HW-token-based Common Random String Setup

István Vajda
Dept. of Informatics, TU Budapest, Hungary
Email: vajda@hit.bme.hu

Abstract:

In the common random string model, the parties executing a protocol have access to a uniformly random bit string. It is known that under standard intractability assumptions, we can realize any ideal functionality with universally composable (UC) security if a trusted common random string (CrS) setup is available. It was always a question of where this CrS should come from since the parties provably could not compute it themselves. Trust assumptions are required, so minimizing the level of such trust is a fundamentally important task. Our goal is to design a CrS setup protocol under a weakened trust assumption. We present an HW-token-based CrS setup for 2-party cryptographic protocols using a single token only. Our protocol is a UC-secure realization of ideal common random string functionality F_{CrS}. We show the multiple-session security of the protocol and we also consider the multi-party extension of it.

Keywords: Common Random String setup, tamperproof hardware token, UC-security, Sigma protocols

1. Introduction

The traditional approach provides security assessments for stand-alone protocol problems. This approach has serious disadvantages. It can give no security guarantees when executing the protocol in a realistic protocol environment like the Internet or when the protocol is used as a component of a larger system. On the contrary, the Universal Composition (UC) approach provides the advantage that a UC secure protocol maintains its security within any protocol environment and supports modular design and analysis. The cost of such strong guarantees is the requirement of a trusted setup or an honest majority. Under standard cryptographic assumptions, any ideal functionality can be UC-realized with Common random String (CrS) setup. The CRS (Common Reference String)-model (a special version of which is the CrS-model) is a powerful proof-technical tool but the real question is how it can be realized. A typical solution is to involve a trusted third party in the implementation. The question is, whether there is a third party that each party trusts. Especially if the parties have conflicting interests.

Security concerns around the trust are very real. A dishonest „trusted" third party may have the capability to break the security of the protocol and get access to private information or distort the output of honest parties. Therefore, it is a fundamentally important task to minimize the level of trust. Our goal is to design a CrS setup under a weakened trust assumption.

Tamper-proof hardware is already in widespread use in the form of SIM cards, credit cards, or e-passports. Simple cryptographic operations (encryption, digital signature) are performed by them with stored cryptographic keys (e.g., (Hofheinz et.al., 2005).

The tamperproof hardware token model suggested in (Katz, 2007) has a fundamental advantage in solving the trust problem. Each party is only required to trust that its own token is tamper-proof. In all other known setup assumptions, a party has to trust other parties in the system. Following this approach, we design a UC-secure CrS setup for two-party protocols in

the hardware token model (Katz, 2007). We use only a single token (per pair of parties). Our (blank) token is assumed to have a property additional to standard assumptions, however, we believe it is practically implementable. One of the parties creates the token, i.e. loads a functionality into a blank tamperproof token. The token will output a fresh, uniformly chosen binary string with the wanted length even if one of the parties acts maliciously, e.g. even if the creator of the token is malicious. This single-session construction is even stronger as it is UC-secure. Such a high level of security provides several advantages. It securely implements independent uniformly distributed common random strings for the party even when it is executed concurrently with an arbitrary number of other instances of the protocol or other even hostile protocols, i.e. when it is used in general concurrent execution environment. Finally, UC security provides the advantage of modular design and analysis of a protocol that uses a CrS setup. Technically the latter advantage means that you can analyse the protocol in F_{CrS}-hybrid. We show the secure multiple-session extension of the protocol and we consider the multiparty extension too.

The organization of the paper is as follows: Section 2 contains related works. In Section 3, we show the assumptions and definitions we use. Section 4 presents the construction and analysis together of the UC-secure single-session protocol. In Section 5, we show the multiple-session extension of the single-session protocol. In Section 6, we consider the multi-party extension of our protocol. Section 7 contains a summary of the complexity of the protocol.

2. Related works

(Katz, 2007) suggested to base UC-secure computations on the assumption of the existence of tamper-proof hardware with the following properties: Any party (creator) accessing a blank token can construct a token running any polynomial-time functionality. Next, the creator gives the token to its peer, called the user. The trust assumptions against the token are as follows: (1) the sender (creator) of the token cannot communicate with the token after giving it to the user (isolation assumption), and (2) the receiver (user) of the token cannot learn anything about the internal workings or secrets of the token apart from its input/output behavior (tamper-resilience assumption).

There are several known cryptographic constructions in Katz's hardware token model, e.g. (Broadnax et.al, 2021), (Chandran et.al., 2019), (Fischlin et.al., 2011), (Goldwasser et.al.,2008), (Jarvinen et.al., 2010). Katz's pioneering work (Katz, 2007) showed how to construct UC-secure commitments by exchanging two tokens bi-directionally. Parties assume that their token is tamperproof against the other party.

There can be practical scenarios where one of the parties may not be able to create his hardware, while the other party may be more powerful financially or technically (e.g., government authorities, financial institutions). (Moran and Segev, 2008) showed a construction for a single-bit commitment using a single token.

Instead of designing a token-based protocol for a special cryptographic task, we focus on the token-based implementation of a "general-purpose" setup. Though UC-secure constructions are mostly designed with a common reference string setup, the common random string setup is getting increasing attention. In the F_{CRS}-hybrid model, universally composable commitments can be obtained assuming the existence of enhanced trapdoor permutations only (Canetti et.al., 2002). However, in this case, the common reference string is not uniformly distributed. Nevertheless, a uniformly distributed string can be used, under additional cryptographic assumptions. A well-known additional assumption is the existence of dense cryptosystems in

the case of a non-adaptive adversary (Canetti et.al., 2002). Typical examples for constructions in a common random string (CrS) setup are UC-secure non-interactive zero-knowledge (NIZK) proofs, e.g. (Blum et.al, 1988), (Fischlin et.al., 2021), (Groth and Ostrovsky, 2007), (Quach et.al, 2019). Recall that, besides their theoretical importance on their own, NIZKs have found numerous applications in cryptography. For a few examples, NIZK is widely used in group signatures, ring signatures, electronic voting, block-chains, and cryptocurrencies.

Our construction uses a single token only. In this respect, the closest work is [18]. We notice that, in principle, we could use commitment protocol (Moran and Segev, 2008) in a modular realization of the CrS functionality. However, the construction we present in this paper is more efficient since the (Moran and Segev, 2008) protocol presents commitment only to a single bit. Although we immediately admit that, we do achieve this advantage by making a special (albeit technically reasonable) assumption regarding the tamperproof token.

3. Assumptions and definitions

We assume a static malicious corruption adversary. The parties have safe physical access to the token therefore this communication channel can be modelled as a perfectly secure channel. The cryptographic assumptions are a perfectly hiding string commitment and a related Sigma-proof as well as a standard secure (EU-CMA) digital signature (the latter primitive part of the token model (Katz, 2007)). We consider security with abort. The assumptions about the token are detailed subsequently.

3.1 Assumptions about the token

The basic security assumptions against the HW token are isolation and tamper-resilience. The token in the model (Katz, 2007) contains a built-in random source. The token uses this random source during the generation of the creator's protocol messages. The user party uses his own random source (external to the token) when computing his messages.

Additionally, we make the following special, however practically realistic assumption.

When the protocol running in the token computes a new message, the message is first placed in temporary storage. (This communication storage is not accessible by the parties.) The token counts the messages and this number becomes available to the user together with a flag bit as soon as a new message has been stored. This bit indicates the order in which the next two messages are exchanged between the token and the user. If it is 0 then the user must first send his next message to the token, otherwise (flag=1) the token will send the stored message first.

Note such reversed time order of message transmissions does not violate security if the user's next message is an independently chosen random element by the specification. If the user were to abort due to the content of the message from the token, it can do it with a short time delay without revealing any private information to the token (since it sends a random element).

In the specification of the protocol, the protocol messages sent by the token to the user will be marked with the time order of transmission as normal or reversed. Honest creator will load the functionality by the specification. Malicious creators cannot load a functionality into the token such that it sets the order of messages maliciously to its own advantage. Indeed, the malicious creator would like to see the value of the challenge in the Sigma-proof before it generates the first message of the proof. However, it is not possible, since reverse time order means that the creator has already sent its message to the token however, the token will hold it until the user sends the challenge message (i.e. we emulate a synchronous transmission between the parties).

This modification of the protocol execution will give a necessary advantage to the simulator.

3.2 Common random string setup

Most known UC-secure protocols are constructed with a Common Reference String (CRS) setup. For our technical convenience in this paper, we show the functionality in a non-usual formulation, where parties receive their output simultaneously (Fig. 1). Functionality F_{CrS} is a special version of functionality F_{CRS} when the underlying probability distribution is uniform over [0, 1]. An instance of functionality F_{CRS} is identified by the value of session identifier sid.

Functionality F_{CRS}
The functionality is parametrized by a distribution D. It proceeds as follows. When receiving input (CRS, sid) from party P, first verify that sid = (*P*, sid') where *P* is a set of party identifiers, and that P∈*P*, else ignores the input. Ignore any subsequent CRS messages (with identifier sid). Next, if there is no value r is recorded then choose a value $r \leftarrow_r D$. Finally, send a public delayed output (CRS, sid, r) to all parties in set *P*.

Fig. 1: Functionality F_{CRS}

An efficient realization of ideal functionality F_{CRS} is impossible if there is no honest majority of the parties (in set *P),* even in the plain model (Canetti, 2002), (Canetti et.al, 2002). This implies that two parties cannot realize CRS setup in the standard model of cryptography. We have to make trust assumptions, most commonly that an honest, non-corruptible third party provides the CRS string (in particular, the ideal functionality itself is run honestly by the trusted third party in the real system). The risk here is that the two parties fully trust a single external party.

Several solutions are based on an honest majority. The (two) parties trust a group of volunteers to generate a CRS jointly. The group executes a multi-party protocol. A similar approach is the multi-string setup model (Groth and Ostrovsky, 2007), where a group of trusted authorities are involved in the generation of the CRS. The authorities publish coin-tossing strings. The trust model is that there is no trust in any single authority, however, it is assumed that a majority of them generate random strings honestly. A practical problem with this latter approach is that we do not know how many external parties (authorities) is required to achieve a trusted majority, say, with a failure probability of 2^{-100} (a "standard small" value in cryptography). For instance, if we assume a binomial model, where p is the probability that a given external party cheats with its random string, we have no idea about the magnitude of p. It is simply because we have no related mass experimental data to estimate the relative frequency of cheating.

Another direction to relax the trust requirement is to change the model of the third party. In the registered public key setup model (Barak et.al., 2004) parties register correctly generated public keys. However, the obvious question is that who the parties can trust to verify the correctness of the keys.

The CrS is a special case of CRS when distribution D is uniform. Our token-based construction provides secure realization of functionality F_{CrS} under the condition that there is no abort event. This means that the real setup may respond with an abort message instead of outputting the

wanted string. However, under the condition of no abort, the setup produces a common string with the expected distribution.

3.3 Wrapper functionality F_{WRAP}

We want to reduce the assumed level of trust while maintaining the possibility that the simulator can simulate the setup. The tamper-proof hardware was modeled as a wrapper functionality F_{WRAP} in (Katz, 2007) that stores a Turing machine and maintains the state of the machine.

The creator of the token (party P1) invokes an instance of functionality F_{WRAP} by loading a program code (M) into a blank token. Next, the creator gives the token to the user (party P2). The user can access code M only in a black-box manner (tamper-resilience assumption). The creator cannot communicate with the token once it gives it to the user (isolation assumption). Parties P1 and P2 can use the token multiple times. In each new execution of the code, the Turing machine running code M uses fresh random elements.

In our proposed construction, an instance of the multiple-coin tossing protocol will run in the token. This protocol is the direct extension of Blum's single-bit tossing protocol to n-bit strings. In a nutshell, the protocol works as follows: Party P1 sends commitment Com(ρ_1, r) to a random n-bit string ρ_1. Next, P2 replies with a random n-bit string ρ_2, then P1 opens the commitment, and finally, both parties send output ρ_1+ρ_1 (mod 2). We assume that commitment Com is a standard secure perfectly hiding commitment.

Blum's single-bit tossing protocol can be extended even to UC-secure multiple-bit tossing protocols via the commit-and-proof technique shown in (Lindell, 2017). However, in this design the ZK-proofs have to be UC-secure, implying the requirement of an appropriate trusted setup (e.g. CRS setup), which is the very problem we want to solve. In (Groth and Ostrovsky, 2007) first a UC-secure commitment is constructed in a multi-string setup model and such a commitment is used in a UC-secure multiple-bit tossing protocol to generate CrS.

4. Realization of ideal functionality F_{CrS}

4.1 Commit-and-prove

We assume a standard secure perfectly hiding commitment. We use the commit-and–prove technique with a Sigma-proof upgraded to a ZK-proof. Instead of standard decommitment of a commitment value c, i.e., instead of revealing a pair of values (m=committed value, r=random element), just the committed value is revealed, and by using an interactive proof, the committer proves that it knows the corresponding random element (r) such that Com(m, r) equals commitment c.

For instance, consider Pedersen's commitment:

$\mathrm{Com}(r, x) = g^r h^x$,

where there is an underlying public group (G,·) of large order q in which the discrete logarithm is hard and elements g and h are two random public generators. Random secret r is chosen in Z_q, and the committed value x is from any subset of that. A usual implementation is when group G is the prime order q subgroup of Z_p^*, where q=(p-1)/2 prime. Considering the latter implementation user party generates parameters (p, g, h) and sends them to the creator party.

The creator party checks that p and (p-1)/2 = q are prime, that p has appropriate length and that g, h are generators of the order-q subgroup $G \subset Z_p^*$, and aborts if these do not hold.

In the corresponding Sigma-proof (Σ) committer proves that it knows a witness r satisfying relation:

$R=\{((c, x, g, h, G); r): g^r = c \cdot h^{-x}\}$.

i.e., a proof for knowing a discrete logarithm (r).

This commitment is perfectly hiding and computationally binding. A dishonest committer can break the binding property, but only with a negligible probability. Note, if a corrupted committer can break commitment c = Com(r, x), i.e. can compute an (x', r') pair such that x'≠x and c = Com(r', x') then it can provide also a successful proof. Therefore, such an event leads to a simulation failure. (Intuitively, the situation is worse for the attacker, because the attacker wants to calculate a second preimage x' of commitment c corresponding to its target output value.)

4.2 Creation of the token

Functionality M loaded into the token by creator party P_1 ($token_{P1,P2}$) is shown in Fig. 3. It is used for the generation of a single CrS string. Session identifier is sid = (P_1, P_2, sid'). The keys of the underlying cryptographic primitives (public parameters of commitment Com and of the Sigma protocol as well as the keys of digital signature) are the realization of the session identifier. (Multiple-session generalization is discussed in Section 5.)

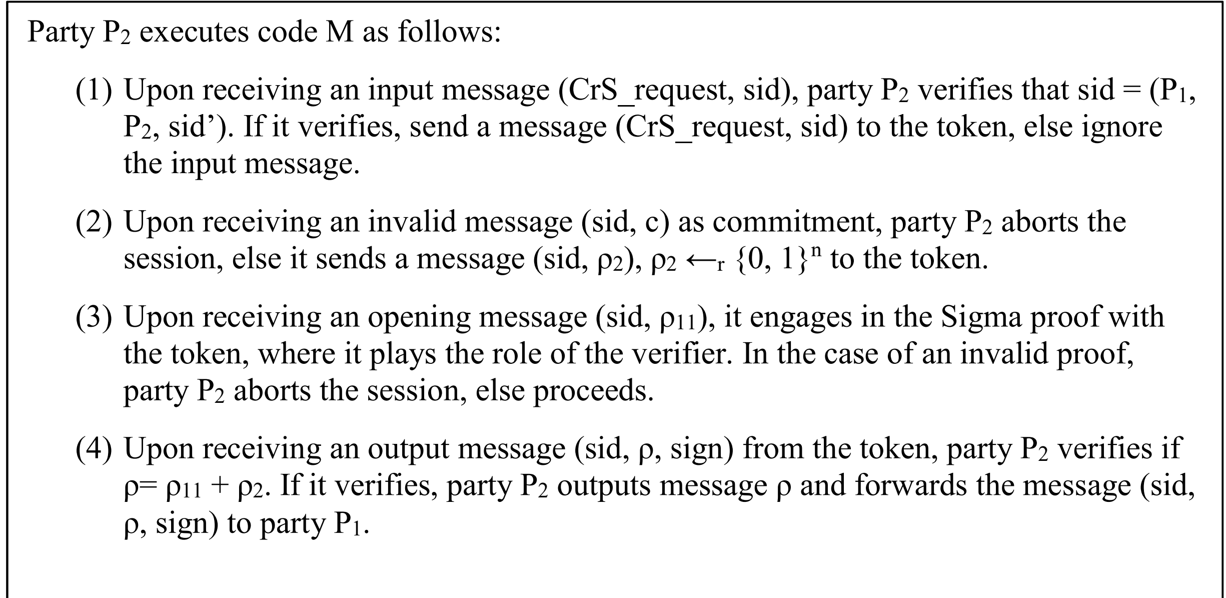
Once the token is created and delivered to user party P_2, this party can interact with it in a black-box manner. This is formalized by allowing P_2 to send messages of its choice to M via the wrapper functionality F_{WRAP}.

> Party P_1 generates (outside of the token) a public-key/secret-key pair (PK; SK) for a secure digital signature scheme, and creates a token by loading into it secret key SK, the public parameters of commitment mapping Com as well as functionality M. Definition of functionality M is as follows:
>
> (0) Upon receiving input message (CrS_request, sid), verify that sid = (P_1, P_2, sid'). If sid is not of that form, then ignore this input, else proceed.
>
> (1) Choose random elements $\rho_1 \in \{0, 1\}^n$ and $r \in \{0, 1\}^{poly(n)}$ and compute commitment $c = Com(\rho_1; r)$. Output commitment (sid, c).
>
> (2) Wait for a message (sid, ρ_2), $\rho_2 \in \{0, 1\}^n$. If no message or invalid message is received then set $\rho_2 = 0^n$, otherwise proceed.
>
> (3) Output committed value (sid, ρ_1).
>
> (4) Execute Σ-protocol in the role of the prover. Prove that ρ_1 is a valid opening of commitment value c for some random element r
>
> (5) Output (sid, ρ, sign), where $\rho=\rho_1 \oplus \rho_2$ and sign=$Sign_{SK}$(sid, ρ) is a digital signature on (sid, ρ) with the signature key SK. Halt.

Fig. 3: The functionality encapsulated in the token.

The order of message transmissions between the token and the user is as follows. The first two messages of the Σ-protocol are in reversed time order in the view of the user, i.e. the user sends the challenge before it sees the first message of the proof.

Party P_2 executes functionality M encapsulated within the token received from creator P_1 as shown in Fig. 4.



Party P_2 executes code M as follows:

(1) Upon receiving an input message (CrS_request, sid), party P_2 verifies that sid = (P_1, P_2, sid'). If it verifies, send a message (CrS_request, sid) to the token, else ignore the input message.

(2) Upon receiving an invalid message (sid, c) as commitment, party P_2 aborts the session, else it sends a message (sid, ρ_2), $\rho_2 \leftarrow_r \{0, 1\}^n$ to the token.

(3) Upon receiving an opening message (sid, ρ_{11}), it engages in the Sigma proof with the token, where it plays the role of the verifier. In the case of an invalid proof, party P_2 aborts the session, else proceeds.

(4) Upon receiving an output message (sid, ρ, sign) from the token, party P_2 verifies if $\rho = \rho_{11} + \rho_2$. If it verifies, party P_2 outputs message ρ and forwards the message (sid, ρ, sign) to party P_1.

Fig. 4: Execution of the token's functionality by party P_2

When party P_1 receives a message (sid, ρ, sign) from party P_2, it verifies the sid and the signature. If the verification is successful then party P_1 outputs message (sid, ρ).

Claim 1: The protocol defined above UC-securely implements ideal functionality F_{CrS}.

4.3 Analysis of the single-session protocol

An overview

The protocol is analysed in the F_{WRAP}-hybrid model. We assume a static corruption adversary. The simulator simulates the wrapper functionality for the corrupted party. We distinguish two cases: the case of a corrupted creator and the case of a corrupted user. Note successful straight-line simulation implies UC-security since parties receive no inputs. We guarantee security assuming the event of no abort. Accordingly, the security analysis is conditioned on the event that no abort happened.

In the first case, the simulator obtains the token code from the *corrupted creator* (adversary), and from then on, the adversary is isolated from the token. Because of this fact, the simulator gets the advantage of *rewinding the token* (since the creator sees no information at all about the details of the execution). The simulator rewinds only once. From the side of the honest party, the simulator has an easy job because there is no private input and random elements can be simulated using the simulator's random tape. Rewinding provides the simulator with the advantage of extracting the committed value ρ_1 and, accordingly, choosing the string $\rho_2 = \rho \oplus \rho_1$.

However, the commitment is (only) computationally binding, and there remains a non-zero (but negligible) probability that the corrupt party cheats when opening the commitment.

In the second case (simulation against a *corrupted user*), the simulator cannot rewind the adversary, more precisely, the simulator cannot rewrite the view of the adversary (corrupted user). Despite this disadvantage, we can simulate it successfully. We give an advantage to the simulator. Having this advantage, the simulator can simulate the Sigma-proof perfectly in case of a corrupted user. The simulator cheats with the message counter, by falsely claiming to the user that the first message of the Σ-proof is ready to be forwarded. The user sends the challenge to the simulated token. Knowing the value of the challenge the simulator can use the SHZVK simulator of the Σ-protocol.

We also note that a corrupt user may attack the protocol successfully also by breaking the EU-CMA property of the digital signature primitive. In order to decouple this problem from the analysis, we do the proof for F_{SIG}-hybrid protocol. The simulator will simulate the ideal signature functionality F_{SIG} functionality. Recall, EU-CMA secure digital signature realizes functionality F_{SIG} UC-securely. Details follow.

The case of a corrupt creator

Simulation:

Ideal-process adversary S simulates a virtual copy of the real-life adversary A (a.k.a. black box adversary) and relays messages of A and the environment Z.

In a nutshell, the beginning steps of the simulation are as follows. The ideal functionality receives input (sid, 1^n) for both parties (the input from honest party P_2 arrives directly from Z, and the input from corrupted party P_1 arrives via simulator S). The ideal functionality sends an output value ρ' first to the simulator (ideal system adversary) by the rushing adversary model. Value ρ' becomes the output constraint for the simulation.

Simulator S simulates the interaction between functionality F_{WRAP} and corruption adversary A. By doing so S has access to the description of a Turing machine M because adversary A sends this code to F_{WRAP}. In details:

1. Adversary A submits a message of the form (*create*, sid, P_1, P_2, M) to the simulated copy of F_{WRAP} functionality on behalf of P_1, and this message is intercepted by S. S gets to know code M.

2. Simulator S chooses coins for M at random and runs an honest execution with M (on behalf of P_2). If this leads to an abort on the part of P_2, then no further action is taken. Otherwise, S tries to force an output value ρ' in the view of the adversary. It attempts to extract the committed value ρ'_1 by sending a random test value ρ'_2. Three different events may happen under the assumption of no abort on the values of an incoming message:

Event E_a: The adversary opens honestly to committed value ρ'_1.

Event E_b: The adversary breaks the binding property and potentially opens to a value ρ''_1 "correlated" with ρ'_2.

Event E_c: The adversary cheats successfully in the Sigma-proof, and in this case adversary may win without breaking the binding property of the commitment.

First, consider event E_a: Since the token is just part of the simulation of F_{WRAP}, and S knows the code the token is executing, S can efficiently rewind it. S rewinds M to step (2) and sends $\rho'_2 = \rho' \oplus \rho'_1$ to M on behalf of the user. In this case, the simulation is successful.

In the case of event E_b: the simulation may fail: the simulator rewinds the code of the token and sends a string with the value $\rho''_2 = \rho' \oplus \rho''_1$ on behalf of the user. The adversary (the code of the token) breaks the binding property and opens to a value "correlated" with ρ''_2, This game of "back and forth" could go on "forever", but our simulator rewinds just once and outputs a "failure" message.

Successful cheating with the proof (event E_c) means the successful opening of the commitment to a string different from the committed value. This is the source of the second type of simulation failure. Fortunately, the probability of event E_c is exponentially small in the length of the challenge string, thereby we can make it arbitrarily small at any fixed value of the security parameter. Event E_c can be considered as kind of statistical error.

3. In case of no failure, the simulator honestly computes the digital signature of value ρ' and sends (sid, ρ', sign) to party P_1. Note S is aware of signature key SK, as part of the description of encapsulated code M.

Analysis of the simulation:

Signature keys (PK, SK) and (PK', SK') are two independently chosen samples from the distribution at the output of the corresponding key setup algorithm within ideal functionality F_{SIG} in the hybrid system and of the key setup of the perfectly simulated F_{SIG} in the ideal system, respectively.

Assuming no failure the joint output of the real system and the ideal system are as follows:

{adversary's output: (ρ, $Sign_{SK}(\rho)$); honest user's output: ρ}

{adversary's output: (ρ', $Sign_{SK'}(\rho')$); honest user's output: ρ'}

where ρ and ρ' are independent uniform random samples from space $\{0,1\}^n$. These joint outputs are indistinguishable.

Now we prove that the probability of the failure of the simulation is negligible. Let denote F as the event of failure, where $F = E_b \cup E_c$. Hence we get $P(F)=P(E_b \cup \overline{E_c})+P(E_c)$, where cheating error $P(E_c)$ can be made exponentially small in the length of the challenge string. Event $E_b \cup \overline{E_c}$ occurs when a corrupt creator can break the binding property of the commitment scheme. This reduction is straightforward. In brief, the target commitment value (computed for uniformly random input) is used as the first message of the protocol and (from the logic of simulation it follows that) failure of simulation can happen only if this commitment can be opened to different committed values, i.e. when the simulator cannot force the wanted output random string. By the assumed computational binding property of the commitment, it follows that probability $P(E_b \cup \overline{E_c})$ is negligible. The upshot is that the probability of failure P(F) is negligible.

The case of a corrupt user

Simulation:

Simulator S simulates the interaction between F_{WRAP}, the corrupted party P_2, and the honest party P_1. We observe that the most the corrupt user can do is to follow the specification or abort on received messages. Firstly, the perfectly hiding property of the commitment implies that the user is forced to choose string ρ_2 honestly. Secondly, by the message order reversion technique, the user is also forced to choose the challenge string honestly within the Sigma sub-protocol.

The simulator starts running a copy of the ideal functionality F_{CrS} and learns the output constraint ρ'. The simulator computes the commitment by executing the code honestly.

In the opening phase, the simulator (in the role of the prover) cheats with the sigma proof (regarding the committed value). It performs a perfect simulation of the Sigma-proof:

The simulator cheats with the ready-message counter, falsely indicating to the corrupted user that the first message of the Σ-protocol is ready to be sent. For this, the user sends the challenge to the token, so the simulator learns it. With this knowledge, the simulator can simulate the first message and the reply message of the proof by using the SHZVK simulator of the Sigma protocol. Recall by the perfect hiding property of the commitment the verifier (dishonest user) has no a priori information about the real committed value (ρ_1) at the start of the Sigma protocol.
Finally, in the knowledge of the signature key, simulator S computes the signature on message ρ'.

Analysis of the simulation:

The analysis below is conditioned on the event that no abort happened. First, we note that the simulation cannot fail.

The joint output of the real system is as follows:

Adversary's view: X=(X1, X2, X3, X4), where

X1=c (=Com(ρ_1; r)),

X2=(ρ_1, ρ_2),

X3 = {view of Sigma proof},

X4= $Sign_{SK}(\rho)$ ($\rho= \rho_1+\rho_2$)

Honest creator's output: ρ

The joint output of the ideal system is as follows:

Adversary's view: X'=(X'1, X'2, X'3, X'4), where

X'1=c' (= Com(r"; r')),

X'2= (ρ'_1, ρ'_2),

X'3= {view of simulated Sigma proof for "committed value" ρ'_1},

X'4 = $Sign_{SK'}(\rho')$ ($\rho'= \rho'_1+\rho'_2$)

Honest creator's output: ρ'

Note, that we only have to focus on the indistinguishability of views X3 and X'3. Recall, the simulation of the Sigma-proof is perfect. It follows the joint outputs are perfectly indistinguishable.

5. Multiple-session extension

Consider the following scenario: We assign tokens to pairs of parties from a set *P* of parties. Tokens run instances of the token-based CRS setup algorithm. These instances may run concurrently. Tokens (within set *P*) share common long-term public parameters of the underlying commitment primitive as well as of the Sigma protocol. The signature keys are chosen fresh in each token. Multiple-session extension provides a more efficient implementation. The corresponding multiple-session CrS ideal functionality, F_{MCrS} is shown in Fig. 4. Note the main difference between functionalities F_{CrS} and F_{MCrS} is that in the case of the

latter functionality arbitrary parties, arbitrary times may use the same instance of F_{MCrS} for the generation of a common random string for them. An instance of functionality F_{MCrS} is identified by the value of identifier sid, an execution of this instance is identified by the value of identifier ssid.

Functionality F_{MCrS}
The functionality is parametrized by a set *P* of parties. It proceeds as follows:

Upon receiving an input message (CrS_request, sid, ssid) from party P_i verify that sid = (*P*, sid'), ssid=(P_i, P_j, sid''), such that $P_i, P_j \in P$. If it does not verify then ignore the input. If there is no value (sid, ssid, r) recorded for some r then choose a value $r \leftarrow_r U$ and store (sid, ssid, r), else ignore the input. Finally, send a public delayed output (CRS, sid, ssid, r) to parties P_i and P_j .

Fig. 4: Multiple-session functionality F_{MCrS}

A little more formally: Functionality F_{MCrS} runs multiple copies of F_{CrS}, where each copy is identified by a sub-session identifier, *ssid*. Upon receiving a message for the copy associated with *ssid*, F_{MCrS} activates the appropriate copy of F_{CRS} (running within F_{MCrS}) and forwards the incoming message to that copy. If no such copy of F_{CRS} exists, then a new copy is invoked and given that *ssid*. Outputs generated by the copies of F_{CRS} are copied to F_{MCrS}'s output.

A sid value is assigned to the set *P* of parties and it is implemented by the public parameters of the commitment primitive and of the Sigma protocol. A ssid=(P_i, P_j, sid'') value differentiates between different pairs of parties using a token and also between multiple executions of the token base algorithm.

The protocol realizing multiple-session CrS functionality F_{MCrs} is shown in Fig. 5.

Party P_1 generates (outside of the token) a public-key/secret-key pair (PK; SK) for a secure digital signature scheme, and creates a token by loading into it a functionality M. The public keys are "hardwired" into functionality M. Party P_1 sets the initial value of ssid. Next, party P_1 gives the token to party P_2.
Functionality M works as follows:

(0) Upon the arrival of message (CrS_request, sid, ssid), update the value of ssid and do Steps 1-5 as follows:

(1) Choose the next random element $\rho_1 \in \{0, 1\}^n$ and compute commitment c = Com(ssid, ρ_1); r). Output commitment (sid, ssid, c).

(2) Wait for a message (sid, ssid, ρ_2), $\rho_2 \in \{0, 1\}^n$. If no message or invalid message is received then set $\rho_2 = 0^n$.

(3) Output committed value (sid, ssid, ρ_1).

(4) Run the Sigma protocol in the role of the prover. Prove that (sid, ssid, ρ_1) is a valid opening of commitment value c.

(5) Output (sid, ssid, ρ, sign), where $\rho=\rho_1 \oplus \rho_2$ and sign is a digital signature on (sid, ssid, ρ) with signature key SK. (Go back to step (0))

Fig. 5: Multiple-session protocol encapsulated in the token by party P_1

Party P_2 executes code M encapsulated within the token received from creator P_1 (see it in Fig. 6).

Party P_2 executes code M encapsulated within the token as follows (shown by steps):

1. Party P_2 starts running code M by sending the message (CrS_request, sid, ssid).
2. Upon receiving an invalid message (sid, ssid, c) as commitment, party P_2 aborts the session, else it sends a message (sid, ssid, ρ_2), $\rho_2 \leftarrow_r \{0, 1\}^n$ to the token.
3. Upon receiving an opening message (sid, ssid, ρ_{11}), it engages into a Sigma proof with party P_1, where it plays the role of the verifier. In the case of an invalid proof, party P_2 aborts the session, else proceeds.
4. Upon receiving an output message (sid, ssid, ρ, sign) from the token, party P_2 verifies if $\rho= \rho_{11} + \rho_2$. If so then it forwards this message to party P_1. A valid signature on (sid, ssid, ρ) is a proof for party P_1 that this value is the output of the $\text{token}_{P1,P2}$ with an identifier (sid, ssid).

Fig. 6: Execution of the multiple-session protocol by party P_2

Claim 2: The multiple-session protocol (defined above) UC-securely implements multi-session CrS functionality F_{MCrS}.

We assume static corruption, i.e. a corruption adversary decides about the corruption of parties before instance F_{MCrS}(sid) starts running. Accordingly, we consider the security of sessions only where one of the parties is corrupted within a session.

Proof :

First, we make two observations, related to the simulation of the multiple execution protocol.

The first is that concurrent sessions (running on different tokens) run “independently”, such that adaptive input selection or malleability attack is not possible. Indeed, the functionality run by the token receives no private input, furthermore, the two protocol messages received from the user party (random elements ρ_2 and the challenge string) are random strings independent from the associated messages computed by the token (the commitment of the creator and the first message of the Sigma protocol, resp.). Furthermore, we can assume that the channel between the user and the token is (physically) perfectly secure (e.g. the token runs within the safe environment of the non-corrupted computing device of the user). In sum, it is sufficient to consider the security of the protocol executed on a token in isolation from the other copies of the token. Accordingly, we can reduce the analysis to a stand-alone token scenario, where we use the same token repeatedly.

The second observation further simplifies the simulation. We observe that a simulator S’ for the s-times repetition scenario can be composed of s-times independent invocation of the single-session simulator S. An (informal) explanation follows.

We recall that all three primitives used in the protocol (commitment, Sigma protocol, digital signature) are secure under multiple execution (a.k.a. repetition). This means that these primitives keep their respective security properties when during the repetition the adversary may also use auxiliary information accumulated from previous executions of the primitive. Our point here is that repeated usage of these primitives via the repeatedly executed protocol does not improve the success probability of breaking the security properties of the primitives. In details:

A corrupt creator cannot choose commitment c adaptively to get a better chance of breaking the computational binding property. The only additional information for the creator (additional to a repetition scenario) is that it sees two independent random samples (ρ_2 and challenge) received from the honest user per session. This “experiment” from the binding-breaking point of view is equivalent to a repetition scenario carried out separately for the primitive alone. (Indeed, the creator itself can simulate the additional information.) Concerning the Sigma-proof, a corrupt creator cannot achieve a soundness error higher than the probability of successful blind guessing of the honestly chosen challenge.

Independently on the number of repetitions, a corrupt user gets zero information about the committed values when it receives commitment messages generated by perfectly hiding commitment mapping (consequently it cannot distort the uniformity and the independence of common random strings).

The point here is that, the adversary cannot use auxiliary information to improve its attack success, and that supports the conclusion that simulation of sessions with a sequence of independent single-session simulators. Accordingly, in the formal analysis, we use s-times independent invocations of the single-session simulator and prove the indistinguishability of joint (s-times) outputs of the real and ideal systems.

A summary of the simulation follows. Similar to the case of the single-session analysis here we also do the analysis for F_{SIG}–hybrid protocol, reduce the number of hard tasks underlying the analyzed protocol from two to one. Let S and S’ denote the simulator for the single-session case and the multiple-session case, respectively. In both cases of corruption,

simulator S' performs the following common steps: simulator S' simulates an instance of ideal functionality F_{SIG}. Simulator S' invokes a fresh copy of S for each session.

In the case of a corrupted creator, assume we are at the session with an identifier (sid, ssid). Simulator S extracts (sid, ssid, ρ_1) using the rewinding technique shown in the proof of Claim 1. Simulator S outputs the message "Failure" or sends a message (sid, ssid, ρ) to signature functionality. In the former case simulator S' outputs the message "Failure", in the latter case it generates a signature to the message (sid, ssid, ρ) with the use of simulated ideal functionality F_{SIG}.

In the case of a corrupted user, simulator S computes the commitment honestly but cheats (with zero cheating error) with the Sigma-proof by running the SHZVK-simulator of the Sigma protocol. Simulator S' signs message (sid, ssid, ρ') via simulating ideal functionality F_{SIG}.

Clearly ρ and ρ' are uniform random samples from space $\{0, 1\}^n$.

A summary of the analysis follows. The analysis follows the ideas of the single session case. Events E_a, E_b, and E_c are extended to the multiple execution scenarios and we denote the corresponding events as E'_a, E'_b, and E'_c . Here event E'_a means that an event of type E_a happens in all sub-sessions and events E'_b and E'_c denote that an event E_b and E_c occurs in some of the sub-sessions, respectively.

Assume no abort happened. First, we consider the case of the corrupt creator. Assuming no failure, the joint outputs of the real and ideal systems are

$[(\rho^{(1)}, \mathit{S}ign_{SK}(\rho^{(1)})); \rho^{(1)}],\ldots, [(\rho^{(s)}, \mathit{S}ign_{SK}(\rho^{(s)})); \rho^{(s)}]$

and

$[(\rho'^{(1)}, \mathit{S}ign_{SK'}(\rho'^{(1)})); \rho'^{(1)}],\ldots, [(\rho'^{(s)}, \mathit{S}ign_{SK'}(\rho'^{(s)})); \rho'^{(s)}]$

Where the different ρ samples are chosen independently and uniformly, furthermore keys SK and SK' are samples taken randomly from the same distribution. These joint outputs are indistinguishable. We notice that this conclusion is not affected by the repetition of execution.

Repetition affects the event of failure. However, the negligibility of the probability of failure does not change. Probability $P(E'_b)$ is negligible. Indeed $P(E'_b) \leq P(E_{b,1}) + P(E_{b,2} | \omega_1) + \ldots + P(E_{b,s} | \omega_{s-1})$, where ω_{i-1} denotes the auxiliary information available for the corrupt creator it repetition step i. The computational binding property of the commitment scheme guarantees that $P(E_{b,i} | \omega_{i-1})$ is negligible, for all i. We also note that $P(E'_c) \leq s2^{-t}$, where t is the length of the challenge string.

Now we consider the case of a corrupt user. No simulation failure can happen. The corrupt user is forced to be semi-honest, it cannot distort either the distribution or the independence of the output string. Its view of the interaction is perfectly indistinguishable from the specification.

□

6. Multi-party extension

Fig.7. shows a realistic multiple-session scenario. Creator parties can afford to purchase a large number of high-quality tamperproof blank tokens. For example, a creator party is a financial institution or a governmental authority. Creator parties distribute tokens to the set of users associated with them. The public parameters of the functionality loaded by a given creator into its tokens are identical. The keys of digital signature within the functionality differ from token

to token. All sessions between creator party and its users have the same identifier (sid) and the sessions are identified by sub-session identifier (sid, ssid). Different sid values identify different creators' sessions. Creator-user pairs can generate multiple common random strings and use them in UC-secure protocols based on such a setup.

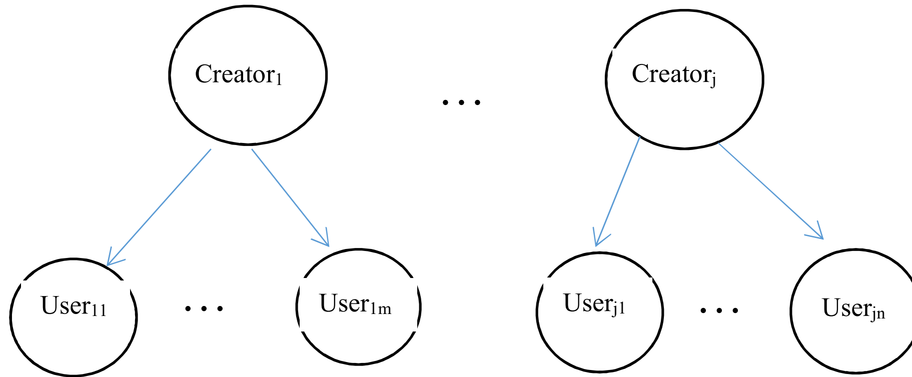


Fig.7. A multiple-session scenario

The protocol we have considered so far provides CrS for a pair of parties. The application possibilities of the protocol increase if it provides CrS for multiple parties. Based on the scenario in Fig. 7. we show how to provide CrS for 3 parties. Assume users U_1 and U_2 come from the same cluster with common creator C. Pairs (C, U_1) and (C, U_2) generate CrS s_1 and s_2, respectively. Party U_1 requests string s_2 from C and U_2. Similarly, party U_2 requests string s_1 from C and U_1. Party U_1 and U_2 check that the received two strings are identical, respectively. If not, the protocol is aborted, otherwise all the three parties, C, U_1 and U_2 compute $s=s_1+s_2$ (mod 2). Note creator C can always compute the correct sum of the two CrS strings. A user party can always detect dishonest action since at least one of its parties is honest. The upshot is that in case of no abort the three parties will have a correct common random string.

7. Complexity

The protocol is very efficient. We exemplify it by the application of Pedersen's commitment. It requires a constant number of operations (per session): 4 exponentiations and 2 multiplications in the underlying group G; one multiplication and one addition mod q; one digital signature computed by the token model. The number of random bits supplied by the token (per session) is ~ 4n, where n bit long CrS is generated.

References

[1] B. Barak, R. Canetti, J. B. Nielsen, and R. Pass. Universally composable protocols with relaxed set-up assumptions. In Proceedings of the 45th Annual Symposium on Foundations of Computer Science, 2004, pp. 186–195.

[2] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. In *proceedings of STOC '88*, pages 103–112, 1988.

[3] Brandon Broadnax, Alexander Koch, Jeremias Mechler, Tobias Müller, Jörn Müller-Quade, Matthias Nagel. Fortified Universal Composability: Taking Advantage of Simple Secure Hardware Modules. Proceedings on Privacy Enhancing Technologies, 2021 (4), pp. 312–338.

[4] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 34th STOC, pages 494-503, 2002.

[5] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. STOC 2002.

[6] Nishanth Chandran, Wutichai Chongchitmate, Rafail Ostrovsky, Ivan Visconti. Universally Composable Secure Computation with Corrupted Tokens. CRYPTO 2019: Advances in Cryptology – CRYPTO 2019, LNSC, vol. 11694, pp. 432–446.

[7] Ivan Damgård, Jesper Buus Nielsen, and Daniel Wichs. Universally composable multiparty computation with partially isolated parties. In Omer Reingold, editor, TCC 2009, LNCS, vol. 5444, pp. 315–331.

[8] Rafael Dowsley, Jörn Müller-Quade Tobias Nilges. Weakening the Isolation Assumption of Tamper-proof Hardware Tokens. In Anja Lehmann and Stefan Wolf, editors, ICITS 15: 8th International Conference on Information Theoretic Security, volume 9063 of Lecture Notes in Computer Science, pages 197, Springer. 2015.

[9] Marc Fischlin, Benny Pinkas, Ahmad-Reza Sadeghi, Thomas Schneider, and Ivan Visconti. Secure set intersection with untrusted hardware tokens. In Aggelos Kiayias, editor, CT-RSA 2011, volume 6558 of LNCS, pages 1–16. Springer, February 2011.

[10] Marc Fischlin, Felix Rohrbach, Single-to-Multi-Theorem Transformations for Non-Interactive Statistical Zero-Knowledge, LNCS, 12711, In: Theory of Public Key Cryptography – PKC 2021, pp. 205-234.

[11] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Onetime programs. In David Wagner, editor, CRYPTO 2008, volume 5157 of LNCS, pp. 39–56.

[12] J. Groth and R. Ostrovsky. Cryptography in the multi-string model. In Advances in Cryptology - CRYPTO ’07, 2007, pp. 323–341.

[13] D. Hofheinz, J. Müller-Quade, and D. Unruh. Universally composable zero-knowledge arguments and commitments from signature cards. In Proceedings of the 5th Central European Conference on Cryptology, 2005.

[14] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Embedded SFE: Offloading server and network using hardware tokens. In Radu Sion, editor, FC 2010, volume 6052 of LNCS, pp. 207–221.

[15] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In Moni Naor, editor, EUROCRYPT 2007, LNCS, vol. 4515, pp. 115–128.

[16] Lindell, Y. How To Simulate It - A Tutorial on the Simulation Proof Technique. In book Tutorials on the Foundations of Cryptography. ISBN: 978-3-319-57048-8, Springer 2017.

[17] Benoıt Libert, Alain Passelegue, Hoeteck Wee, David J. Wu New. Constructions of Statistical NIZKs: Dual-Mode DV-NIZKs and More. EUROCRYPT 2020: 39th Annual

International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part III, pp. 410–441.

[18] Tal Moran and Gil Segev. David and Goliath Commitments: UC Computation for Asymmetric Parties Using Tamper-Proof Hardware, EUROCRYPT 2008, LNSC, volume 4965, pp. 527–544.

[19] Willy Quach, Ron D. Rothblum, and Daniel Wichs. Reusable designated-verifier NIZKs for all NP from CDH. In book: Advances in Cryptology – EUROCRYPT 2019, pp.593-621.