly, $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ allows the core of a party to commit to a λ-bit string s_i; the ideal functionality stores s_i and informs the corresponding firewall that the core has sent a commitment. Hence, via the sanitation interface, the firewall of that party is allowed to forward to the functionality a blinding factor $r_i \in \{0,1\}^\lambda$ that is used to blind s_i, yielding a sanitized input $\hat{s}_i = s_i \oplus r_i$. At this point, all other parties are informed by the functionality that a commitment took place. Finally, each party

is allowed to open the commitment via the functionality, in which case all other parties learn the sanitized input $\hat{s}_i$.

Functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$

The sanitizable string commitment functionality $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ runs with parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ (each consisting of a core C_i and a firewall F_i), and an adversary $\mathcal{S}$. The functionality consists of the following communication interfaces for the cores and the firewalls respectively.

Interface IO

- Upon receiving a message $(\textsc{Commit}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, s_i)$ from C_i, where $s_i \in \{0,1\}^\lambda$, record the tuple $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, s_i)$ and send the message $(\textsc{Receipt}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i)$ to F_i. Ignore subsequent commands of the form $(\textsc{Commit}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$.
- Upon receiving a message $(\textsc{Open}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i)$ from C_i, proceed as follows: If the tuple $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \hat{s}_i)$ is recorded and the message $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$ was sent to $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$, then send the message $(\textsc{Open}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \hat{s}_i)$ to all $\mathsf{C}_{j \neq i}$ and $\mathcal{S}$. Otherwise, do nothing.

Interface S

- Upon receiving a message $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, r_i)$ from F_i, where $r_i \in \{0,1\}^\lambda$, proceed as follows: If the tuple $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, s_i, \cdot)$ is recorded, then modify the tuple to be $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \hat{s}_i = s_i \oplus r_i)$ and send the message $(\textsc{Blinded}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, r_i)$ to C_i, and $(\textsc{Receipt}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i)$ to all $\mathsf{C}_{j \neq i}$ and $\mathcal{S}$; otherwise do nothing. Ignore future commands of the form $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$.

3.2 Protocol from DDH

Next, we present a protocol that UC-realizes $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ in the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model. For simplicity, let us first consider the case where there are only two parties. The CRS in our protocol is a tuple $\mathsf{crs} = (g, h, T_1, T_2)$ satisfying the following properties:

- The element g is a generator of a cyclic group $\mathbb{G}$ with prime order q, and $h, T_1, T_2 \in \mathbb{G}$. Moreover, the DDH assumption holds in $\mathbb{G}$.[5]
- In the real-world protocol, the tuple (g, h, T_1, T_2) corresponds to a non-DH tuple. Namely, it should be the case that $T_1 = g^x$ and $T_2 = h^{x'}$, for $x \neq x'$.
- In the security proof, the simulator will set the CRS as (g, h, T_1, T_2), where $T_1 = g^x$ and $T_2 = h^x$. By the DDH assumption, this distribution is computationally indistinguishable from the real-world distribution. In addition, the simulator will be given the trapdoor (x, t) for the CRS $\mathsf{crs} = (g, h, T_1, T_2)$, such that $h = g^t$ and $T_1 = g^x$.

As explained in Section 1.3, the above ideas can be generalized to the multi-party setting by using a different CRS for each pair of parties.

[5] Recall that the DDH assumption states that the distribution ensembles $\{g, h, g^x, h^x : x \leftarrow \mathbb{Z}_q\}$ and $\{g, h, g^x, h^{x'} : x, x' \leftarrow \mathbb{Z}_q\}$ are computationally indistinguishable.

Protocol $\widehat{\Pi}_{\mathsf{sCOM}}$ (Sanitizable UC Commitment Protocol)

The protocol is executed between parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ each consisting of a core C_i and a firewall F_i. In what follows, let party $\mathsf{P}_j = (\mathsf{C}_j, \mathsf{F}_j)$ be the committer, and all other parties $\mathsf{P}_{k \neq j}$ act as verifiers.

Public inputs: Group $\mathbb{G}$ with a generator g, field $\mathbb{Z}_q$, and $\mathsf{crs} = (\mathsf{crs}_{j,k})_{j,k \in [n], k \neq j} = (g_{j,k}, h_{j,k}, T_{1,j,k}, T_{2,j,k})_{j,k \in [n], k \neq j}$.
Private inputs: The committer (or core) C_j has an input $s \in \{0,1\}^\lambda$ which we parse as $s = (s[1], \cdots, s[\lambda])$. We will encode each bit $s[i] \in \{0,1\}$ with a value $u[i] \in \{-1, 1\}$, so that $u[i] = 1$ if $s[i] = 1$ and $u[i] = -1$ if $s[i] = 0$. The firewall F_j has an input $r = (r[1], \cdots, r[\lambda]) \in \{0,1\}^\lambda$ (*i.e.*, the blinding factor).
Commit phase: For all $i \in [\lambda]$, the core C_j samples a random $\alpha_{j,k}[i] \leftarrow \mathbb{Z}_q$ and computes the values $B_{j,k}[i] = g_{j,k}^{\alpha_{j,k}[i]} \cdot T_{1,j,k}^{u[i]}$ and $H_{j,k}[i] = h_{j,k}^{\alpha_{j,k}[i]} \cdot T_{2,j,k}^{u[i]}$. Hence, it sends $c_{j,k} = (c_{j,k}[1], \cdots, c_{j,k}[\lambda])$ to the firewall F_j where $c_{j,k}[i] = (B_{j,k}[i], H_{j,k}[i])$. For all $i \in [\lambda]$, the firewall F_j picks random $\beta_{j,k} = (\beta_{j,k}[1], \cdots, \beta_{j,k}[\lambda]) \in \mathbb{Z}_q^\lambda$ and does the following:
– If $r[i] = 0$, it lets $\widehat{B}_{j,k}[i] = B_{j,k}[i] \cdot g_{j,k}^{\beta_{j,k}[i]}$ and $\widehat{H}_{j,k}[i] = H_{j,k}[i] \cdot h_{j,k}^{\beta_{j,k}[i]}$;
– Else if $r[i] = 1$, it lets $\widehat{B}_{j,k}[i] = B_{j,k}[i]^{-1} \cdot g_{j,k}^{\beta_{j,k}[i]}$ and $\widehat{H}_{j,k}[i] = H_{j,k}[i]^{-1} \cdot h_{j,k}^{\beta_{j,k}[i]}$.
Hence, F_j sends $\hat{c}_{j,k} = (\hat{c}_{j,k}[1], \cdots, \hat{c}_{j,k}[\lambda])$ to all other parties $\mathsf{P}_{k \neq j}$, where $\hat{c}_{j,k}[i] = (\widehat{B}_{j,k}[i], \widehat{H}_{j,k}[i])$.
Opening phase: The core C_j sends $(s, \alpha_{j,k})$ to the firewall F_j, where $s \in \{0,1\}^\lambda$ and $\alpha_{j,k} \in \mathbb{Z}_q^\lambda$. Upon receiving $(s, \alpha_{j,k})$ from C_j, the firewall F_j parses $s = (s[1], \cdots, s[\lambda])$ and $\alpha_{j,k} = (\alpha_{j,k}[1], \cdots, \alpha_{j,k}[\lambda])$. Thus, for all $i \in [\lambda]$, it does the following:
– If $r[i] = 0$, it lets $\hat{s}[i] = s[i]$ and $\hat{\alpha}_{j,k}[i] = \alpha_{j,k}[i] + \beta_{j,k}[i]$;
– Else if $r[i] = 1$, it lets $\hat{s}[i] = -s[i]$ and $\hat{\alpha}_{j,k}[i] = -\alpha_{j,k}[i] + \beta_{j,k}[i]$.
Hence, F_j sends $(\hat{s}, \hat{\alpha}_{j,k})$ to all other parties $\mathsf{P}_{k \neq j}$, where $\hat{s} = (\hat{s}[1], \cdots, \hat{s}[\lambda])$ and $\hat{\alpha}_{j,k} = (\hat{\alpha}_{j,k}[1], \cdots, \hat{\alpha}_{j,k}[\lambda])$.
Verification phase: Upon receiving $(\hat{c}_{j,k}, (\hat{s}, \hat{\alpha}_{j,k}))$ from P_j, each party $\mathsf{P}_{k \neq j}$ parses $\hat{c}_{j,k} = ((\widehat{B}_{j,k}[1], \widehat{H}_{j,k}[1]), \cdots, (\widehat{B}_{j,k}[\lambda], \widehat{H}_{j,k}[\lambda]))$, $\hat{\alpha}_{j,k} = (\hat{\alpha}_{j,k}[1], \cdots, \hat{\alpha}_{j,k}[\lambda])$, and encodes $\hat{s} = (\hat{s}[1], \cdots, \hat{s}[\lambda]) \in \{0,1\}^\lambda$ as $\hat{u} = (\hat{u}[1], \cdots, \hat{u}[\lambda]) \in \{-1, 1\}^\lambda$. Hence, for all $i \in [\lambda]$, it verifies whether $\widehat{B}_{j,k}[i] = g_{j,k}^{\hat{\alpha}_{j,k}[i]} \cdot T_{1,j,k}^{\hat{u}[i]}$ and $\widehat{H}_{j,k}[i] = h_{j,k}^{\hat{\alpha}_{j,k}[i]} \cdot T_{2,j,k}^{\hat{u}[i]}$. If for any $i \in [\lambda]$, the above verification fails, party P_k aborts; otherwise P_k accepts the commitment.

Theorem 2. *The protocol $\widehat{\Pi}_{\mathsf{sCOM}}$ srUC-realizes the $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ functionality in the $\mathcal{F}_{\mathsf{SAT}}$-hybrid model in the presence of up to $n-1$ static malicious corruptions.*

We defer the proof of Theorem 2 to the full version [12].

4 Coin Tossing

In this section, we build a sanitizing protocol that implements the regular coin tossing functionality. Our protocol is described in the $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$-hybrid model, and therefore must implement the firewall that interacts with the $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$ functionality.

4.1 The Coin Tossing Functionality

We start by recalling the regular $\mathcal{F}_{\mathsf{TOSS}}$ functionality below. Intuitively, the functionality waits to receive an initialization message from all the parties. Hence, it samples a uniformly random λ-bit string s and sends s to the adversary. The adversary now can decide to deliver s to a subset of the parties. The latter restriction comes from the fact that it is impossible to toss a coin fairly so that no adversary can cause a premature abort, or bias the outcome, without assuming honest majority [14].

Functionality $\mathcal{F}_{\mathsf{TOSS}}$

The coin tossing functionality $\mathcal{F}_{\mathsf{TOSS}}$ runs with parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$, and an adversary $\mathcal{S}$. It consists of the following communication interface.

- Upon receiving a message $(\textsc{Init}, \mathsf{sid}, \mathsf{P}_i)$ from P_i: If this is the first such message from P_i then record $(\mathsf{sid}, \mathsf{P}_i)$ and send $(\textsc{Init}, \mathsf{P}_i)$ to $\mathcal{S}$. If there exist records $(\mathsf{sid}, \mathsf{P}_j)$ for all $(\mathsf{P}_j)_{j\in[n]}$, then sample a uniformly random bit string $s \in \{0,1\}^\lambda$ and send s to the adversary $\mathcal{S}$.
- Upon receiving a message $(\textsc{Deliver}, \mathsf{sid}, \mathsf{P}_i)$ from $\mathcal{S}$ (and if this is the first such message from $\mathcal{S}$), and if there exist records $(\mathsf{sid}, \mathsf{P}_j)$ for all $(\mathsf{P}_j)_{j\in[n]}$, send s to P_i; otherwise do nothing.

4.2 Sanitizing Blum's Protocol

Next, we show how to sanitize a variation of the classical Blum coin tossing protocol [5]. In this protocol, each party commits to a random string s_i and later opens the commitment, thus yielding $s = s_1 \oplus \cdots \oplus s_n$. The firewall here samples an independent random string r_i which is used to blind the string s_i chosen by the (possibly specious) core. We defer the security proof to the full version [12].

Protocol $\widehat{\Pi}_{\mathsf{TOSS}}$ (Sanitizing Blum's Coin Tossing)

The protocol is described in the $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$-hybrid model, and is executed between parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ each consisting of a core C_i and a firewall F_i. Party $\mathsf{P}_i = (\mathsf{C}_i, \mathsf{F}_i)$ proceeds as follows (the code for all other parties is analogous).

1. The core C_i samples a random string $s_i \in \{0,1\}^\lambda$ and sends $(\textsc{Commit}, \mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i, s_i)$ to $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$.
2. Upon receiving $(\textsc{Receipt}, \mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i)$ from $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$, the firewall F_i samples a random string $r_i \in \{0,1\}^\lambda$ and sends $(\textsc{Blind}, \mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i, r_i)$ to $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$.
3. Upon receiving $(\textsc{Blinded}, \mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i, r_i)$ from $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$, as well as $(\textsc{Receipt}, \mathsf{sid}_j, \mathsf{cid}_j, \mathsf{C}_j)$ for all other cores $\mathsf{C}_{j\neq i}$, the core C_i sends the message $(\textsc{Open}, \mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i)$ to $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$.

4. Upon receiving $(\textsc{Open}, \mathsf{sid}_j, \mathsf{cid}_j, \mathsf{C}_j, \hat{s}_j)$ from $\widehat{\mathcal{F}}_{\mathsf{sCOM}}$, for each core $\mathsf{C}_{j \neq i}$, the core C_i outputs $s := s_i \oplus r_i \oplus \bigoplus_{j \neq i} \hat{s}_j$. (If any of the cores C_j do not open its commitment, then C_i sets $\hat{s}_j = 0^\lambda$.)

Theorem 3. *The protocol* $\widehat{\Pi}_{\mathsf{TOSS}}$ *wsrUC-realizes the* $\mathcal{F}_{\mathsf{TOSS}}$ *functionality in the* $(\mathcal{F}_{\mathsf{SAT}}, \widehat{\mathcal{F}}_{\mathsf{sCOM}})$*-hybrid model in the presence of up to* $n-1$ *malicious corruptions.*

5 Completeness Theorem

In this section, we show how to sanitize the classical compiler by Goldreich, Micali and Wigderson (GMW) [20], for turning MPC protocols with security against *semi-honest* adversaries into ones with security against *malicious adversaries*. On a high level, the GMW compiler works by having each party commit to its input. Furthermore, the parties run a coin tossing protocol to determine the randomness to be used in the protocol; since the random tape of each party must be secret, the latter is done in such a way that the other parties only learn a commitment to the other parties' random tape. Finally, the commitments to each party's input and randomness are used to enforce semi-honest behavior: Each party computes the next message using the underlying semi-honest protocol, but also proves in zero knowledge that this message was computed correctly using the committed input and randomness.

5.1 Sanitizable Commit & Prove

The GMW compiler was analyzed in the UC setting by Canetti, Lindell, Ostrovsky and Sahai [9]. A difficulty that arises is that the receiver of a UC commitment obtains no information about the value that was committed to. Hence, the parties cannot prove in zero knowledge statements relative to their input/randomness commitment. This issue is solved by introducing a more general commit-and-prove functionality that essentially combines both the commitment and zero-knowledge capabilities in a single functionality. In turn, the commit-and-prove functionality can be realized using commitments and zero-knowledge proofs.

In order to sanitize the GMW compiler, we follow a similar approach. Namely, we introduce a sanitazable commit-and-prove functionality (denoted $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ and depicted below) and show that this functionality suffices for our purpose. Intuitively, $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ allows the core C_i of each party P_i to (i) commit to multiple secret inputs x, and (ii) prove arbitrary NP statements y (w.r.t. an underlying relation R that is a parameter of the functionality) whose corresponding witnesses consist of all the values x. Whenever the core C_i commits to a value x, the firewall F_i may decide to blind x with a random string r (which is then revealed to the core). Similarly, whenever the core proves a statement y, the firewall F_i may check if the given statement makes sense, in which case, and assuming the statement is valid, the functionality informs all other parties that y is indeed a correct statement proven by P_i.

Functionality $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$

The sanitizable commit-and-prove functionality $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ is parameterized by an NP relation R, and runs with parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ (each consisting of a core C_i and a firewall F_i) and an adversary $\mathcal{S}$. The functionality consists of the following communication interfaces for the cores and the firewalls respectively.

Interface IO
- Upon receiving a message $(\textsc{Commit}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, x)$ from C_i, where $x \in \{0,1\}^*$, record the tuple $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, x)$ and send the message $(\textsc{Receipt}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i)$ to F_i. Ignore future commands of the form $(\textsc{Commit}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$.
- Upon receiving a message $(\textsc{Prove}, \mathsf{sid}, \mathsf{C}_i, y)$ from C_i, if there is at least one record $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$ and a corresponding $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$ message was sent to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, then send the message $(\textsc{Sanitize}, \mathsf{sid}, \mathsf{C}_i, y)$ to F_i.

Interface S
- Upon receiving a message $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, r)$ from F_i, where $r \in \{0,1\}^*$, proceed as follows: if the tuple $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, x)$ is recorded, modify the tuple to be $(\mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \hat{x} = x \oplus r)$ and send the message $(\textsc{Blinded}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, r)$ to C_i, and $(\textsc{Receipt}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i)$ to all $\mathsf{C}_{j \neq i}$ and $\mathcal{S}$; otherwise do nothing. Ignore future commands of the form $(\textsc{Blind}, \mathsf{sid}, \mathsf{cid}, \mathsf{C}_i, \cdot)$.
- Upon receiving a message $(\textsc{Continue}, \mathsf{sid}, \mathsf{C}_i, y)$ from F_i, retrieve all tuples of the form $(\mathsf{sid}, \cdot, \mathsf{C}_i, \hat{x})$ and let $\overline{x}$ be the list containing all (possibly sanitized) witnesses $\hat{x}$. Then compute $R(y, \overline{x})$: if $R(y, \overline{x}) = 1$ send the message $(\textsc{Proved}, \mathsf{sid}, \mathsf{C}_i, y)$ to all $\mathsf{C}_{j \neq i}$ and $\mathcal{S}$, otherwise ignore the command.

In the full version [12], we show how to realize the sanitazable commit-and-prove functionality from *malleable dual-mode commitments*, a primitive which we introduce, and re-randomizable NIZKs for all of NP. Our commitment protocol from Section 3 can be seen as a concrete instantiation of malleable dual-mode commitments based on the DDH assumption.

5.2 Sanitizing the GMW Compiler

We are now ready to sanitize the GMW compiler. Let Π be an MPC protocol. The (sanitized) protocol $\widehat{\Pi}_{\mathsf{GMW}}$ is depicted below and follows exactly the ideas outlined above adapted to the UC framework with reverse firewalls.

Protocol $\widehat{\Pi}_{\mathsf{GMW}}$ (Sanitizing the GMW compiler)

The protocol is described in the $(\widehat{\mathcal{F}}_{\mathsf{C\&P}}, \mathcal{F}_{\mathsf{TOSS}})$-hybrid model, and is executed between parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ each consisting of a core C_i and a firewall F_i. Party $\mathsf{P}_i = (\mathsf{C}_i, \mathsf{F}_i)$ proceeds as follows (the code for all other parties is analogous).

Random tape generation: When activated for the first time, party P_i generates its own randomness with the help of all other parties:
1. The core C_i picks a random $s_i \in \{0,1\}^\lambda$ and sends $(\textsc{Commit}, \mathsf{sid}_i, \mathsf{cid}_i, s_i)$ to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$.

2. Upon receiving (Receipt, $\mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i$) from $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, the firewall F_i picks a random $r_i \in \{0,1\}^\lambda$ and sends (Blind, $\mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i, r_i$) to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$.
3. All the cores interact with $\mathcal{F}_{\mathsf{TOSS}}$ in order to obtain a public random string s_i^* that is used to determine the random tape of C_i. Namely, each core C_j, for $j \in [n]$, sends (Init, $\mathsf{sid}_{i,j}, \mathsf{P}_j$) to $\mathcal{F}_{\mathsf{TOSS}}$ and waits to receive the message (Delivered, $\mathsf{sid}_{i,j}, \mathsf{P}_j, s_i^*$) from the functionality.
4. Upon receiving (Blinded, $\mathsf{sid}_i, \mathsf{cid}_i, \mathsf{C}_i, r_i$) from $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, the core C_i defines $\hat{r}_i = s_i^* \oplus (s_i \oplus r_i)$.

Input commitment: When activated with input x_i, the core C_i sends (Commit, $\mathsf{sid}_i, \mathsf{cid}'_i, x_i$) to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ and adds x_i to the (initially empty) list of inputs $\overline{x}_i$ (containing the inputs from all the previous activations of the protocol). Upon receiving (Receipt, $\mathsf{sid}_i, \mathsf{cid}'_i, \mathsf{C}_i$) from $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, the firewall F_i sends (Blind, $\mathsf{sid}_i, \mathsf{cid}'_i, \mathsf{C}_i, 0^{|x_i|}$) to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$.

Protocol execution: Let $\tau \in \{0,1\}^*$ be the sequence of messages that were broadcast in all activations of Π until now (where τ is initially empty).

1. The core C_i runs the code of Π on its input list $\overline{x}_i$, transcript τ, and random tape $\hat{r}_i$ (as determined above). If Π instructs P_i to broadcast a message, proceed to the next step.
2. For each outgoing message μ_i that P_i sends in Π, the core C_i sends (Prove, $\mathsf{sid}_i, \mathsf{C}_i, (\mu_i, s_i^*, \tau)$) to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, where the relation parameterizing the functionality is defined as follows:
$$R := \{((\mu_i, s_i^*, \tau), (\overline{x}_i, s_i, r_i)) : \mu_i = \Pi(\overline{x}_i, \tau, s_i^* \oplus (s_i \oplus r_i))\}.$$
In words, the core C_i proves that the message μ_i is the correct next message generated by Π when the input sequence is $\overline{x}_i$, the random tape is $\hat{r}_i = s_i^* \oplus (s_i \oplus r_i)$, and the current transcript is τ. Thus, C_i appends μ_i to the current transcript τ.
3. Upon receiving (Sanitize, $\mathsf{sid}_i, \mathsf{C}_i, (\mu_i, s_i^*, \tau)$) from $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, the firewall F_i verifies that s_i^* is the same string obtained via $\mathcal{F}_{\mathsf{TOSS}}$ and that τ consists of all the messages that were broadcast in all the activations up to this point. If these conditions are not met, F_i ignores the message and otherwise it sends (Continue, $\mathsf{sid}_i, \mathsf{C}_i, (\mu_i, s_i^*, \tau)$) to $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$ and appends μ_i to the current transcript τ.
4. Upon receiving (Proved, $\mathsf{sid}_j, \mathsf{C}_j, (\mu_j, s_i^*, \tau)$) from $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$, both the core C_i and the firewall F_i append μ_j to the transcript τ and repeat the above steps.

Output: Whenever Π outputs a value, $\widehat{\Pi}_{\mathsf{GMW}}$ generates the same output.

A few remarks are in order. First, and without loss of generality, we assume that the underlying protocol Π is reactive and works by a series of activations, where in each activation, only one of the parties has an input; the random tape of each party is taken to be a λ-bit string for simplicity. Second, each party needs to invoke an independent copy of $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$; we identify these copies as sid_i, where we can for instance let $\mathsf{sid}_i = \mathsf{sid}||i$. Third, we slightly simplify the randomness generation phase using the coin tossing functionality $\mathcal{F}_{\mathsf{TOSS}}$. In particular, each core C_i commits to a random string s_i via $\widehat{\mathcal{F}}_{\mathsf{C\&P}}$; the corresponding firewall F_i blinds s_i with a random string r_i. Thus, the parties obtain public randomness

s_i^* via $\mathcal{F}_{\mathsf{TOSS}}$, yielding a sanitized random tape $\hat{r}_i = s_i^* \oplus (s_i \oplus r_i)$ for party P_i. Note that it is crucial that the parties obtain independent public random strings s_i^* in order to determine the random tape of party P_i. In fact, if instead we would use a single invocation of $\mathcal{F}_{\mathsf{TOSS}}$ yielding common public randomness s, two malicious parties P_i and P_j could pick the same random tape by choosing the same values s_i, r_i, s_j, r_j. Clearly, the latter malicious adversary cannot be reduced to a semi-honest adversary.
The theorem below states the security of the GMW compiler with reverse firewalls. The proof is deferred to the full version [12].

Theorem 4. *Let $\mathcal{F}$ be any functionality for n parties. Assuming that Π UC realizes $\mathcal{F}$ in the presence of up to $t \leq n-1$ semi-honest corruptions, then the compiled protocol $\widehat{\Pi}_{\mathsf{GMW}}$ wsrUC realizes $\mathcal{F}$ in the $(\mathcal{F}_{\mathsf{SAT}}, \widehat{\mathcal{F}}_{\mathsf{C\&P}}, \mathcal{F}_{\mathsf{TOSS}})$-hybrid model in the presence of up to t malicious corruptions.*

6 Conclusions and Future Work

We have put forward a generalization of the UC framework by Canetti [7,6], where each party consists of a core (which has secret inputs and is in charge of generating protocol messages) and a reverse firewall (which has no secrets and sanitizes the outgoing/incoming communication from/to the core). Both the core and the firewall can be subject to different flavors of corruption, modeling the strongly adversarial setting where a subset of the players is maliciously corrupt, whereas the remaining honest parties are subject to subversion attacks. The main advantage of our approach is that it comes with very strong composition guarantees, as it allows, for the first time, to design subversion-resilient protocols that can be used as part of larger, more complex protocols, while retaining security even when protocol sessions are running concurrently (under adversarial scheduling) and in the presence of subversion attacks.

Moreover, we have demonstrated the feasibility of our approach by designing UC reverse firewalls for cryptographic protocols realizing pretty natural ideal functionalities such as commitments and coin tossing, and, in fact, even for arbitrary functionalities. Several avenues for further research are possible, including designing UC reverse firewalls for other ideal functionalities (such as oblivious transfer and zero knowledge), removing (at least partially) trusted setup assumptions, and defining UC subversion-resilient MPC in the presence of adaptive corruptions.

References

1. Ateniese, G., Magri, B., Venturi, D.: Subversion-resilient signature schemes. In: ACM CCS 2015. ACM Press (Oct 2015)
2. Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: ACM CCS 2015. ACM Press (Oct 2015)

3. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: CRYPTO 2014, Part I. LNCS, Springer, Heidelberg (Aug 2014)
4. Berndt, S., Liskiewicz, M.: Algorithm substitution attacks from a steganographic perspective. In: ACM CCS 2017. ACM Press (Oct / Nov 2017)
5. Blum, M.: Coin flipping by telephone. In: CRYPTO'81. U.C. Santa Barbara, Dept. of Elec. and Computer Eng. (1981)
6. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000), https://eprint.iacr.org/2000/067
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. IEEE Computer Society Press (Oct 2001)
8. Canetti, R., Fischlin, M.: Universally composable commitments. In: CRYPTO 2001. LNCS, Springer, Heidelberg (Aug 2001)
9. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: 34th ACM STOC. ACM Press (May 2002)
10. Canetti, R., Sarkar, P., Wang, X.: Efficient and round-optimal oblivious transfer and commitment with adaptive security. In: ASIACRYPT 2020, Part III. LNCS, Springer, Heidelberg (Dec 2020)
11. Chakraborty, S., Dziembowski, S., Nielsen, J.B.: Reverse firewalls for actively secure MPCs. In: CRYPTO 2020, Part II. LNCS, Springer, Heidelberg (Aug 2020)
12. Chakraborty, S., Magri, B., Nielsen, J.B., Venturi, D.: Universally composable subversion-resilient cryptography. Cryptology ePrint Archive (2022), https://www.iacr.org
13. Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Malleable proof systems and applications. In: EUROCRYPT 2012. LNCS, Springer, Heidelberg (Apr 2012)
14. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: 18th ACM STOC. ACM Press (May 1986)
15. Degabriele, J.P., Paterson, K.G., Schuldt, J.C.N., Woodage, J.: Backdoors in pseudorandom number generators: Possibility and impossibility results. In: CRYPTO 2016, Part I. LNCS, Springer, Heidelberg (Aug 2016)
16. Dodis, Y., Ganesh, C., Golovnev, A., Juels, A., Ristenpart, T.: A formal treatment of backdoored pseudorandom generators. In: EUROCRYPT 2015, Part I. LNCS, Springer, Heidelberg (Apr 2015)
17. Dodis, Y., Mironov, I., Stephens-Davidowitz, N.: Message transmission with reverse firewalls—secure communication on corrupted machines. In: CRYPTO 2016, Part I. LNCS, Springer, Heidelberg (Aug 2016)
18. Fischlin, M., Janson, C., Mazaheri, S.: Backdoored hash functions: Immunizing HMAC and HKDF. In: CSF 2018 Computer Security Foundations Symposium. IEEE Computer Society Press (2018)
19. Ganesh, C., Magri, B., Venturi, D.: Cryptographic reverse firewalls for interactive proof systems. In: ICALP 2020. LIPIcs, Schloss Dagstuhl (Jul 2020)
20. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: 19th ACM STOC. ACM Press (May 1987)
21. Mironov, I., Stephens-Davidowitz, N.: Cryptographic reverse firewalls. In: EUROCRYPT 2015, Part II. LNCS, Springer, Heidelberg (Apr 2015)
22. Young, A., Yung, M.: The dark side of "black-box" cryptography, or: Should we trust capstone? In: CRYPTO'96. LNCS, Springer, Heidelberg (Aug 1996)
23. Young, A., Yung, M.: Kleptography: Using cryptography against cryptography. In: EUROCRYPT'97. LNCS, Springer, Heidelberg (May 1997)