Universally Composable Multi-party Computation Using Tamper-Proof Hardware

Jonathan Katz*

Dept. of Computer Science, University of Maryland
jkatz@cs.umd.edu

Abstract. Protocols proven secure within the *universal composability (UC) framework* satisfy strong and desirable security properties. Unfortunately, it is known that within the "plain" model, secure computation of general functionalities without an honest majority is impossible. This has prompted researchers to propose various "setup assumptions" with which to augment the bare UC framework in order to bypass this severe negative result. Existing setup assumptions seem to inherently require *some* trusted party (or parties) to initialize the setup in the real world.

We propose a new setup assumption — more along the lines of a *physical* assumption regarding the existence of tamper-proof hardware — which also suffices to circumvent the impossibility result mentioned above. We suggest this assumption as potentially leading to an approach that might alleviate the need for trusted parties, and compare our assumption to those proposed previously.

1 Motivation

For many years, researchers considered the security of protocols in a stand-alone setting where a single protocol execution was considered in isolation. Unfortunately, a proof of stand-alone security for a protocol does not, in general, provide any guarantees when the protocol is executed multiple times in a concurrent fashion (possibly by different sets of parties), or in a network where other protocol executions are taking place. This realization has motivated a significant amount of work aimed at providing models and security definitions that explicitly address such concerns.

The universal composability (UC) framework, introduced by Canetti [6], gives strong security guarantees in exactly such a setting. (Other frameworks with similar guarantees also exist [20], but we adopt the UC model in this work.) We refer the reader to Canetti's paper for a full discussion of the advantages of working within this framework, and focus instead on the question of *feasibility*. Canetti's initial work already demonstrates broad feasibility results for realizing any (polynomial-time computable) multi-party functionality in the presence of

* This research was supported by NSF Trusted Computing grant #0310751, NSF CAREER award #0447075, and US-Israel Binational Science Foundation grant #2004240. Portions of this work were done while visiting IPAM.

a strict majority of honest players. Unfortunately, this was soon followed by results of Canetti and Fischlin [8] showing that the setting without an honest majority is substantially different: even for the case of two parties (one of whom may be malicious) there exist natural functionalities that cannot be securely computed within the UC framework. Subsequent work of Canetti, et al. [9], further characterizing those two-party functionalities which cannot be securely realized in the UC framework, rules out essentially all non-trivial functions.

The impossibility results mentioned above hold for the so-called "plain model" where there is no additional infrastructure beyond the communication channels available to the parties. (The term "plain model" is actually a bit misleading, since even the plain model usually incorporates quite strong — though standard — assumptions about the communication channels, such as the existence of authenticated channels between all pairs of parties as well as a broadcast channel/"bulletin board" [6, Sect. 6.2]. We stress that the impossibility results hold even in this case.) In contrast, the impossibility results can be bypassed if one is willing to assume some stronger form of "setup" in the network. This idea was first proposed in the UC framework by Canetti and Fischlin [8], who suggest using a common reference string (CRS) in order to circumvent the impossibility results shown in their paper. (The use of a CRS in other contexts has a long history going back to [4].) In fact, a CRS turns out to suffice for universally composable multi-party computation of any (well-formed) functionality, for any number of corrupted parties [10].

If universally composable protocols are ever to be used in practice, one important research direction is to further explore setup assumptions that suffice to obtain feasibility results in the UC framework similar to those of [10]. Having multiple setup assumptions available would offer options to protocol designers; furthermore, some assumptions may be more attractive than others depending on the scenario in which protocols are to be run. Indeed, a variety of setup assumptions have been investigated recently including variations of trusted "public-key registration" services [2,7] (see also [6, Sect. 6.6]), or the use of government-issued "signature cards" [15]; these are discussed further in the following section.

From a high-level perspective, one very important research direction is to determine whether (or to what extent) *trusted parties* are needed for obtaining broad feasibility results in the UC framework. It appears in particular that all existing solutions require some trusted party to initialize the setup in the real world. (See the discussion in the following section.) It might be possible, however, to replace this trust with some *physical* assumption about the environment in which the protocol is run. Using physical assumptions to circumvent impossibility results is not without precedent in cryptography (though it has not been considered previously in the context of the UC framework); examples analogous to what we have in mind include the assumption of a physical broadcast channel (or even "multicast channels" [12]) to circumvent impossibility results [18] regarding the fraction of malicious players that can be tolerated; or the assumption of noisy channels [21,14,16] or the laws of quantum mechanics [3] to achieve information-theoretically secure key agreement over public channels.

We present in this paper what is intended to be a partial step toward this goal. Specifically, we introduce an assumption that has the flavor of a physical assumption regarding the possibility of tamper-proof hardware, and show that UC multi-party computation is realizable with respect to this assumption. A difficulty, of course, is that although there may be some intuitive idea of the properties possessed by tamper-proof hardware in the real world, it is not at all clear what is the most appropriate way to mathematically *model* tamper-proof hardware in the UC framework. We do not claim to have found the "right" formalization. Instead, we intend this work only to serve as an indication of what might be possible, and as inspiration for subsequent work in this direction.

1.1 A Brief Review of Existing Solutions

As mentioned earlier, a variety of setup assumptions have been explored in an attempt to circumvent the impossibility results of [8]. We briefly discuss these now.

Common reference string (CRS). The use of a CRS was suggested by [8] (in the UC setting) and has been used in much subsequent work. It is fair to say that this is the setup assumption that has so far received the most attention. In the CRS model, a string is generated according to some prescribed distribution by a trusted party and given to the parties running an execution of a protocol. We remark that only the parties running the protocol are supposed to have access to the string, and so this is not quite a "common" reference string as in the original work of [4] (see [7] for a discussion of this point).

If the party publishing the CRS is malicious, this party can potentially set things up so that it can learn all private data in the network or cheat undetectably in protocol executions in which it is involved. These problems can be mitigated to some extent by having the CRS generated in a threshold manner so that, say, security holds as long as a majority of the parties involved in generation of the CRS are honest. Nevertheless, this still requires all parties in the network to jointly agree to place their trust in a small set of parties, and also assumes the availability of some set of parties willing to take responsibility for generating a CRS.

For some protocols, the CRS is simply a uniformly-random string (this is often called the common *random* string model) and here one might hope that the string could be generated based on naturally-occurring (random) events and without relying on a trusted party. The main drawback of this approach is that although certain natural events can be viewed as producing bit-sources with high min-entropy, the resulting bit-sources may not be uniformly random (and, furthermore, may not allow for deterministic extraction).

Public-key registration services. Existing proposals for public-key registration services [2,7] that can be used to circumvent the impossibility results in the UC framework go beyond the "traditional" model in which parties simply publish their public keys. (The latter corresponds to the "basic" registration functionality described in [6, Sect. 6.2], for which the impossibility results regarding secure computation still hold.) The functionality in [2], for example, essentially prevents an adversary from registering any public key that is not "well-formed" and for which the adversary does not know the corresponding secret key. It is unclear how this would be implemented in practice without a significant assumption of trust on the part of existing certification authorities.

Signature cards. An interesting idea pursued by [15] is to use government-issued signature cards as a form of global setup. Roughly speaking, such cards hold an honestly-generated public-/secret- key pair for a secure signature scheme; sign any message given to them; and never reveal the secret key to anyone (including the legitimate owner of the card) under any circumstances. Cards with this functionality are apparently being issued by some European governments [15] indicating that such cards may, in fact, represent a realistic assumption. The main drawback of these signature cards is that the producer and/or issuer of these cards must be completely trusted.

1.2 Relying on Tamper-Proof Hardware

As discussed above, all existing setup assumptions that imply general feasibility results in the UC framework seem to inherently require a great deal of *trust* in at least some parties in the system. It is natural to wonder whether such trust is essential, or whether other setup assumptions — perhaps of a slightly different character — might also suffice.

We suggest that it might be possible to eliminate the need for any trusted parties if one is willing instead to rely on a *physical* assumption regarding the existence (and practicality) of tamper-proof hardware. We hasten to add that a complete elimination of all trusted parties using our approach may not be practical, possible, or desirable in realistic scenarios; nevertheless, our proposals indicates that at least in theory this might be achievable. Alternately, one can view the approach explored here as allowing a reduced level of trust that we might be comfortable with; after all, we generally trust that our packets will be routed correctly over the Internet, but may not be willing to trust a corporation to generate a CRS.

Our assumption is that tamper-proof hardware exists, in the sense that (1) an honest user can construct a hardware token T_F implementing any desired (poly-time) functionality F but (2) an adversary given T_F can do no more than observe the input/output characteristics of this token. An honest player given a token $T'_{F'}$ by an adversary has no guarantee whatsoever regarding the function F' that this token implements (other than what the honest user can deduce from the input/output of this device). We show how this, seemingly-basic primitive can be used along with standard cryptographic assumptions to realize the commitment functionality (and hence general secure computation [10]) in the UC framework.

The above is a rather informal summary of the properties we assume; a more formal discussion of how to model tamper-proof hardware (as well as a concrete ideal functionality meant to capture that requirements evidenced in that discussion) is given in Section 2.

The idea of using secure hardware to achieve stronger security properties is not entirely new; it was directly inspired by work of Chaum, Pedersen, Brands, and Cramer [11,5,13] who propose the use of *observers* in the context of e-cash. Roughly speaking, it was suggested in that line of work that a bank could issue each user an "observer" (i.e., a smartcard) T_F implementing some functionality F, and a user would interact with both the bank and T_F whenever it executed an instance of a blind signature protocol to withdraw an e-coin (the bank and T_F could not communicate directly). The observer, by monitoring the actions of the user, could enforce some sort of honest behavior on the part of the user in the protocol execution. On the other hand, the user was guaranteed that even if the bank were malicious (and, e.g., sent a smartcard that was programmed in some arbitrary manner), anonymity of the user could not be violated. In some sense our work can be seen as formalizing this earlier work on observers, and extending its applicability from the domain of e-cash to the case of secure computation of arbitrary functionalities.

1.3 Have We Gained Anything?

Our assumption regarding tamper-proof hardware does not seem to trivially imply any of the setup assumptions discussed in Section 1.1. For example, two parties A and B cannot generate a CRS by simply having A send to B a token implementing a coin-tossing protocol: if A is malicious, a simulator will indeed be able to "rewind" the hardware token provided by A to B, and thus be able to "force" the value of the CRS output in this case to any desired value. On the other hand, if B is malicious then the simulator must (informally speaking) send some token to A, but then cannot "rewind" A to force the value of the CRS. We also do not see any way to trivially implement key-registration using our approach: for example, if each party sends to the other a token that checks public keys for validity (and then, say, outputs a signed receipt) then even the honest party will have to produce a secret key corresponding to its public key, which is not the case in the key-registration functionality of [2] (indeed, the security proofs in that work break down if this is the case). Another problem, unrelated to this, is that the signed receipt output by the device might be used as a "covert channel" to leak information about honest users' private keys.

As for whether we fundamentally gain anything by introducing our new assumption, this is (of course) subject to debate though we hope to convince the reader that the answer is "yes." In what follows we will simply assume that tamper-proof hardware is (or will someday be) available; clearly, if this assumption is false (and it may well be) then the entire discussion is meaningless. Under this assumption, we summarize our arguments in favor of relying on tamper-proof hardware as follows:

Possible elimination of trust. An advantage of our approach is that it seems to potentially allow for the elimination of trust in anyone but oneself. This is because, in theory, each user could construct the hardware token itself (or, more likely, buy a "blank" token and program it itself) without having to rely on anyone else. This distinguishes our approach from the "signature card" approach described earlier, where it is essential that a specific third party produce the cards (and a user cannot produce cards by itself).

An objection here is that we still assume secure channels and also secure distribution of tokens, and so trust has not been completely eliminated. We first emphasize that existing impossibility results hold even if secure channels are available, and so in that sense being able to eliminate the additional need for a trusted CRS represents progress in the right direction. If secure channels do not exist (and if secure distribution of tokens is not possible) we would seem to degenerate to a security model like that of [1] which still guarantees a non-trivial level of security. Finally, secure distribution of tokens is possible if a physical meeting of parties can be arranged; given this, a key can be stored on the token at the same time so as to bootstrap secure channels.

We do not mean to minimize the above concerns, only to suggest how they might be overcome. Developing a complete solution eliminating all trust in a practical manner remains an interesting direction for future work.

Possible reduction of trust. The above is a bit of an extreme scenario. But it is indicative of the fact that our approach may allow for *more relaxed requirements on trust*. In particular, under our approach each party could choose to buy pre-programmed tokens from any vendor of their choice; other parties executing the protocol do not need to approve of this choice, and can in turn buy from any vendors of their choice. This is not the case for any of the other setup assumptions mentioned in Section 1.1: parties must agree on which CRS to use; must approve of the registration authorities used by other parties; or must be sure that other parties use signature cards produced by a trusted entity.

Accountability. A final important point is the *accountability* present in our approach, which does not seem to be present when parties use a CRS or a key-registration authority. (It does seem to be available in the signature card scenario.) In the case of a CRS, for example, it seems impossible to prove that a CRS generated by some party is "bad" — in particular, the CRS might come from the exactly correct distribution except that the party has "neglected" to erase the trapdoor information associated with this CRS. Similarly in the case of a registration authority: how would one prove that an adversary's key is *not* well-formed?

On the other hand, one could imagine independent labs demonstrating that hardware sold by some vendor is *not* tamper-proof, or that supposedly blank tokens contained some (hidden) embedded code. This is in some sense reminiscent of the distinction suggested by Naor [17] between "falsifiable" assumptions and "unfalsifiable" ones.

2 Modeling Tamper-Proof Hardware

In this section, we suggest an ideal functionality that is intended to model tamper-proof hardware. More accurately, we define a "wrapper" functionality

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

$\mathcal{F}_{\mathsf{wrap}}$ is parameterized by a polynomial p and an implicit security parameter k

"Creation" Upon receiving $(\mathsf{create}, \mathsf{sid}, P, P', M)$ from P, where P' is another user in the system and M is an interactive Turing machine, do:

1. Send $(\mathsf{create}, \mathsf{sid}, P, P')$ to P'.
2. If there is no tuple of the form $(P, P', \star, \star, \star)$ stored, then store $(P, P', M, 0, \emptyset)$.

"Execution" Upon receiving $(\mathsf{run}, \mathsf{sid}, P, \mathsf{msg})$ from P', find the unique stored tuple $(P, P', M, i, \mathsf{state})$ (if no such tuple exists, then do nothing). Then do:

Case 1 ($i = 0$): Choose random $\omega \leftarrow \{0,1\}^{p(k)}$. Run $M(\mathsf{msg}; \omega)$ for at most $p(k)$ steps, and let out be the response (set $\mathsf{out} = \perp$ if M does not respond in the allotted time). Send $(\mathsf{sid}, P, \mathsf{out})$ to P'. Store $(P, P', M, 1, (\mathsf{msg}, \omega))$ and erase $(P, P', M, i, \mathsf{state})$.

Case 2 ($i = 1$): Parse state as (msg_1, ω). Run $M(\mathsf{msg}_1 \| \mathsf{msg}; \omega)$ for at most $p(k)$ steps, and let out be the response (set $\mathsf{out} = \perp$ if M does not respond in the allotted time). Send $(\mathsf{sid}, P, \mathsf{out})$ to P'. Store $(P, P', M, 0, \emptyset)$ and erase $(P, P', M, i, \mathsf{state})$.

Fig. 1. The $\mathcal{F}_{\mathsf{wrap}}$ functionality, specialized for the case when M is a 2-round (i.e., 4-message) protocol

which is intended to model the following sequence of events in the real world: (1) a party takes some software and "seals" it inside a tamper-proof hardware token; (2) this party gives the token to another party, who can then access the embedded software in a black-box manner. We will sometimes refer to the first party as the *creator* of the token, and the other party as the token's *user*.

The wrapper functionality is presented in Figure 1. The formalism in the description obscures to some extent what is going on, so we give a high-level description here. The functionality accepts two types of messages: the first type is used by a party P to create a hardware token (encapsulating an interactive protocol M) and to "give" this token to another party P'. The functionality enforces that P can send at most one token to P' which is used for all their protocol interactions throughout their lifetimes (and not just for the interaction labeled by the sid used when the token is created); since this suffices for honest parties we write the functionality this way in an effort to simplify things.

Once the token is "created" and "given" to P', this party can interact with the token in an arbitrary black-box manner. This is formalized by allowing P' to send messages of its choice to M via the wrapper functionality $\mathcal{F}_{\mathsf{wrap}}$. Note that each time a new copy of M is invoked, a fresh random tape is chosen for M.

To simplify the description of the functionality, we have assumed that M represents a 2-round (4-message) protocol since our eventual construction of commitment will use an M of this form. It should be clear how the functionality can be extended for the more general case.

A real-world action that is not modeled here is the possible (physical) *transference* of a token from one party to another. An honest party is never supposed to transfer a token; furthermore, in our eventual construction, tokens created by honest parties allow easy identification of their creator. Thus, transference does not represent a viable adversarial action, and so for simplicity we have not modeled such an action within $\mathcal{F}_{\mathsf{wrap}}$.

The following real-world assumptions underly the existence of $\mathcal{F}_{\mathsf{wrap}}$:

– We assume that the party creating a hardware token "knows" the code corresponding to the actions the token will take. This is evidenced by the fact that the creator P must explicitly provide $\mathcal{F}_{\mathsf{wrap}}$ with a description of M. Looking ahead, this property will allow the simulator to "extract" the code within any adversarially-created token.

– The hardware token must be completely tamper-proof, so that the user P' cannot learn anything about M that it could not learn given black-box access. Furthermore, P' cannot cause M to use a "bad" (i.e., non-uniform) random tape, or to use the same random tape more than once. We are thus also assuming that the token has access to a built-in source of randomness. This latter requirement is not needed if we are willing to assume that the token can maintain state — in that case, we can use a hard-coded key for a pseudorandom function to generate the random tape as needed. Unfortunately we do not know how to prove security of this approach (for our particular protocol) without relying on complexity leveraging.

– We also assume that the creator of a token cannot send messages to the token once it is given to another party. (On the other hand, the token can send messages to its creator, either directly or via a covert channel.)

Our results are meaningful only to the extent that one is prepared to accept these assumptions as reasonable, or at least more reasonable than the existence of a common reference string or the other setup assumptions discussed earlier.

3 Using Tamper-Proof Hardware for Secure Computation

We now show how to securely realize the multiple commitment functionality $\mathcal{F}_{\mathrm{mcom}}$ (see [8]) in the $\mathcal{F}_{\mathsf{wrap}}$-hybrid model, for static adversaries. By the results of [8,10], this implies the feasibility of computing any (well-formed) two-party functionality, again fir static adversaries. It is also not hard to see that the techniques used in [10] can be used to show that our results imply the feasibility of computing any (well-formed) multi-party functionality as well. We omit further details from the present abstract.

For convenience, the multiple commitment functionality is given in Figure 2. Although we could optimize our construction to allow commitment to strings, for simplicity we focus on commitment to a single bit.

Before describing our protocol we introduce some notation. A tuple $(p, g, h, \hat{g}, \hat{h})$ is called a *Diffie-Hellman tuple* if (1) p and $q \overset{\text{def}}{=} \frac{p-1}{2}$ are prime; (2) $g, h, \hat{g}, \hat{h}$

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

Commit phase Upon receiving $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid}, P, P', b)$ from P, where $b \in \{0,1\}$, record (cid, P, P', b) and send $(\mathsf{receipt}, \mathsf{sid}, \mathsf{cid}, P, P')$ to P' and the adversary. Ignore subsequent values $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid}, P, P', \star)$ from P.

Decommitment phase Upon receiving $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, P, P')$ from P, if the tuple (cid, P, P', b) is recorded then send $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, P, P', b)$ to P' and the adversary. Otherwise do nothing.

Fig. 2. The $\mathcal{F}_{\mathsf{mcom}}$ functionality

are in the order-q subgroup $\mathbb{G} \subset \mathbb{Z}_p^*$, with g, h generators; and (3) $\log_g \hat{g} = \log_h \hat{h}$. If the first two conditions hold but $\log_g \hat{g} \neq \log_h \hat{h}$, then we refer to the tuple as a *random tuple*. Given $\mathsf{tuple} = (p, g, h, \hat{g}, \hat{h})$ with q as defined above, we let $\mathsf{Com}_{\mathsf{tuple}}(b)$ denote the commitment defined by the two group elements $g^{r_1}h^{r_2}$, $\hat{g}^{r_1}\hat{h}^{r_2}g^b$, for randomly-chosen $r_1, r_2 \in \mathbb{Z}_q$. It is well-known (and easy to check) that if tuple is a random tuple then this commitment scheme is perfectly hiding; on the other hand if tuple is a Diffie-Hellman tuple and $r = \log_g \hat{g} = \log_h \hat{h}$ is known, then b can be efficiently recovered from the commitment.

We now describe a complete protocol for realizing $\mathcal{F}_{\mathsf{mcom}}$ for a sender P and a receiver P'. The security parameter is denoted by k.

Commitment phase. The parties perform the following steps:

1. P generates a public-key/secret-key pair (PK, SK) for a secure digital signature scheme, and constructs and sends a token to P' encapsulating the following functionality M:
   (a) Wait for a message (p, g, h). Check that p and $\frac{p-1}{2} = q$ are prime, that p has length k, and that g, h are generators of the order-q subgroup $\mathbb{G} \subset \mathbb{Z}_p^*$, and aborts if these do not hold.
   (b) Choose random elements $g_1, h_1 \in \mathbb{G}$. Using the Pedersen (perfectly-hiding) commitment scheme [19] and the generators received in the previous step, commit to g_1, h_1.
   (c) Wait for a message (g_2, h_2) where $g_2, h_2 \in \mathbb{G}$. (Abort if an invalid message is received.)
   (d) Set $\hat{g} = g_1 g_2$ and $\hat{h} = h_1 h_2$. Define $\mathsf{tuple}_{P \to P'} \overset{\text{def}}{=} (p, g, h, \hat{g}, \hat{h})$, and compute $\sigma_{P \to P'} = \mathsf{Sign}_{SK}(P, P', \mathsf{tuple}_{P \to P'})$. As the final message, send $\sigma_{P \to P'}$ as well as decommitment information for the commitments sent in the previous round.

   P' symmetrically constructs and sends a token to P.
2. P interacts with the token sent to it by P' and in this way obtains $\mathsf{tuple}_{P' \to P}$ and $\sigma_{P' \to P}$. (If cheating on the part of the token is detected, then P aborts

the entire protocol.) Party P' acts symmetrically. From now on, the parties communicate directly with each other and no longer need to access their tokens.

3. P sends $\mathsf{tuple}_{P' \to P}$ and $\sigma_{P' \to P}$ to P', and P' acts symmetrically. Then P checks that $\mathsf{Vrfy}_{PK}(\mathsf{tuple}_{P \to P'}, \sigma_{P \to P'}) = 1$ and, if not, it aborts the protocol. Party P' acts symmetrically.
   At the end of this step each party holds $\mathsf{tuple}_{P' \to P}$ and $\mathsf{tuple}_{P \to P'}$.
4. This step is the first that depends on the input bit b to be committed. P first commits to b using any statistically-binding commitment scheme; let C denote the resulting commitment. P also chooses random r_1, r_2 and computes $\mathsf{com} = \mathsf{Com}_{\mathsf{tuple}_{P \to P'}}(b)$. It sends C and com to P', and then gives an (interactive) witness indistinguishable proof that either (1) both C and com are commitments to the same bit b, or (2) $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple.
5. Upon successful completion of the previous step, party P' outputs ($\mathsf{receipt}$, sid, cid, P, P').

We remark that steps 1–3 need only be carried out once by parties P and P', after which the values $\mathsf{tuple}_{P \to P'}$ and $\mathsf{tuple}_{P' \to P}$ can be used by these same parties to commit to each other (with either party acting as the sender) arbitrarily-many times.

Decommitment phase. P sends b to P' and gives a witness indistinguishable proof that (1) C is a commitment to b, or (2) $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple. Upon successful completion of this step, P' outputs $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, P, P', b)$.

3.1 Proof Intuition

The intuition underlying the security of the scheme is as follows. We need to argue that for any real-world adversary $\mathcal{A}$ (interacting with parties running the above protocol in the $\mathcal{F}_{\mathsf{wrap}}$-hybrid model), there exists an ideal-model simulator S (running in the $\mathcal{F}_{\mathsf{mcom}}$-hybrid model), such that no PPT $\mathcal{Z}$ can distinguish whether it is interacting with $\mathcal{A}$ or with S. When party P is honest and party P' is malicious, the simulator S will be unable to “rewind” P' (specifically, in the interaction of P' with the token that S must provides on behalf of P), and so the simulator cannot “force” the value of $\mathsf{tuple}_{P \to P'}$ to some desired value. On the other hand, an information-theoretic argument shows that, with all but negligible probability, a value $\mathsf{tuple}_{P \to P'}$ obtained by an interaction of P' with P's token is always a *random tuple* regardless of the behavior of P' (note that P' might interact with the token provided by P polynomially-many times, even though it is supposed to interact with it only once). Security of the signature scheme used (within the token) on behalf of the honest party P implies that P' can only send a value $\mathsf{tuple}_{P \to P'}$ that was output by the token. The upshot is that, with all but negligible probability, a value $\mathsf{tuple}_{P \to P'}$ used by P in step 4 of the protocol will be a random tuple.

On the other hand, continuing to assume that P is honest and P' is malicious, S *can* force the value of $\mathsf{tuple}_{P' \to P}$ to any desired value in the following way. By simulating $\mathcal{A}$'s access to the $\mathcal{F}_{\mathsf{wrap}}$ functionality, S obtains from $\mathcal{A}$ the code $M_{P'}$ that is "placed" in the token that $\mathcal{A}$ provides (on behalf of the malicious party P') to the honest party P. By rewinding $M_{P'}$, it is possible for S to "force" the output $\mathsf{tuple}_{P' \to P}$ to, in particular, a (random) *Diffie-Hellman tuple* (for which it knows the necessary discrete logarithms evidencing this fact). Under the assumption that Diffie-Hellman tuples and random tuples are indistinguishable, this difference will not be detectable to $\mathcal{A}$ or $\mathcal{Z}$. The upshot is that S can set $\mathsf{tuple}_{P' \to P}$ to be a Diffie-Hellman tuple in an undetectable manner.

Given the above, simulation follows in a fairly straightforward manner. Say the honest party P is committing to some value. The simulator, who does not yet know the value being committed to, will simply set C to be a commitment to "garbage" while choosing the elements of com uniformly at random. Note that Com here is perfectly hiding since $\mathsf{tuple}_{P \to P'}$ is a random tuple, so this aspect of the simulation is fine. Furthermore, S can give a successful witness indistinguishable proof that it prepared the commitments correctly since $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple (and S knows an appropriate witness to this fact).

In the decommitment phase, when S learns the committed value, it can simply send this value and again give a successful witness indistinguishable proof that it acted correctly (using again the fact that $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple with discrete logarithm known to S).

The second case to consider is when the honest party P is the receiver. Say the malicious sender P' sends values C and com in step 4, and also gives a successful witness indistinguishable proof in that round. Since $\mathsf{tuple}_{P \to P'}$ is a random tuple, this means that (with all but negligible probability) C and com are indeed commitments to the same value. Furthermore, since $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple (with the appropriate discrete logarithm known to S), it is possible for S to extract the committed value b from com. Arguing similarly shows that in the decommitment phase P' will only be able to successfully decommit to the value b thus extracted.

Further details are provided in the following section.

3.2 Proof of Security

In this section, we sketch the proof that the protocol given earlier securely realizes the $\mathcal{F}_{\mathsf{mcom}}$ functionality. Let $\mathcal{A}$ be a static adversary interacting with parties running the above protocol in the $\mathcal{F}_{\mathsf{wrap}}$-hybrid model. We describe an ideal-model simulator S running in the $\mathcal{F}_{\mathsf{mcom}}$-hybrid model, such that no PPT environment $\mathcal{Z}$ can distinguish whether it is interacting with $\mathcal{A}$ or with S.

S runs an internal copy of $\mathcal{A}$, forwarding all messages from $\mathcal{Z}$ to $\mathcal{A}$ and vice versa. We now specify the actions of S in response to messages received from $\mathcal{F}_{\mathsf{mcom}}$:

Initialization. When a commitment is about to be carried out between parties P, P' for the first time, the simulator S does the following: say P is honest and

P' is corrupted (situations when both parties are corrupted or both parties are honest are easy to simulate).

1. Adversary $\mathcal{A}$ submits a message of the form $(\mathsf{create}, \mathsf{sid}, P', P, M)$ to the (simulated copy of the) $\mathcal{F}_{\mathsf{wrap}}$ functionality on behalf of P', and this message is intercepted by S. Simulator S chooses coins for M at random and runs an honest execution (on behalf of P) with M. If this leads to an abort on the part of P, then no further action is needed. Otherwise, in the standard way, S "rewinds" M and tries to generate an execution in which the output $\mathsf{tuple}_{P' \to P}$ is a (randomly-chosen) Diffie-Hellman tuple, with discrete logarithm known to S. Using standard techniques and assuming the hardness of the decisional Diffie-Hellman problem, this can be done with all but negligible probability in expected polynomial time.
2. S, simulating the $\mathcal{F}_{\mathsf{wrap}}$ functionality, sends the message $(\mathsf{create}, \mathsf{sid}, P)$ to P'. It then runs an honest execution of the token functionality with $\mathcal{A}$ (who is acting on behalf of P'). We stress that S does no rewinding here — indeed, it cannot since it is not given the ability to rewind $\mathcal{A}$ (or, equivalently, $\mathcal{Z}$). S continues to simulate the actions of an honestly-generated token as many times as $\mathcal{A}$ chooses (note that $\mathcal{A}$ may even request further interactions at some later point in time).

Commitment when the sender is corrupted. Say S receives a message $(\mathsf{receipt}, \mathsf{sid}, \mathsf{cid}, P', P)$, the initialization as described above has already been carried out, and the sender P' is corrupted but the receiver P is honest. S begins by sending the value $\mathsf{tuple}_{P' \to P}$ and the corresponding signature (generated as discussed above) to P'. Then S receives values $\mathsf{tuple}_{P \to P'}$ and $\sigma_{P \to P'}$ from P' (this corresponds to step 3 of the commitment phase). If the signature does not verify, then P can abort and no further action is needed. If the signature verifies but $\mathsf{tuple}_{P \to P'}$ was not generated in one of the executions of P' in the initialization phase described above, then S aborts. (This does not correspond to a legal action in the real world, but occurs with only negligible probability by security of the signature scheme.) Otherwise, in the following round S will receive values C and com from P'. It then acts as an honest receiver in the witness indistinguishable proof given by P'. If the proof fails, P will again abort as in the real world. Otherwise, S extracts the committed bit b from com (this is possible since $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple) and sends $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid}, P', P, b)$ to $\mathcal{F}_{\mathsf{mcom}}$ (on behalf of corrupted party P').

In the decommitment phase, S again acts an an honest verifier. If the proof fails, then no further action is required. Otherwise, assuming the bit b sent by P' in this phase matches the bit extracted by S in the commitment phase, S simply sends $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, P, P')$ to $\mathcal{F}_{\mathsf{mcom}}$. The other possibility is that the proof succeeds but the bit b is *different*; however, as argued informally in the previous section, this will occur with only negligible probability.

Commitment when the receiver is corrupted. Say S receives a notification $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid}, P, P')$ from $\mathcal{F}_{\mathsf{mcom}}$ that the honest party P has committed to a bit, and the initialization described earlier has already been carried out. S begins by sending the value $\mathsf{tuple}_{P' \to P}$ and the corresponding signature to P'. Then S receives values $\mathsf{tuple}_{P \to P'}$ and $\sigma_{P \to P'}$ from P' (this corresponds to step 3 of the commitment phase). If the signature does not verify, then P can abort and no further action is needed. If the signature verifies but $\mathsf{tuple}_{P \to P'}$ was not generated in one of the executions of P' in the initialization phase described above, then S aborts. (This does not correspond to a legal action in the real world, but occurs with only negligible probability by security of the signature scheme.) Otherwise, S proceeds as follows: it computes C as a commitment to the all-0 string, and sets the two components of commitment com to random elements of the appropriate group. It sends these values to P' and then acts as the honest prover in the witness indistinguishable proof, but using the witness (that it knows) for the fact that $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple.

When S later receives notification $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, P, P', b)$ that P has opened to the bit b, the simulator simply sends this value b and then, again, acts as the honest prover in the witness indistinguishable proof, but using the witness (that it knows) for the fact that $\mathsf{tuple}_{P' \to P}$ is a Diffie-Hellman tuple.

We defer to the full version of this paper the details of the proof that S as described above provides a good simulation of $\mathcal{A}$.

4 Conclusions and Future Directions

UC multi-party computation is impossible without some extension to the so-called "plain model." We now know of a variety of extensions, or "setup assumptions," that enable this impossibility result to be circumvented. An important direction of research is to find *realistic* setup assumptions that could feasibly be implemented and used. The suggestion made in this paper is to consider physical assumptions instead of (or possibly in addition to) trust-based assumptions. A particular example based on tamper-proof hardware was proposed, and shown to be sufficient for realizing UC multi-party computation.

Some intriguing questions are left open by this work. Of course, alternate (weaker?) models of tamper-proof hardware could be explored in an effort to obtain easier-to-realize conditions under which UC multi-party computation exists. One interesting possibility here is to use tamper-*evident* tokens (that could be returned to their creator at some intermediate point of the protocol) in place of tamper-*resistant* ones. (This idea was suggested by an anonymous referee.) Coming back to the model proposed here, it would be nice to show a protocol secure against adaptive adversaries, and it would be especially gratifying to construct a protocol based on general assumptions. (It is not hard to see that the protocol can be based on a variety of standard number-theoretic assumptions other than the DDH assumption.)

Acknowledgments

I am very grateful to the anonymous referees for their supportive comments and helpful suggestions.

References

1. B. Barak, R. Canetti, Y. Lindell, R. Pass, and Tal Rabin. Secure Computation Without Authentication. Crypto 2005.
2. B. Barak, R. Canetti, J.B. Nielsen, and R. Pass. Universally Composable Protocols with Relaxed Set-Up Assumptions. FOCS 2004.
3. C. Bennett and G. Brassard. Quantum Cryptography: Public Key Distribution and Coin Tossing. Intl. Conf. on Computers, Systems, and Signal Processing, 1984.
4. M. Blum, P. Feldman, and S. Micali. Non-Interactive Zero-Knowledge and its Applications. STOC '88.
5. S. Brands. Untraceable Off-line Cash in Wallets with Observers. Crypto '93.
6. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. FOCS 2001. Full version available at `http://eprint.iacr.org/2000/067`.
7. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally Composable Security with Global Setup. TCC 2007.
8. R. Canetti and M. Fischlin. Universally Composable Commitments. Crypto 2001.
9. R. Canetti, E. Kushilevitz, and Y. Lindell. On the Limitations of Universally Composable Two-Party Computation Without Set-Up Assumptions. *J. Cryptology* 19(2): 135–167, 2006.
10. R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. STOC 2002. Full version available at `http://eprint.iacr.org/2002/140`.
11. D. Chaum and T. Pedersen. Wallet Databases with Observers. Crypto '92.
12. J. Considine, M. Fitzi, M. Franklin, L.A. Levin, U. Maurer, and D. Metcalf. Byzantine Agreement Given Partial Broadcast. *J. Cryptology* 18(3): 191–217, 2005.
13. R. Cramer and T. Pedersen. Improved Privacy in Wallets with Observers. Eurocrypt '93.
14. I. Csiszár and J. Körner. Broadcast Channels with Confidential Messages. *IEEE Trans. Info. Theory* 24(3): 339–348, 1978.
15. D. Hofheinz, J. Müller-Quade, and D. Unruh. Universally Composable Zero-Knowledge Arguments and Commitments from Signature Cards. 5th Central European Conference on Cryptology, 2005. A version is available at `http://homepages.cwi.nl/~hofheinz/card.pdf`.
16. U. Maurer Secret Key Agreement by Public Discussion from Common Information. *IEEE Trans. Info. Theory* 39(3): 733–742, 1993.
17. M. Naor. On Cryptographic Assumptions and Challenges. Crypto 2003.
18. M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM* 27(2): 228–234, 1980.
19. T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. Crypto '91.
20. B. Pfitzmann and M. Waidner. Composition and Integrity Preservation of Secure Reactive Systems. ACM CCCS 2000.
21. A.D. Wyner. The Wire-Tap Channel. *Bell System Technical Journal* 54(8): 1355–1387, 1975.