New Constructions for UC Secure Computation using Tamper-proof Hardware

Nishanth Chandran*, Vipul Goyal**, and Amit Sahai***

Department of Computer Science, UCLA
{vipul, ryan, rafail, sahai}@cs.ucla.edu

Abstract. The Universal Composability framework was introduced by Canetti to study the security of protocols which are concurrently executed with other protocols in a network environment. Unfortunately it was shown that in the so called plain model, a large class of functionalities cannot be securely realized. These severe impossibility results motivated the study of other models involving some sort of setup assumptions, where general positive results can be obtained. Until recently, all the setup assumptions which were proposed required some trusted third party (or parties).

Katz recently proposed using a *physical setup* to avoid such trusted setup assumptions. In his model, the physical setup phase includes the parties exchanging tamper proof hardware tokens implementing some functionality. The tamper proof hardware is modeled so as to assume that the receiver of the token can do nothing more than observe its input/output characteristics. It is further assumed that the sender *knows* the program code of the hardware token which it distributed. Based on the DDH assumption, Katz gave general positive results for universally composable multi-party computation tolerating any number of dishonest parties making this model quite attractive.

In this paper, we present new constructions for UC secure computation using tamper proof hardware (in a stronger model). Our results represent an improvement over the results of Katz in several directions using substantially different techniques. Interestingly, our security proofs do not rely on being able to rewind the hardware tokens created by malicious parties. This means that we are able to relax the assumptions that the parties *know* the code of the hardware token which they distributed. This allows us to model real life attacks where, for example, a party may simply pass on the token obtained from one party to the other without actually knowing its functionality. Furthermore, our construction models the interaction with the tamper-resistant hardware as a simple request-reply protocol. Thus, we show that the hardware tokens used in our construction can be *resettable*. In fact, it suffices to use token

* Research supported in part by grants listed below.

** Research supported in part by grants listed below.

*** This research was supported in part by NSF ITR and Cybertrust programs (including grants 0627781, 0456717, 0716389, and 0205594), a subgrant from SRI as part of the Army Cyber-TA program, an equipment grant from Intel, an Okawa Research Award, and an Alfred P. Sloan Foundation Research Fellowship.

which are completely stateless (and thus cannot execute a multi-round protocol). Our protocol is also based on general assumptions (namely enhanced trapdoor permutations).

1 Introduction

The universal composability (UC) framework of security, introduced by Canetti [Can01], provides a model for security when protocols are executed multiple times in a network where other protocols may also be simultaneously executed. Canetti showed that any polynomial time computable multi-party functionality can be realized in this setting when a strict majority of the players are honest. Canetti and Fischlin [CF01] then showed that without an honest majority of players, there exists functionalities that cannot be securely realized in this framework. Canetti, Kushilevitz and Lindell [CKL06] later characterized the two-party functionalities that cannot be securely realized in the UC model ruling out almost all non-trivial functions. These impossibility results are in a model without any setup assumptions (referred to as the "plain" model). These results can be bypassed if one assumes a setup in the network. Canetti and Fischilin suggest the use of common reference string (CRS) and this turns out to be a sufficient condition for UC-secure multi-party computation for any polynomial time functionality, for any number of dishonest parties [CLOS02]. Some other "setup" assumptions suggested have been trusted "public-key registration services" [BCNP04,CDPW07a], government issued signature cards [HMQU05] and so on.

UC Secure Computation based on Tamper Proof Hardware. Recently, Katz [Kat07] introduced the model of tamper resistant hardware as a setup assumption for universally composable multi-party computation. An important attraction of this model is that it eliminates the need to trust a party, and instead relies on a physical assumption. In this model, a party P creates a hardware token implementing a functionality and sends this token to party P'. Given this token, P' can do nothing more than observe the input/output characteristics of the functionality. Based on the DDH assumption, Katz gave general feasibility results for universally composable multi-party computation tolerating any number of dishonest parties.

Our Contributions. In this paper, we improve the results of Katz in several directions using completely different techniques. Our results can be summarized as follows:

– Knowing the Code: A central assumption made by Katz [Kat07] is that all parties (including the malicious ones) *know* the program code of the hardware token which they distributed. This assumption is precisely the source of extra power which the simulator gets in the security proofs [Kat07]. The simulator gets the power of *rewinding the hardware token* which is vital for

the security proofs to go through. However we argue that this assumption might be undesirable in practice. For example, it does not capture real life adversaries who may simply pass on hardware tokens obtained from one party to another. As noted by Katz [Kat07], such attacks may potentially be prevented by making the creator of a token easily identifiable (e.g., the token could output the identity of the creator on certain fixed input). However, we note that a non-sophisticated fix of this type might to susceptible to attacks where a malicious party builds a *wrapper* around the received token to create a new token and passes it on to other parties. Such a wrapper would use the token inside it in a black-box way while trying to answer the user queries. Secondly, one can imagine more sophisticated attacks where tokens of one type received as part of one protocols may be used as tokens of some other type in other protocols. Thus, while it may be possible to design constructions based on this assumption, it seems like significant additional analysis might be needed to show that this assumption holds.
 We relax this assumption in this work. In other words, we make no assumptions on how malicious parties create the hardware token which they distribute.
- Resettability of the token: The security of the construction in [Kat07] also relies on the ability of the tamper-resistant hardware to maintain state (even when, for example, the power supply is cut off)[1]. In particular, the parties need to execute a two-round interactive protocol with the tamper-resistant hardware. It is explicitly assumed that the hardware cannot be *reset* [CGGM00]. In contrast, our construction models the interaction with the tamper-resistant hardware as a simple one round request-reply protocol. Thus, we are able to show that the hardware tokens used in our construction can be *resettable*. In fact, it suffices to use token which are completely stateless (and thus cannot even execute a multi-round protocol). We argue that relaxing this assumption about the capability of the tamper resistant tokens is desirable and may bring down their cost considerably.
- Cryptographic Assumptions: An open problem left in [Kat07] was to construct a protocol in this model which is based on *general assumptions*. We settle this problem by presenting a construction which is based on enhanced trapdoor permutations previously used in CLOS [CLOS02] and other works.

Our communication model for the token also has an interesting technical difference from the one in [Kat07]. In [Kat07], it is assumed that once P creates a hardware token and hands it over to P', then P cannot send any messages to the token (but can receives messages from it). We require the opposite assumption; once the token has been handed to P', it cannot send any messages to P (but can potentially receive messages from it). It is easy to see that if the communication

[1] As Katz [Kat07] noted, this assumption can be relaxed if the token has an inbuilt source of randomness and thus messages sent by the token in the protocol are different in different execution (even if the other party is sending the same messages). Note that a true randomness source is needed to relax this assumption and cryptographic techniques such as pseudo random functions do not suffice.

is allowed in both directions, then this model degenerates to the plain model which is the subject of severe impossibility results [CF01,CKL06].

Our Techniques. Recall that all the participating parties exchange tamper proof hardware tokens with each other before the protocols starts. To execute the protocol, the parties will presumably make queries to the tokens received from other parties. We observe that the simulator (in the proof of security) can be given *access to all the queries which any dishonest party makes to a token designed by an honest party.* Our first idea to exploit this extra power (and make the simulator non-rewinding) is to extract the inputs of the dishonest parties as follows. If party P_1 wants to commit to its input to party P_2, it will first have to feed the opening to the commitment to the token provided by P_2 which will output a signature on the commitment (certifying that it indeed saw the opening). One may observe that this is very close in spirit to how proofs are done in the Random Oracle model. One problem which we face is that P_1 cannot give a signature obtained from the token directly to P_2 (since these signatures can potentially help establish a covert communication channel between the token and P_2). Thus, the party P_1 instead gives a commitment to the signature obtained (and will later prove that this commitment is a commitment to valid signature).

While the above basic idea is simple and elegant, significant more work is required to turn it into a construction that achieves our main goals (in a way that the construction relies only on general assumptions). The first issue we face is: how to prove that the commitment given is a commitment of a valid signature? While executing a UC commitment scheme, P_1 might be interacting with multiple parties at the same time. We use concurrent zero-knowledge (ZK) proofs [DNS98,PRS02] for this purpose. Although concurrent ZK proofs are not directly usable as building blocks in larger protocols (since they are secure only under concurrent *self composition* rather than general composition), we show that they can be used as a building block in our case by presenting a direct analysis of the resulting scheme to prove its security (under concurrent attacks).

The most difficult issue which we face is: how to prove that a dishonest P_1 cannot commit to a valid signature without actually making a query to P_2's token. This is because if P_1 commits to a valid signature (and even gives a proof of knowledge of the commitment) without making a query to the token, the UC-simulator cannot rewind P_1 to extract this signature (and contradict security of the underlying signature scheme). We get around this issue by showing that the analysis of this case can be separated from the UC-Simulator. In a separate *extraction abort lemma* proven "outside the UC framework", we show that if this case happens, the Environment has the capability to forge signatures (in other words, we rewind the environment and extract a forged signature). Thus, we reduce the failure probability of our simulator to the probability with which the signatures can be forged. Similarly, we have a *decommitment abort lemma* proven outside the UC framework where we reduce the success probability of an adversary opening to a different string than the one committed to (in a UC commitment scheme we construct) to the soundness of an underlying (sequentially secure) zero knowledge proof. Other problems that we deal with are:

the issue of selective abort by the hardware token (where the token refuses to give a valid signature for some particular inputs only) and the issue of equivocating the commitment while keeping the UC-Simulator straightline.

We are able to incorporate all the above ideas into a construction that achieves the multiple commitment functionality in the UC framework. We remark that in the end, the analysis of our construction is admittedly somewhat complex. While one can consider alternative approaches to how a device would extract, several problems like the issue of selective abort (which was simpler to deal with in our approach) again seem to imply that the final solution (which would take care of all these problems) will be no simpler.

Concurrent Independent Work. Independent of our work, Damgard et al [DNW07] proposed a new construction for UC secure computation in the tamper proof hardware model. The main thrust of their work seems to obtain a scheme where the hardware tokens only need to be *partially isolated*. In other words, there exists a pre-defined threshold on the number of bits that the token can exchange with the outside world (potentially in both directions). Their construction is also based on general assumptions (albeit their assumptions are still stronger than ours).

Damgard et al [DNW07] however do not solve the main problems addressed by this work. In particular, their work is in the same *rewinding based simulator* paradigm as Katz [Kat07] and thus requires the same assumption that the sender is aware of the program code of the hardware token which it distributed. Furthermore, the security of their construction relies upon the assumption that the hardware token is able to keep state (i.e., is not resettable).

2 Our Model

Our model is a modification of the model in [Kat07]. The central modifications we need are to allow for adversaries who may supply hardware tokens to other parties without knowing the code of the functionality implemented by the hardware token. To model adversaries who give out tokens without actually "knowing" the code of the functionality of the tokens, we consider an ideal functionality $\mathcal{F}_{Adv}$ that models the adversarial procedure used to create these tokens. The security of our protocol will be defined over all probabilistic polynomial time (PPT) adversaries $\mathcal{F}_{Adv}$. The ideal functionality $\mathcal{F}_{wrap}$ implements the tamper-resistant hardware as in [Kat07].

We first formally define the $\mathcal{F}_{wrap}$ functionality which is a modification of the $\mathcal{F}_{wrap}$ functionality of [Kat07]. This functionality formalizes the intuition that an honest user can create a hardware token T_F implementing any polynomial time functionality, but an adversary given the token T_F can do no more than observe its input/output characteristics. $\mathcal{F}_{wrap}$ models the hardware token (sent by P_i to P_j) encapsulating a functionality M_{ij}. The only changes from [Kat07] we make is that M_{ij} is now an Oracle machine (instead of a 2-round interactive Turing machine) and does not require any externally supplied randomness. $\mathcal{F}_{wrap}$ models the following sequence of events: (1) a party P_i (also known as *creator*) takes software implementing a particular functionality M_{ij} and seals this software into a tamper-resistant hardware token, (2) The creator then gives this token to another party P_j (also known as the *receiver*) who can use the hardware token as a black-box implementing M_{ij}. Figure 1 shows the formal description of $\mathcal{F}_{wrap}$ based on an algorithm M_{ij} (modified from [Kat07]). Note that M_{ij} could make black box calls to other tokens implementing M_{xy} (to model the tokens created by an adversarial party) in a way that the circularity problems are avoided.

$\mathcal{F}_{wrap}$ is parameterized by a polynomial p and an implicit security parameter k. There are 2 main procedures:

Creation. Upon receiving $(create, sid, P_i, P_j, M_{ij})$ from P_i or from $\mathcal{F}_{Adv}$, where P_j is another user in the system and M_{ij} is an Oracle machine, do:

1. Send $(create, sid, P_i, P_j)$ to P_j.
2. If there is no tuple of the form $(P_i, P_j, \star)$ stored, then store (P_i, P_j, M_{ij}).

Execution. Upon receiving (run, sid, P_i, msg) from P_j, find the unique stored tuple (P_i, P_j, M_{ij}) (if no such tuple exists, then do nothing). Run M_{ij} with input msg for at most $p(k)$ steps and let out be the response (set $out = \perp$ if M_{ij} does not respond in the allotted time). Send (sid, P_i, out) to P_j.

Fig. 1. The $\mathcal{F}_{wrap}$ functionality

We now formally describe the Ideal/Real world for multi-party computation in the tamper-proof hardware model. Let there be n parties $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ (P_i holding input x_i) who wish to compute a function $f(x_1, x_2, \cdots, x_n)$. Let the adversarial parties be denoted by $\mathcal{M} \subset \mathcal{P}$ and the honest parties be denoted by $\mathcal{H} = \mathcal{P} - \mathcal{M}$. We consider only static adversaries. As noted before, to model adversaries who give out tokens without actually "knowing" the code of the functionality of the tokens, we consider an ideal functionality $\mathcal{F}_{Adv}$ that models the adversarial procedure used to create these tokens. $\mathcal{F}$ is the ideal functionality that computes the function f that the parties $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ wish to compute, while $\mathcal{F}_{wrap}$ (as discussed earlier) models the tamper-resistant device.

Real World. Our real world is the $(\mathcal{F}_{Adv}, \mathcal{F}_{wrap})$-hybrid world. In the real world, when a party P_i begins a protocol with another party P_j it exchanges a hardware token with P_j. We note that this exchange of token need be done only once in the protocol. This is modeled as follows. If P_i is malicious, then P_i sends arbitrary messages to $\mathcal{F}_{Adv}$ functionality ($\mathcal{F}_{Adv}$ could use this information for the code creation of the adversarial token to be sent to P_j). At the end of this interaction, $\mathcal{F}_{Adv}$ sends a program code (corresponding to the token that is to be given to P_j) to $\mathcal{F}_{wrap}$. This program code can make black box calls to tokens of other (possibly honest) parties. If P_i is honest, then P_i sends a program code directly to $\mathcal{F}_{wrap}$ that will serve as the code for the hardware token to be sent to P_j. During protocol execution, all queries made to tamper-resistant hardware tokens are made to the $\mathcal{F}_{wrap}$ functionality. The parties execute the protocol and compute the function $f(x_1, x_2, \cdots, x_n)$.

IDEAL WORLD. The ideal world is the $(\mathcal{F}_{Adv}, \mathcal{F}_{wrap}, \mathcal{F})$-hybrid world. The simulator S simulates the view of the adversarial parties. As in the real world, when a party P_i begins a protocol with party P_j it has to specify the code for the hardware token to be sent to P_j. If P_i is adversarial, then P_i initially sends arbitrary messages to $\mathcal{F}_{Adv}$. $\mathcal{F}_{Adv}$ sends a program code (corresponding to the token that is to be given to P_j) to $\mathcal{F}_{wrap}$. This program code can make black box calls to tokens of other parties. If P_i is honest, then the simulator S generates the program code for the token to be sent to P_j (S does this honestly according to the protocol specifications for creating the program code). S sends this program code to $\mathcal{F}_{wrap}$. When an adversarial party queries a token created by another adversarial party, the simulator S forwards the query to $\mathcal{F}_{wrap}$ and then upon receiving the response from $\mathcal{F}_{wrap}$, forwards it to the querying party. When an adversarial party queries a token created by an honest party, the simulator S replies with the response to the querying party on its own. Honest parties send their inputs to the trusted functionality $\mathcal{F}$. Simulator extracts inputs from adversarial parties and sends them to $\mathcal{F}$. The ideal functionality $\mathcal{F}$ returns the output to all honest parties and to the simulator S who then uses it to complete the simulation for the malicious parties.

Remark. To be able to model an adversary which takes honest party tokens received in one protocols and uses them as subroutines for creating its tokens in some other protocol, we consider the GUC framework introduced by Canetti et al [CDPW07b]. The proofs in this paper can be modified so as to prove that our protocol for a functionality $\mathcal{F}$ $F_{wrap} -$ EUC-realizes $\mathcal{F}$. This ensures that $\mathcal{F}_{wrap}$ has tokens created by honest parties even as part of other protocols.

3 Preliminaries

As in [Kat07], we will show how to securely realize the multiple commitment functionality $\mathcal{F}_{mcom}$ in the $(\mathcal{F}_{Adv}, \mathcal{F}_{wrap})-$ hybrid model for static adversaries. This will imply the feasibility of UC-secure multi-party computation for any well formed functionality ([CF01,CLOS02]). The primitives we need for the construction of the commitment functionality are non-interactive perfectly binding commitments, a secure signature scheme, pseudorandom function and concurrent zero knowledge proofs (that are all implied by one-way permutations [GL89,NY89,HILL99,Gol01,Gol04,DNS98,PRS02]).

Non-interactive perfectly binding bit commitment. We denote the non-interactive perfectly binding commitment to a string or bit a (from [GL89]) by $\mathsf{Com}(a)$. $\mathsf{Open}(\mathsf{Com}(a))$ denotes the opening to the commitment $\mathsf{Com}(a)$ (which includes a as well as the randomness used to create $\mathsf{Com}(a)$).

Secure signature scheme. We use a secure signature scheme (security as defined in [GMR88]) with public key secret key pair (PK, SK) that can be constructed from one-way permutations ([NY89]). By $\sigma_{PK}(m)$ we denote a signature on message m under the public key PK. We denote the verification algorithm by $\mathsf{Verify}(PK, m, \sigma)$ that takes as input a public key PK, message m and purported signature σ on message m. It returns 1 if and only if σ is a valid signature of m under PK.

Concurrent Zero knowledge. Informally, concurrent zero knowledge proofs (introduced by [DNS98]) are zero-knowledge proofs that remain zero knowledge even when executed in the concurrent setting. In the concurrent setting, several protocols may be executed at the same time, with many verifiers talking simultaneously with one or more provers. Adversarial verifiers may interleave executions of different protocols and may base their messages on partial executions of other protocols. We shall use the concurrent zero knowledge protocol of Prabhakaran, Rosen and Sahai [PRS02]. For further details we refer the reader to [PRS02].

4 The Construction

We show how to securely realize the multiple commitment functionality $\mathcal{F}_{mcom}$ in the $(\mathcal{F}_{Adv}, \mathcal{F}_{wrap})-$ hybrid model for all PPT static adversaries and for all PPT $\mathcal{F}_{Adv}$. We will first give a construction that realizes the single commitment functionality in the $(\mathcal{F}_{Adv}, \mathcal{F}_{wrap})-$ hybrid model for static adversaries and then note that this can be extended to realize $\mathcal{F}_{mcom}$. P_1 wishes to commit to a string a (of length n bits) to P_2.

Token Exchange phase. P_2 generates a public-key/secret-key pair (PK, SK) for a secure signature scheme, a seed s for a pseudorandom function $F_s(\cdot)$ and sends a token to P_1 encapsulating the following functionality M_{21}:

– Wait for message $I = (\mathsf{Com}(b), \mathsf{Open}(\mathsf{Com}(b)))$. Check that the opening is a valid opening to the commitment. If so, generate signature $\sigma = \sigma_{PK}(\mathsf{Com}(b))$ and output the signature. The randomness used to create these signatures is obtained from $F_s(I)$.

We note that the token exchange phase can take place any time before P_2 begins a protocol with P_1 and needs to take place only once.

Commitment phase. We denote the protocol in which P_1 commits to a string a (of length n bits) to P_2 by $\mathsf{UC\text{-}Com}(P_1, P_2, a)$. The parties perform the following steps:

1. For every commitment to a string a of length n, P_1 generates n commitments to 0 and n commitments to 1. P_1 interacts with the token sent to it by P_2 and obtains signatures on these $2n$ commitments. In order to commit to the

i^{th} bit of a string a (denoted by a_i), P_1 selects a commitment to either 0 or 1 whose signature it had obtained from the device sent by P_2 (depending on what a_i is).

- We note that P_1 cannot give the hardware token commitments to the bits of a alone and obtain the signatures on these commitments. Doing this would allow P_2's hardware token to perform a selective failure attack. In other words, P_2's hardware could be programmed to respond and output signatures only if some condition is satisfied by the input string a (e.g., all its bits are 0). Thus if P_1 still continues with the protocol, P_2 gains some non-trivial information about a. Hence, P_1 obtains signatures on n commitments to 0 and n commitments to 1 and then selects commitments (and their signatures) according to the string a. This makes sure that the interaction of P_1 with the hardware token is independent of the actual input a.

Let $B_i = \mathsf{Com}(a_i)$ and let the signature obtained by P_1 from the device on this commitment be $\sigma_i = \sigma_{PK}(B_i)$. P_1 now computes a commitment to σ_i for all $1 \leq i \leq n$ denoted by $C_i = \mathsf{Com}(\sigma_i)$.
Let $Com_i = (B_i, C_i)$. Now $A = \mathsf{COM}(a) = \{Com_1, Com_2, ..., Com_n\}$ (in other words, A is the collection of commitments to the bits of a and commitments to the obtained signatures on these commitments). P_1 sends A to P_2.

- Note here that P_1 does not send the obtained signatures directly to P_2, but instead sends a commitment to these signatures. This is because the signatures could have been maliciously generated by the hardware token created by P_2 to leak some information about a.

2. Let w be a witness to the NP statement that for all i, C_i is a commitment to a valid signature of B_i under P_2's public key PK and that B_i is a valid commitment to a bit. More formally, w is a witness to the following NP statement: "L: For all i,
   – There exists a valid opening of B_i to a bit a_i under the commitment scheme $\mathsf{Com}(\cdot)$
   – There exists a valid opening of C_i to a string σ_i under the commitment scheme $\mathsf{Com}(\cdot)$ such that $\mathsf{Verify}(PK, B_i, \sigma_i) = 1$."

   P_1 picks $l(k)$ random pairs $\{(w_0^1, w_1^1), (w_0^2, w_1^2), \cdots, (w_0^{l(k)}, w_1^{l(k)})\}$ ($l(k)$ is a super-logarithmic function in security parameter k) such that for all $1 \leq t \leq l(k), w_0^t \oplus w_1^t = w$. P_1 sends commitments to these $l(k)$ pairs. In other words, P_1 sends $\mathsf{Com}(w_0^t), \mathsf{Com}(w_1^t)$ for all t.
3. P_2 picks $l(k)$ random bits $\{q_1, q_2, \cdots, q_{l(k)}\}$ and sends it to P_1.
4. P_1 opens the commitment $\mathsf{Com}(w_{q_t}^t)$ for all t by sending $\mathsf{Open}(\mathsf{Com}(w_{q_t}^t))$.
5. P_1 now gives a concurrent zero-knowledge proof ([PRS02]) that w is a witness to statement L being true and that $w_0^t \oplus w_1^t = w$ for all t.
   - We use the specific concurrent zero knowledge protocol of [PRS02] as we require indistinguishability of simulated proof from real proof when the NP statement being proven is not fixed, but publicly predictable given the history of the protocol (as noted in [BPS06]).

Decommitment phase. The parties perform the following steps:

1. P_1 sends P_2 the string that was initially committed to. In particular, P_1 sends a to P_2.
   - Note that P_1 does not send the actual opening to the commitment. P_1 will later prove in zero knowledge that a was the string committed to in the commitment phase. This is to allow equivocation of the commitment by the simulator during protocol simulation.
2. We denote the following steps by the protocol $\mathsf{HardwareZK}(P_1, P_2, a)$:
   (a) P_2 picks a string R uniformly at random from $\{0,1\}^{p(k)}$ and executes the commitment protocol $\mathsf{UC\text{-}Com}(P_2, P_1, R)$.
      - P_1 will prove in zero knowledge that a was the string committed to in the commitment phase. Since we require straight-line simulation, the simulator would have to know in advance the challenge queries made by P_2 in this zero knowledge proof. Hence before this zero knowledge proof is given, P_2 commits to his randomness R using the UC-secure commitment protocol.
      - We note that the decommitment to R need not be equivocable by the UC-simulator and hence we avoid having to use the UC-secure decommitment protocol itself, which would have lead to circularity!

   (b) P_1 gives a standard zero knowledge proof that a is the string that was committed to in the commitment phase of the protocol. The randomness used by P_2 in this zero knowledge proof is R and along with every message sent in the zero knowledge protocol, P_2 proves using a standard zero knowledge proof that the message uses randomness according to the string R.
   Denote by R_i and a_i the i^{th} bits of R and a respectively. More formally, the statement P_1 proves to P_2 is "There exists randomness such that for all $i, B_i = \mathsf{Com}(a_i)$, where B_i is as sent in the commitment phase." Let the value $\mathsf{COM}(R)$ sent during $\mathsf{UC\text{-}Com}(P_2, P_1, R)$ be denoted by Z. Note that Z is of the form $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_n, Y_n)\}$ where $X_i = \mathsf{Com}(R_i)$ and Y_i is a commitment to the signature of X_i under P_1's public key. The statement P_2 proves to P_1 is "There exists string R, such that
      – For all i, there exists an opening of X_i to R_i under the commitment scheme $\mathsf{Com}(\cdot)$
      – R was the randomness used to compute this message."

   (c) P_2 accepts the decommitment if and only if the proof given by P_1 was accepted.

5 Security Proofs

5.1 Description of Simulator

In order to prove UC security of the commitment functionality, we will need to construct a straight-line simulator that extracts the committed value in the commitment phase of the protocol and that can equivocate a commitment to a given value in the decommitment phase of the protocol. Below, we describe such a simulator that runs straight-line both while extracting the committed string when interacting with a committer P_1, as well as when equivocating a commitment to a receiver P_2.

Token Exchange phase. In this phase, before a party P_i begins a protocol with P_j, if P_i is honest then the UC-simulator S creates the program code for the token to be created by P_i and sent to P_j (according to the honest token creation protocol) and sends a copy of the program code to $\mathcal{F}_{wrap}$. If P_i is malicious, it creates the token by interacting with $\mathcal{F}_{Adv}$ as described before. We again note that the token creation can be done at any point before P_i begins a protocol with P_j.

Handling token queries. Whenever an adversarial party queries a token created by another adversarial party, the simulator S forwards the request to $\mathcal{F}_{wrap}$. When simulating the view during the adversary's interaction with a token created by an honest party, S generates the response according to the request by the adversarial party and the program code of the token.

For every pair of parties (P_i, P_j) such that $P_i \in \mathcal{M}$ and $P_j \in \mathcal{H}$, S creates a table T_{ij}. When a malicious party P_i queries the token of an honest party P_j, S stores the query in table T_{ij}. In other words, the simulator S builds a list of all the commitments (along with their openings) that the malicious party queries to a token created by an honest party (for getting a signature). We shall show below that no matter how the tokens of malicious parties are created, the malicious parties cannot obtain any information about the inputs of honest parties.

When a malicious party P_i queries the token of a malicious party P_j, S simply forwards the query to $\mathcal{F}_{wrap}$ and forwards the response received from $\mathcal{F}_{wrap}$ back to P_i. We note that these queries can only make black box calls to tokens of honest parties (as malicious tokens can be created only with black box calls to tokens of honest parties). Hence whatever information an adversary can obtain from this query, the adversary could have obtained itself by making a black box query to the token of an honest party. Hence querying this token gives no additional information to an adversary.

Case 1: Committer is corrupted

Commitment Phase: In this case, the simulator S executes the protocol honestly as a receiver in the commitment phase. In more detail:

1. Let $A = \mathsf{COM}(a) = \{Com_1, Com_2, ..., Com_n\}$ according to the commitment protocol described earlier. P_1 sends A to S (Of course, P_1 may not follow the protocol).
2. Let w be a witness to the NP statement that for all i, C_i is a commitment to a valid signature of B_i under P_2's public key PK and that B_i is a valid commitment to a bit.

P_1 picks $l(k)$ random pairs $\{(w_0^1, w_1^1), (w_0^2, w_1^2), \cdots, (w_0^{l(k)}, w_1^{l(k)})\}$ ($l(k)$ is a super-logarithmic function in security parameter k) such that for all $1 \leq t \leq l(k)$, $w_0^t \oplus w_1^t = w$. P_1 sends commitments to these $l(k)$ pairs. In other words, P_1 sends $\mathsf{Com}(w_0^t), \mathsf{Com}(w_1^t)$ for all t.

3. S picks $l(k)$ random bits $\{q_1, q_2, \cdots, q_{l(k)}\}$ and sends it to P_1.
4. P_1 opens the commitment $\mathsf{Com}(w_{q_t}^t)$ for all t by sending $\mathsf{Open}(\mathsf{Com}(w_{q_t}^t))$.
5. P_1 now gives a concurrent zero-knowledge proof (from [PRS02]) that w is a witness to statement L being true and that $w_0^t \oplus w_1^t = w$ for all t.

The simulator S accepts the commitment if it accepts the zero-knowledge proof. If the zero knowledge proof was accepted, S looks up the commitments to the bits of a (i.e., B_i) in the table T_{12}. Note that T_{12} contains a list of all commitments that were queried by P_1 to the token created by honest party P_2. If any of the commitments are not found, then the simulator aborts the simulation. We call this an *Extraction Abort*. By a reduction to the security of the underlying signature scheme, we prove in Lemma 1 that Extraction Abort occurs with negligible probability. If the simulator did not abort, this means that the commitments to the bits of a were queried by P_1 to the device. Hence, the simulator S has already recorded the openings to these commitments and can extract a by looking up the opening of all these commitments B_i's in the table T_{12}.

Decommitment Phase: S follows the decommitment protocol honestly as a receiver. In more detail:

1. P_1 sends S the string a that was initially committed to. Dishonest P_1 may cheat and send $a' \neq a$ to S.
2. S picks a string R uniformly at random from $\{0,1\}^{p(k)}$ and executes the commitment protocol $\mathsf{UC\text{-}Com}(S, P_1, R)$ honestly.
3. P_1 gives a zero knowledge proof that a' is the string that was committed to in the commitment phase of the protocol. The randomness used by S in this zero knowledge proof is R and along with every message sent in the zero knowledge protocol, S proves in zero knowledge that the message uses randomness according to the string R.
4. S accepts the decommitment if the proof given by P_1 was accepted. Upon accepting the decommitment, S checks if a' was the string that was initially committed to in the UC-commitment protocol. If this is not the case, then S aborts. We call this a *Decommit Abort*. We show in Lemma 2 that Decommit Abort occurs with negligible probability.

We note that when the committer is corrupted, the simulator (as the receiver) follows the protocol honestly during protocol simulation and hence the simulated protocol is identical to the real protocol.

Case 2: Receiver is corrupted

Commitment Phase: The UC-simulator does as follows:

1. S sets the string a to be a string whose all the bits are 0 and then sends $A = \mathsf{COM}(a) = \{Com_1, Com_2, ..., Com_n\}$ according to the commitment protocol described earlier.
2. Let w be a witness to the NP statement that for all i, C_i is a commitment to a valid signature of B_i under P_2's public key PK and that B_i is a valid commitment to a bit.
   S picks $l(k)$ random pairs $\{(w_0^1, w_1^1), (w_0^2, w_1^2), \cdots, (w_0^{l(k)}, w_1^{l(k)})\}$ ($l(k)$ is a super-logarithmic function in security parameter k) such that for all $1 \leq t \leq l(k)$, $w_0^t \oplus w_1^t = w$. S sends commitments to these $l(k)$ pairs. In other words, S sends $\mathsf{Com}(w_0^t), \mathsf{Com}(w_1^t)$ for all t.
3. P_2 sends challenge bits $\{q_1, q_2, \cdots, q_{l(k)}\}$ to S.
4. S opens the commitment $\mathsf{Com}(w_{q_t}^t)$ for all t by sending $\mathsf{Open}(\mathsf{Com}(w_{q_t}^t))$.
5. S now gives a concurrent zero-knowledge proof that w is a witness to statement L being true and that $w_0^t \oplus w_1^t = w$ for all t.

Decommitment Phase: The UC-simulator has to equivocate the commitment to some value a' in the decommitment phase. The simulator proceeds as follows:

1. S sends a' to P_2.
2. P_2 picks a string R of length $p(k)$ and executes the commitment protocol $\mathsf{UC\text{-}Com}(P_2, S, R)$. Again, P_2 may not execute the protocol honestly. If this commitment is accepted, the simulator looks up the commitments to the bits of R in the table T_{21}. If any of the commitments are not found, then the simulator does an extraction abort. Otherwise, the simulator has obtained R.
3. The simulator S now has to give a zero knowledge proof that a' is the string that was committed to in the commitment phase of the protocol. Now given R, all of P_2's messages in this zero knowledge proof protocol are deterministic.
   S internally runs the simulation of this zero knowledge protocol (using the simulator S_{zk} for the underlying zero knowledge protocol). It runs the simulation as the verifier in the protocol (using the messages according to randomness R). Note that S can do this by interacting with prover S_{zk} and generating all messages of the verifier using randomness R. S obtains the simulated transcript of this protocol. Let the messages sent by S in this transcript be denoted by $m_1^V, m_2^V, \cdots, m_d^V$ and let the messages sent by S_{zk} (as the prover) in this simulated transcript be $m_1^P, m_2^P, \cdots, m_d^P$.
4. S will "force" this transcript upon P_2. That is, S sends messages to the party P_2 according to the simulated zero knowledge protocol transcript. At step t of the zero knowledge protocol, it sends the message m_t^P to P_2 and expects to receive m_t^V as response .
   Party P_2 is forced to use the randomness R because P_2, along with every message sent in the zero knowledge protocol, has to prove in zero knowledge that the message uses randomness according to the string R. By the soundness property of this zero knowledge proof (given by P_2), if P_2 sends a message that is not according to randomness R, it will fail in the zero knowledge proof.

We show in the full version of the paper [CGS07] that the view of the adversary in the simulation and in the real protocol are computationally indistinguishable in the commitment as well as decommitment phase.

5.2 Abort Lemmas

Lemma 1. *(Extraction Abort)*
Let ϵ denote the probability with which the simulator S aborts the simulation in the commitment phase (say for some session t and some committer $P_i \in \mathcal{M}$ and receiver $P_j \in \mathcal{H}$). Then, ϵ is negligible in k.

Proof. Let s be the total number of commitment sessions in the protocol. Pick at random the t^{th} commitment session between parties P_i and P_j (with $P_i \in \mathcal{M}$ and $P_j \in \mathcal{H}$). We note that with probability $> \frac{\epsilon}{s}$, during the t^{th} session between malicious P_i and honest P_j, the simulator for the first time in the protocol aborted the simulation. This is the commitment session in the protocol that first terminates in an abort by the simulator. We now focus on this particular session between P_i and P_j.

In this commitment protocol, consider the point upto when P_i (after sending $\mathsf{COM}(a)$) gives a commitment to $l(k)$ random pairs of the form (w_0^t, w_1^t) with $w_0^t \oplus w_1^t = w$. Let this point in the protocol be denoted by λ. We note that the probability with which the simulator aborted the simulation for the first time at session t between P_i and P_j given the prefix of the protocol upto λ is still $> \frac{\epsilon}{s}$ (This probability includes the probability with which this prefix happens.). Now, S goes forward in the simulation with malicious P_i in this session. The simulator completes the simulation of this session between P_i and P_j (The simulator might have to simulate sessions between other parties before finishing the simulation of this particular session.). If the simulator runs into an Extraction Abort in some other commitment session, then the simulator simply aborts the simulation as in that case, the t^{th} session between P_i and P_j was not the first time the simulator had to do an Extraction Abort. Similarly, if the simulator runs into a Decommit Abort in some parallel session, then the simulator aborts the simulation in that case as well. If the dishonest party aborts or does not respond in some parallel session, the simulator aborts in that case as well. We note that the probability with which the simulator completes this commitment session between P_i and P_j and then has to do an extraction abort is $> \frac{\epsilon}{s}$.

Upon aborting the t^{th} session between P_i and P_j, the simulator rewinds the environment back to point λ in the protocol. Now, using fresh randomness the simulator simulates this session between P_i and P_j (once again simulating other parallel sessions if needed). The probability with which the simulator completes the simulation of this commitment session and then does an Extraction Abort (using the fresh randomness) is again $> \frac{\epsilon}{s}$. Hence, the probability with which the simulator will abort at the end of the t^{th} session between P_i and P_j in both executions is $> \frac{\epsilon^2}{s^2}$. The probability with which adversary P_i commits to random shares that do not exclusive-or to the witness and then succeeds in giving a false zero knowledge proof is negligible. This follows from the soundness of the concurrent zero-knowledge proof. The probability with which the simulator picked the same randomness in both simulations (and hence failed to extract the witness) is $\frac{1}{2^{l(k)}}$. Hence with probability $> [\frac{\epsilon^2}{s^2}(1-\frac{1}{2^{l(k)}}) - g(k)]$ (where $l(k)$ is a super-logarithmic function in k and $g(k)$ is any negligible function in k), the simulator will extract a valid witness to the statement P_i was proving to P_j in the t^{th} session.

Since the simulator aborted at the end of this session, this means that there exists a commitment $B_f = \mathsf{Com}(a_f)$ made by P_i whose signature $\sigma_{PK_j}(B_f)$ was not queried by P_i to the device created by P_j. Note that the witness of the statement (which P_i was proving to P_j) contains signatures of all commitments made in that session and, in particular, it contains $\sigma_{PK_j}(B_f)$. Hence with probability $> \frac{\epsilon^2}{s^2} - negl(k)$, we get a forgery of a signature in the existential forgery security game with P_j's public verification key PK_j. From the security of the signature scheme, it follows that $\frac{\epsilon^2}{s^2}$ is negligible in the security parameter and hence ϵ is also negligible in k. □

Lemma 2. *(Decommit Abort)*
Let μ denote the probability with which the simulator S aborts the simulation in the decommitment phase (say for some session t and some committer $P_i \in \mathcal{M}$ and receiver $P_j \in \mathcal{H}$). Then, μ is negligible.

Proof. We shall first show that the protocol $\mathsf{HardwareZK}(P_i, S, a)$ is computationally sound in the stand-alone setting. Consider the zero-knowledge proof $\mathsf{HardwareZK}(P_i, S, a)$. The steps in this proof are as follows:

– S picks a string R uniformly at random from $\{0,1\}^{p(k)}$ and executes the commitment protocol $\mathsf{UC\text{-}Com}(S, P_i, R)$ honestly.
– P_i gives a standard zero knowledge proof that a' is the string that was committed to in the commitment phase of the protocol. The randomness used by S in this zero knowledge proof is R and along with every message sent in the zero knowledge protocol, S proves using a standard zero knowledge proof that the message uses randomness according to the string R.
– S accepts the decommitment if the proof given by P_i was accepted.

Through a sequence of hybrid arguments, we will now show that this protocol has computational soundness in the stand-alone setting.

Hybrid H_0: This hybrid is exactly the same as the above protocol.

Hybrid H_1: This hybrid is exactly the same as H_0 except that the simulator will give simulated zero knowledge proofs in the second step (even though it has a witness). Since this proof is zero knowledge in the stand-alone setting, we have that the simulated proof is computationally indistinguishable from the real proof and hence H_1 is computationally indistinguishable from H_0.

Hybrid H_2: Hybrid H_2 to H_4 deal with proving that the commitment scheme

$\mathsf{UC\text{-}Com}$ is computationally hiding in the stand alone setting. Hybrid H_2 is exactly the same as H_1 except that the simulator replaces concurrent zero knowledge proof given in $\mathsf{UC\text{-}Com}(S, P_i, R)$ by a simulated zero knowledge proof. Note that we do not require the concurrency property of the zero knowledge proof here (as we are considering only the stand-alone setting). Hence, it follows from the zero knowledge property of this proof that H_2 is indistinguishable from H_1.

Hybrid H_3: This hybrid is exactly the same as H_2 except that the simulator replaces the commitments to input R in the first step of $\mathsf{UC\text{-}Com}(S, P_i, R)$ to commitments to a value R' (chosen independently at random). It follows from the computational hiding property of these commitments that H_3 is indistinguishable from H_2.

Hybrid H_4: In $\mathsf{UC\text{-}Com}(S, P_i, R)$, the simulator gave a commitment to R in the first step of the protocol. Let w_{old} be a witness to the NP statement that for all i, C_i is a commitment to a valid signature of B_i under P_2's public key PK and that B_i is a valid commitment to a bit. In this case B_i is a commitment to the i^{th} bit of R. The simulator then followed the rest of the protocol according to this commitment. In particular, in the next step of the commitment phase, the simulator committed to random shares w_0^t, w_1^t such that $w_0^t \oplus w_1^t = w_{old}$. Note that in H_3, the commitments B_i were changed to commitments to R'. Hence, we now have a new witness w_{new} that proves that C_i is a commitment to a valid signature of B_i under P_2's public key PK and that B_i is a valid commitment to a bit.

Hybrid H_4 is exactly the same as H_3 except that the simulator changes the commitments to shares of w_{old} (i.e., commitments to w_0^t, w_1^t) to shares such that they exclusive-OR to w_{new}. Note that these commitments are not used anywhere else in the protocol as the simulator uses simulated concurrent zero knowledge proofs in the commitment phase. From the computationally hiding property of the commitments it follows from a standard hybrid argument that H_4 is indistinguishable from H_3.

Hybrid H_5: This hybrid is exactly the same as H_4 except that the simulator replaces the simulated zero knowledge proof in the $\mathsf{UC\text{-}Com}(S, P_i, R)$ protocol to honest concurrent zero knowledge proof. Again since we are only considering the stand-alone setting, it follows from the zero knowledge property of this proof that H_5 is indistinguishable from H_4.

We note that the difference from H_0 to H_5 is that the commitment $\mathsf{UC\text{-}Com}(S, P_i, R)$ has been replaced by $\mathsf{UC\text{-}Com}(S, P_i, R')$. The simulator still uses simulated zero knowledge proof that messages sent as verifier in the zero knowledge proof are according to randomness R. We shall now argue that if an adversary P^* can violate the soundness of the proof system in Hybrid H_5, then we can construct an adversary p^* that will violate the soundness of the underlying standard zero knowledge proof. p^* will act as verifier V in the above simulated protocol with

P^* and as prover p^* in the underlying standard zero knowledge proof with verifier v. p^* as verifier V will initially commit to a random value R to P^* using $\mathsf{UC\text{-}Com}(S, P_i, R)$. V will then forward messages that it receives from P^* to v as messages of the prover p^*. Upon receiving a message from verifier v, p^* will send this message (as verifier V) to P^* along with a simulated zero knowledge proof that the randomness used to construct this message is R' (chosen independently at random). Now, if P^* can violate the soundness of the proof in the simulated protocol, then p^* can violate the soundness of the underlying zero knowledge proof. Thus, the proof in the simulated protocol is sound. By the indistinguishability of Hybrid H_5 from H_0, it follows that the zero knowledge protocol $\mathsf{HardwareZK}(P_i, S, a)$ has computational soundness in the stand-alone setting.

Now, let s be the total number of decommitment sessions in the protocol. Pick at random the t^{th} session between parties P_i and P_j (with $P_i \in \mathcal{M}$ and $P_j \in \mathcal{H}$). We note that with probability $> \frac{\mu}{s}$, during the t^{th} session between malicious P_i and honest P_j, the simulator for the first time in the protocol does a decommit abort. We now focus on this particular session between P_i and P_j. In this decommitment protocol, the decommitter P_i sends value a' as the first message and then executes protocol $\mathsf{HardwareZK}(P_i, S, a)$ with the simulator. We showed stand-alone soundness of $\mathsf{HardwareZK}(P_i, S, a)$. Since soundness is composable, this implies that $\mathsf{HardwareZK}(P_i, S, a)$ is computationally sound in the concurrent setting. Hence, a dishonest decommitter can only decommit to the value initially committed to. We note that while simulating the t^{th} session between P_i and P_j, the simulator might have to simulate other sessions (commitment and decommitment). If the simulator runs into a Decommit Abort in some other session, then the simulator aborts the simulation since then the t^{th} session between P_i and P_j will not be the first time that the simulator does a Decommit Abort. We note that simulator (except with negligible probability) will not run into an Extraction Abort in a parallel session (as argued in Lemma 1). Hence, μ is negligible. $\square$

References

[BCNP04] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *FOCS*, pages 186–195, 2004.

[BPS06] Boaz Barak, Manoj Prabhakaran, and Amit Sahai. Concurrent non-malleable zero knowledge. In *FOCS*, pages 345–354, 2006.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CDPW07a] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, pages 61–85, 2007.

[CDPW07b] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, Lecture Notes in Computer Science, pages 19–40. Springer, 2001.

[CGGM00] Ran Canetti, Oded Goldreich, Shafi Goldwasser, and Silvio Micali. Resettable zero-knowledge (extended abstract). In *STOC*, pages 235–244, 2000.

[CGS07] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for uc secure computation using tamper-proof hardware. Cryptology ePrint Archive, 2007. http://eprint.iacr.org/2007/334.

[CKL06] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *J. Cryptology*, 19(2):135–167, 2006.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. In *STOC*, pages 409–418, 1998.

[DNW07] Ivan Damgaard, Jesper Buus Nielsen, and Daniel Wichs. Universally composable multiparty computation with partially isolated parties. Cryptology ePrint Archive, 2007. http://eprint.iacr.org/2007/332.

[GL89] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *STOC*, pages 25–32, 1989.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, Cambridge, UK, 2001.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications.* Cambridge University Press, Cambridge, UK, 2004.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[HMQU05] Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. Universally composable zero-knowledge arguments and commitments from signature cards. In *5th Central European Conference on Cryptology*, page A version is available at http://homepages.cwi.nl/ hofheinz/card.pdf., 2005.

[Kat07] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, Lecture Notes in Computer Science, pages 115–128. Springer, 2007.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *STOC*, pages 33–43, 1989.

[PRS02] Manoj Prabhakaran, Alon Rosen, and Amit Sahai. Concurrent zero knowledge with logarithmic round-complexity. In *FOCS*, pages 366–375, 2002.