Universally Composable Secure Computation with Corrupted Tokens

Nishanth Chandran
Microsoft Research India, India
nichandr@microsoft.com

Wutichai Chongchitmate*
Chulalongkorn University, Thailand
wutichai.ch@chula.ac.th

Rafail Ostrovsky†
UCLA, USA
rafail@cs.ucla.edu

Ivan Visconti
Università di Salerno, ITALY
visconti@unisa.it

Abstract

We introduce the *corrupted token model*. This model generalizes the *tamper-proof token model* proposed by Katz (EUROCRYPT '07) relaxing the trust assumption on the honest behavior of tokens. Our model is motivated by the real-world practice of outsourcing hardware production to possibly corrupted manufacturers. We capture the malicious behavior of token manufacturers by allowing the adversary to corrupt the tokens of honest players at the time of their creation.

We show that under minimal complexity assumptions, i.e., the existence of one-way functions, it is possible to UC-securely realize (a variant of) the tamper-proof token functionality of Katz in the corrupted token model with n stateless tokens assuming that the adversary corrupts at most $n-1$ of them (for any positive n). We then apply this result to existing multi-party protocols in Katz's model to achieve UC-secure MPC in the corrupted token model assuming only the existence of one-way functions.

Finally, we show how to obtain the above results using tokens of small size that take only short inputs. The technique in this result can also be used to improve the assumption of UC-secure hardware obfuscation recently proposed by Nayak *et al.* (NDSS '17). While their construction requires the existence of collision-resistant hash functions, we can obtain the same result from only one-way functions. Moreover using our main result we can improve the trust assumption on the tokens as well.

*Work done while the author was at Department of Computer Science, University of California, Los Angeles, USA.

†Research supported in part by NSF grant 1619348, DARPA SafeWare subcontract to Galois Inc., DARPA SPAWAR contract N66001-15-1C-4065, US-Israel BSF grant 2012366, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. The views expressed are those of the authors and do not reflect position of the Department of Defense or the U.S. Government.

1 Introduction

UC-secure MPC. Secure multi-party computation [25] (MPC) allows mutually distrustful parties to jointly compute a function f, while preserving the privacy of their inputs/outputs. Canetti [7] introduced the notion of universal composability (UC) to model secure MPC in an environment where multiple concurrent executions of different protocols take place. Unfortunately UC security is significantly harder to achieve than plain secure computation. In fact, in the plain model (i.e., without trusted set-up assumptions, physical assumptions, superpolynomial-time simulation and so on) most functionalities can not be UC-realized [9, 11]. Impossibility results exist even for the basic case of self-concurrent composition with static inputs [4, 1, 23].

In light of these impossibility results, various trust assumptions have been studied in order to obtain UC-secure constructions. Among these, one of the most studied is that of tamper-proof hardware tokens. Hofheinz *et al.* [30], considered the notion of "tamper-proof devices" in the form of signature cards in order to realize UC-secure protocols. They show how to construct UC-secure commitment and zero-knowledge arguments using tamper-proof signature cards as the setup assumption. The more general formalization of tamper-proof hardware tokens was given by Katz [33]. Katz's tamper-proof token functionality abstractly captures a physical tamper-proof hardware token that is created and sent by a sender to a receiver. The receiver can use the token to execute the program stored in it multiple times as a black-box, on inputs of his choice. Tokens can be either *stateful* (i.e., they retain an updatable memory between executions; this is a stronger trust assumption because it additionally assumes a tamper-proof updatable memory) or *stateless* (i.e., all executions start with the same configuration). Motivated by the practical relevance of the model, as well as the challenging open questions on the feasibility of protocols in this model, UC-security with tamper-proof tokens has been widely explored with recent focus on the more challenging case of stateless tokens [12, 34, 27, 15, 13, 18, 29, 37].

Token manufacturing. Assuming that tokens are honestly generated is clearly a very demanding assumption that essentially requires honest players to rely on the honesty of a token manufacturer that they trust. Hence, while the tamper-proof token model as a physical assumption in theory, in practice it degenerates into a model where the security of an honest player depends on the honest behavior of an external player chosen by the honest player[1]. All prior works that consider hardware-based security critically rely on the honest player being able to reliably construct tamper-proof tokens. An attempt to relax this assumption was done in [21] focusing only on the set intersection functionality, without considering UC security. More recently, [2] considered the problem of outsourcing circuit fabrication where a given circuit is compiled into smaller components, each of which can be outsourced to a possibly malicious manufacturer. The components (both honestly and maliciously manufactured) are then re-assembled honestly to get a "secure hardware token". Their (stand-alone) security model only allows an adversary black-box access to the rebuilt circuit, and not the individual components and additionally also requires one "small" tamper-proof token that can be generated honestly in a trusted manner. In contrast, we do not wish to make any assumptions on small trusted components and consider composability. The above state of affairs brings us to the following natural question.

[1] A similar question for the case of Common Reference String (CRS) generation was answered in the multi-string model [28], where multiple corruptible CRS's are generated.

"Can we obtain UC-secure hardware-based security in a world where most hardware token manufacturers may be corrupt?"

1.1 Our Results

We resolve the above open problem in the positive under minimal complexity assumptions. We now discuss all our contributions in detail.

The corrupted token model. We consider the concrete scenario where the sender of a token does not have the ability to physically create a tamper-proof token, but instead has to rely on possibly untrusted manufacturers. In case a manufacturer is corrupted (and may be colluding with other parties), the program embedded in the token may be leaked, or replaced in its entirety. In other words, tokens in this model can be tampered arbitrarily at the time of creation.

To model security, we define a functionality for UC security allowing the design of protocols that make use of tokens generated by potentially adversarial manufacturers. In turn, we propose a new model extending the stateless version of Katz's tamper-proof token model in [33], that we call *corrupted token model*. In our new model, the adversary is allowed to corrupt tokens when they are created by honest parties. The attack happens during the token creation phase, and the adversary learns all information that the honest player wanted to store in the token. Moreover the adversary is allowed to replace the token with a different token of its choice, including even a stateful one.

The corrupted token model abstractly represents the process of outsourcing the production of hardware tokens to possibly corrupted manufacturers. This is the reason why we allow corruption to occur only at the time the tokens are created. Our model also allows adaptive corruption of the manufacturers, in the sense that the adversary may choose to corrupt the next request of token generation of an honest player depending on what has been learnt so far. Finally, the adversary can freely decide the content of corrupted tokens and can even make it stateful. This is similar to dealing with a real-world hardware Trojan as described in [19, 36] with a few key differences. In the model of [19], there exists an incorruptible "master circuit," whose role is to manage communication between tokens honestly. The model of [36] also has a "controller" circuit, whose role is similar to the master circuit in [19], but is allowed to be compromised. On the other hand, our model does not have a token with a specific role and allows any token to be corrupted. Both [19] and [36] do not consider UC security and additionally the construction of [36] is based on variants of ElGamal public-key encryption and Schnorr signature scheme, whose security are based on hardness of elliptic curve Diffie-Hellman and discrete logarithm, respectively. Our construction, on the other hand, is UC-secure and is based on the minimal assumption of OWFs.

Katz's token functionality in the corrupted token model. We construct a protocol in the corrupted token model using n tokens that UC-realizes a stronger variant of Katz's tamper-proof token functionality. We call such a variant the *tamper-proof token with abort* functionality. The difference between the tamper-proof token with abort functionality and the original Katz's tamper-proof token functionality is that our variant allows the adversary to learn that a token has been sent (even between honest parties), and can choose to abort and prevent the delivery of that token. This captures the realistic scenario where an adversary physically prevents token delivery and thus stops the protocol that relies on tokens. Still the adversary learns nothing about the program in the uncorrupted token generated by the honest party. The need for abort in the functionality is unavoidable as seen through the following reasoning. Suppose the tamper-proof

token functionality (without abort) can be realized by n corruptible tokens. Then, the adversary in the corrupted token model corrupts all but one of the tokens and replaces them with corrupted tokens that do nothing. Now, if the tamper-proof token functionality without abort is realized with the remaining (uncorrupted) token, then this token must hold the complete program and secrets of the honest party (so that it can carry out the computation by itself). However, in an alternate corruption strategy, this token is also susceptible to corruption, and if the adversary had instead corrupted only this token, she would have learnt all the secrets of the original honest token in Katz's model resulting in the insecurity of the protocol. Hence, we must model the functionality to allow for aborts.

It is easy to see that this argument extends to the case of any dishonest majority (i.e., even with only $\lfloor n/2+1 \rfloor$ corrupted tokens). Our protocol UC-realizes the tamper-proof token with abort functionality assuming that the adversary corrupts at most $n-1$ tokens where n is the number of token creations invoked by an honest party. We remark that, if we were willing to make the assumption that a majority of token manufacturers were honest, then we can avoid aborting the protocol even when the adversary corrupts a (minority) fraction of the tokens.

The notion of token transfer across the environment in various sessions in the Global-UC (GUC) framework has been studied recently by Hazay *et al.* [29]. Obtaining GUC security is way more challenging and we leave the question of GUC security in the corrupted token model to future research. Still, we stress that in many natural scenarios, UC security already suffices and achieves a very strong level of security under composition with any other protocol as long as there is a way to avoid the *sharing of the same setup among sessions.*

A compiler to reduce trust in tokens. As our main result, we show how to transform any protocol in Katz's tamper-proof token model into a protocol in our corrupted token model thereby improving the trust assumption of several hardware token-based protocols. Indeed the transformed protocol remains secure even when $n-1$ out of the n tokens created by honest parties are corrupted at the time of creation. Our transformation preserves UC security and only assumes the existence of one-way functions (OWF). We focus on stateless tokens since this is the milder physical assumption and is the most challenging case. We remark here that requiring one token to be uncorrupted is unavoidable. To see this, suppose for the sake of contradiction, there is a protocol UC-realizing a tamper-proof token functionality using n corruptible tokens that remains secure even when an adversary corrupts all n tokens. Now suppose an adversary in the secure computation protocol corrupts all but one of the parties and corrupts all the tokens manufactured by this party. Now, if the resulting protocol still remains secure, then this would give us a UC-secure MPC protocol (secure against all-but-one corruption) with no trusted setup, as all "trusted" components created by the honest party are corrupted (in more detail, generating and sending a token could then be replaced by sending a message with the description of the program of the token). This contradicts known impossibility results [9, 11]. Hence, we must assume that at least one hardware token created by every honest party is uncorrupted. Additionally, the existence of OWFs is the minimal assumption that one can hope for since, as argued in [27], any unbounded adversary can query a stateless tokens exponentially many times to learn the programs embedded.

Our transformation can be applied to existing protocols in the Katz's token model to obtain new results in the corrupted token model. For instance, starting with the recent UC-secure MPC constructions in the tamper-proof token model based on OWFs [29], we get the same results in the corrupted token model assuming only OWFs.

Other results and sub-protocols. As an additional result, we improve the result of [37] by removing the need of collision-resistant hash functions, and apply our transformation to obtain an obfuscation protocol in the corrupted token model based solely on OWFs. Moreover, as a building block for our constructions, we present a simultaneous resettable zero-knowledge (sim-res ZK) argument and UC-secure MPC for any well-formed functionality in the correlated randomness model assuming OWFs only. In the correlated randomness model, each party has access to a private, input-independent, honestly generated, string before the execution of the protocol by the correlated randomness functionality. These protocols may be of independent interest. We stress that correlated randomness is not required as a setup for our construction achieving UC security in the corrupted token model and is only used as an intermediate building block.

1.2 High-Level Overview of Our Constructions

Realizing Katz's token functionality. We begin by describing how to UC-realize Katz's token functionality in the $(n, n-1)$-token-corruptible hybrid model, (i.e., the model where n tokens are generated by an honest player and at most $n-1$ are corrupted by the adversary at the time of token generation). We refer to the final protocol that realizes Katz's functionality as Π. Protocol Π will make use of a UC-secure n-party protocol Π' and a sim-res ZK argument Π_{rsZK} with straight-line simulator, both in the correlated randomness model[2]. At a very high level, we construct Π as follows. Given a description of the program P for Katz's tamper-proof token (such a description is specified by the protocol in Katz's model) we first create n shares of the description of P using an n-out-of-n threshold secret sharing scheme. Then n tokens are created as follows. The program of the i-th token includes: 1) the i-th share; 2) commitments of all shares; 3) decommitment information for the i-th share; 4) correlated randomness to run the n-party UC-secure MPC in the correlated randomness model; 5) correlated randomness to run a simultaneous resettable ZK in the correlated randomness model; 6) a seed for a PRF; and 7) a random tape for commitment of the seed.

When a user must query a Katz token implementing program P on input x, he/she must first send each token this input value (a dishonest user may send different values to different tokens). We shall refer to the "version" that the i^{th} token receives by x_i. Upon queried with an input x_i, the i-th token first commits to its input (i.e., x_i) and to its seed for the PRF (see point 6 above). The randomness used for the first commitment comes from evaluating the PRF using the above seed and the input x_i while the randomness for the second commitment is the string stored in the token (see point 7 above). These commitments provided by all the tokens together is called the determining message. For the remaining execution of Π, each token obtains the random tape needed by Π' and Π_{rsZK} by computing the PRF on the determining message (and a unique ID value) using its seed. The tokens will execute an n-party UC-secure MPC protocol in the correlated randomness model, Π' (see point 4 above). More specifically, the i-th token, if honest, will run the code of the i-th player of Π' on input the following pair: the received input (i.e., x_i) and the i-th share of P (see step 1 above). Π' will securely compute the functionality that reconstructs P from the shares that are part of the inputs of the players and then executes P on input x. The reconstruction aborts if $x \neq x_i$ for some i. Each Π' message m sent by the i^{th} token is followed by a simultaneous resettable ZK argument of knowledge (see step 5 above) proving that the message m is computed correctly according to the committed PRF seed and the i-th committed share (see step 2 and 3 above) of P.

[2]The correlated randomness is the key that allows us to avoid the impossibility of resettably-secure computation in the standard model proven in [23]. However, we stress that correlated randomness is not required by our main theorem for UC security with tamper-proof stateless corruptible tokens from OWFs.

The resettable soundness of the ZK argument guarantees that a corrupted token cannot deviate from the underlying MPC protocol Π' even when the adversary can execute any tokens any number of times on any inputs of his choice (even after resetting the state of the honest token several times). Moreover, the security of Π' guarantees that the adversary corrupting all but one token does not learn anything about the inputs of uncorrupted tokens other than x_i he chooses and the output. This means the adversary only learns at most $n-1$ shares of the program P, and thus learns nothing about P by the security of the secret sharing scheme.

Since the tokens are stateless, we employ the technique in [37] to encrypt the state of the token and output it along with the message. Each subsequent invocation of the token requires an encrypted previous state as an additional input. A symmetric key encryption scheme is used to prevent the adversary to learn or modify states of uncorrupted tokens. This allows us to construct a simulator for the corruptible token protocol that simulates both the MPC and the ZK messages using their simulators.

Simultaneous resettable ZK argument in the correlated randomness model. The above discussion assumed the existence of a sim-res ZK argument Π_{srZK} with straight-line simulator in the correlated randomness model. We obtain this result in 2 steps starting from a 3-round public-coin ZK argument Σ, in the CRS model with straight-line simulation based on OWFs (such as the one in [35]).

First, we add the argument of knowledge (AoK) property with straight-line witness extractor to Σ in the correlated randomness model. For this, we use a technique similar to the one used in [22] where a prover encrypts a witness, sends the encryption as a message, and then uses Σ to prove that it is the encryption of the right witness. To avoid the use of stronger assumptions than OWFs, we replace a public-key encryption scheme in [22] with a secret-key encryption scheme and a commitment scheme. The commitment of a secret key and its corresponding decommitment information are given to the prover while only the commitment is given to the verifier as part of their correlated randomness. The resulting protocol is still 3-round, public-coin and with straight-line simulation.

In the second step, we add a simultaneous resettable witness indistinguishability (sim-res WI) argument of knowledge from OWFs to construct a simultaneous resettable zero-knowledge argument in the correlated randomness model with straight-line simulation. To prevent a malicious prover from resetting, the verifier uses a PRF applied to the statement and the prover's message to generate a string c to play in the second round of Σ instead of uniformly sampling her message. Then, to prevent a malicious verifier from resetting, the verifier runs the prover of the sim-res WI to prove that c is generated honestly or that a given long string d is an output of a PRG on input a short seed. Since d is uniformly chosen as part of the correlated randomness, the verifier cannot maliciously manipulate c. Resettable soundness can then be shown through a hybrid experiment where d is generated from the PRG. This technique is akin to the one used to construct robust NIZK in [17].

UC-secure n-party computation in the correlated randomness model. Our main result also assumed Π', i.e., a UC-secure n-party computation protocol in the correlated randomness model for any well-formed functionality. We next outline how we construct Π' based only on OWFs.

First, we consider UC-secure MPC in the OT-hybrid model against a malicious adversary corrupting all but one party such as the one in [32]. Since OT can be generated using correlated randomness [5], we turn our focus towards obtaining a UC-secure MPC in the correlated randomness

model. However, in our setting, we face a new challenge. Since our final protocol can be executed on polynomially many inputs, where the polynomial is not apriori known to the correlated randomness generator, we must be able to produce "different" randomness for any input on which the protocol is generated. This would require a stronger version of the OT extension technique from [6] that allows the extension to super-polynomial number of OTs. This is similar in spirit to constructing a PRF that can generate "super-polynomial" randomness from a short seed (even though it will only be evaluated on polynomially many inputs). In particular, we modify the technique in [6] to construct UC-secure unbounded number of OTs from a small number of OTs distributed as setup in the correlated randomness model. We do this as follows. In the OT extension protocol in [6], a sender uses a circuit that first computes a PRG that takes a small input and outputs a large random string and then uses the string to obtain a large number of OTs. The circuit is then garbled using Yao's garbled circuit and sent to a receiver. The receiver then uses a small number of OTs to obtain a garbled input correspond to its small random seed. In our approach, the sender uses a PRF that allows us to generate super-polynomial number of such random strings. While a computationally bounded sender cannot compute or send a garbled circuit of super-polynomial size, it only needs to send a smaller subcircuit to compute the ith string in each execution. This garbled circuit is of polynomial size as in Beaver's version, computing only the required amount of OTs at a time. This is repeated to give an (apriori) unbounded number of OTs. Composing the UC-secure unbounded number of OTs in the correlated randomness model and a UC-secure MPC in the OT-hybrid model, we get a UC-secure MPC in the correlated randomness model.

Getting rid of correlated randomness. While the building blocks Π' and Π_{rsZK} are in the correlated randomness model, our main protocol Π is not. Both subprotocols will be run by n tokens created by a single honest party to emulate the token functionality of a single well-formed token in Katz's model. Therefore, in Π, the party requesting the creation of a token can generate and give the correlated randomness to n different manufacturers. Hence, the correlated randomness is not a setup of our main result, and is computed by an honest player in our protocol to run subprotocols that need it. An adversary can replace the correlated randomness in $n-1$ of those tokens arbitrarily, and still our protocol is secure because Π' and Π_{rsZK} are secure with respect to such behavior.

Reducing the token size. In order to ensure that the queries to tokens are short and the size of each token is small, we consider a technique used in [37] where a large input is fed into a token in blocks of small size. To ensure the consistency of the input, in [37] a Merkle's tree based on CRHFs is used to commit to the input beforehand. We improve on this technique by replacing the Merkle's tree with a new construction based on OWFs. At a very high level, we require the user to "commit" to his/her input by feeding the input bit-by-bit into to the token. The token will produce an authentication tag for every bit of the input sequentially, such that the final authentication tag will act as a "commitment" to the user's input. This result is of independent interest as an improvement on the assumption of [37]. We generalize the technique of "bounded-size" tokens to our corrupted token protocol. We first give a variant of corrupted token functionality where the token size is independent of program P and the token can only accept queries of (apriori) fixed, constant size. We then construct a protocol that UC-realizes the corrupted token functionality in the corrupted "bounded-size" token hybrid model using the above technique. Finally, we combine this protocol with our main result to give a protocol that UC-realizes 'standard' tamper-proof token

functionality in the corrupted "bounded-size" token hybrid model.

1.3 Organization of the paper

We present the building blocks used in our construction (such as interactive argument systems, resettable zero-knowledge and so on) as well as describe the correlated randomness model and UC security in Section 2. In Section 3, we present our first result: a simultaneous resettable straight-line zero-knowledge argument in the correlated randomness model, based solely on OWFs. In Section 4, we show how to construct a UC-secure MPC protocol in the correlated randomness model based on OWFs. We define our corrupted tamper-proof token model in Section 5 and we present our main compiler that converts any protocol in the Katz's tamper-proof token model into our $(n, n-1)-$corrupted tamper-proof token model in Section 6 of the paper. Finally, we show an application of our compiler to secure obfuscation in Section 7.

2 Preliminaries

A polynomial-time relation R is a relation for which it is possible to verify in time polynomial in $|x|$ whether $R(x, w) = 1$. Let us consider an $\mathcal{NP}$-language L and denote by R_L the corresponding polynomial-time relation such that $x \in L$ if and only if there exists w such that $R_L(x, w) = 1$. We will call such a w a *valid witness for* $x \in L$. Let λ denote the security parameter. A *negligible* function $\nu(\lambda)$ is a non-negative function such that for any constant $c < 0$ and for all sufficiently large λ, $\nu(\lambda) < \lambda^c$. We will denote by $\Pr_r[X]$ the probability of an event X over coins r, and $\Pr[X]$ when r is not specified. The abbreviation "PPT" stands for probabilistic polynomial time. For a randomized algorithm A, let $A(x; r)$ denote running A on an input x with random coins r. If r is chosen uniformly at random with an output y, we denote $y \leftarrow A(x)$. For a pair of interactive Turing machines (P, V), let $\langle P, V\rangle(x)$ denote V's output after interacting with P upon common input x. We say V accepts if $\langle P, V\rangle(x) = 1$ and rejects if $\langle P, V\rangle(x) = 0$. We denote by $\mathtt{view}_{V(x,z)}^{P(w)}$ the view (i.e., its private coins and the received messages) of V during an interaction with $P(w)$ on common input x and auxiliary input z. We will use the standard notion of computational indistinguishability [26].

2.1 Building Blocks

As mentioned above, the main building blocks of our construction include simultaneous resettable ZK arguments and MPC in the correlated randomness model. We present various definitions related to interactive argument systems, zero-knowledge arguments of knowledge, witness indistinguishability and resettability in the correlated randomness model, adapted from their counterparts [10, 20, 3] in the plain model. We also present other standard definitions of commitments, secret sharing schemes and pseudorandom functions that we make use of in our construction.

Definition 2.1 (interactive argument system in the correlated randomness model)**.** *An* interactive argument system *for the language L consists of a correlated random string generation algorithm K and a pair of interactive Turing machines (P, V) where V runs on input (σ_V, x) and P runs on input (σ_P, x, w) where w is a witness for x such that:*

- *Efficiency: K, P and V are* PPT.

- *(Perfect) Completeness: For every $\lambda \in \mathbb{N}$ and for every pair (x, w) such that $(x, w) \in R_L$,*

$$\Pr[(\sigma_P, \sigma_V) \leftarrow K(1^\lambda) : \langle P(\sigma_P, w), V(\sigma_V)\rangle(x) = 1] = 1.$$

- *Soundness: There exists a negligible function $\nu(\cdot)$ such that for any non-uniform* PPT *adversary $P^* = (P_1^*, P_2^*)$*

$$\Pr[(x, z) \leftarrow P_1^*(\sigma_P) : x \notin L \wedge \langle P_2^*(\sigma_P, z), V(\sigma_V)\rangle(x) = 1] < \nu(\lambda).$$

If K always outputs $\perp$, we say that (P, V) is an interactive argument (in the plain model). If K outputs $\sigma_P = \sigma_V$, we say that (P, V) is an interactive argument in the CRS model.

Definition 2.2 (zero-knowledge arguments)**.** *Let (K, P, V) be an interactive argument system for a language L. We say that (K, P, V) is zero knowledge (ZK) if, for any probabilistic polynomial-time adversary V^*, there exists probabilistic polynomial-time algorithms $S_{V^*} = (S_1, S_2)$ such that, for all auxiliary inputs z and all pairs $(x, w) \in R_L$*

$$|\Pr[(\sigma_P, \sigma_V) \leftarrow K(1^\lambda) : \langle P(\sigma_P, w), V^*(\sigma_V, z)\rangle(x) = 1]$$

$$-\Pr[(\sigma_P, \sigma_V, \tau) \leftarrow S_1(1^\lambda) : \langle S_2(\tau), V^*(\sigma_V, z)\rangle(x) = 1]| < \nu(\lambda)$$

Definition 2.3 (witness indistinguishability)**.** *Let L be a language in $\mathcal{NP}$ and R_L be the corresponding relation. An interactive argument (K, P, V) for L is* witness indistinguishable *(WI) if for every verifier V^*, every pair (w_0, w_1) such that $(x, w_0) \in R_L$ and $(x, w_1) \in R_L$ and every auxiliary input z, for $\sigma \leftarrow K(1^\lambda)$, the following ensembles are computationally indistinguishable:*

$$\{\mathtt{view}_{V^*(\sigma,x,z)}^{P(\sigma,w_0)}\} \quad \textit{and} \quad \{\mathtt{view}_{V^*(\sigma,x,z)}^{P(\sigma,w_1)}\}.$$

Definition 2.4 (argument of knowledge)**.** *Let (K, P, V) be an interactive argument system for a language L. We say that (K, P, V) is* argument of knowledge *if there exists probabilistic polynomial-time algorithms $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2)$ such that*

- *for all non-uniform polynomial-time adversary $\mathcal{A}$,*

$$\Pr[(\sigma_P, \sigma_V) \leftarrow K(1^\lambda) : \mathcal{A}(\sigma_P) = 1] = \Pr[(\sigma_P, \sigma_V, \tau) \leftarrow \mathcal{E}_1(1^\lambda) : \mathcal{A}(\sigma_P) = 1]$$

- *for all non-uniform polynomial-time adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, the following experiments are indistinguishable:*

$\mathsf{Exp}_{\mathcal{A}}(\lambda)$:	$\mathsf{Exp}_{\mathcal{A}}^{\mathcal{E}}(\lambda)$:
1. $(\sigma_P, \sigma_V) \leftarrow K(1^\lambda)$.	1. $(\sigma_P, \sigma_V, \tau) \leftarrow \mathcal{E}_1(1^\lambda)$.
2. $(x, z) \leftarrow \mathcal{A}_1(\sigma_P)$.	2. $(x, z) \leftarrow \mathcal{A}_1(\sigma_P)$.
3. $b \leftarrow \langle \mathcal{A}_2(z), V(\sigma_V)\rangle(x)$.	3. $(b, w) \leftarrow \langle \mathcal{A}_2(z), \mathcal{E}_2(\tau)\rangle(x)$.
4. *Output b.*	4. *Output 1 iff $b = 1$ and $(x, w) \in R_L$.*

Definition 2.5 (resetting adversary)**.** *Let (K, P, V) be an interactive argument system for a language L, $t = \mathsf{poly}(\lambda)$, $\bar{x} = x_1, \ldots, x_t$ be a sequence of common inputs and $\bar{w} = w_1, \ldots, w_t$ the corresponding witnesses (i.e., $(x_i, w_i) \in R_L$) for $i = 1, \ldots, t$. Let $r_1, \ldots, r_t$ be independent random tapes. We say that a PPT V^* is a* resetting *verifier if for $(\sigma_P, \sigma_V) \leftarrow K(1^\lambda)$, it concurrently interacts with an unbounded number of independent copies of $P(\sigma_P)$ by choosing for each interaction the value i so that the common input will be $x_i \in \bar{x}$, and the prover will use witness w_i, and choosing j so that the prover will use r_j as randomness, with $i, j \in \{1, \ldots, t\}$. The scheduling or the messages to be sent in the different interactions with P are freely decided by V^*. Moreover we say that the transcript of such interactions consists of the common inputs $\bar{x}$ and the sequence of prover and verifier messages exchanged during the interactions. We refer to $\mathtt{view}^{P(\sigma_P, \bar{w})}_{V^*(\sigma_V, \bar{x}, z)}$ as the random variable describing the content of the random tape of V^* and the transcript of the interactions between P and V^* using $(\sigma_P.\sigma_V)$ as correlated random strings, where z is an auxiliary input received by V^*.*

Definition 2.6 (resettable zero knowledge)**.** *Let (K, P, V) be an interactive argument system for a language L. We say that (K, P, V) is resettable zero knowledge (rZK) if, for any PPT resetting verifier V^* there exists a expected probabilistic polynomial-time algorithm $S_{V^*} = (S_1, S_2)$ such that the for all pairs $(\bar{x}, \bar{w}) \in R_L$, for $(\sigma_P, \sigma_V, \tau) \leftarrow S_1(1^\lambda)$, the ensembles $\{\mathtt{view}^{P(\sigma_P, \bar{w})}_{V^*(\sigma_V, \bar{x}, z)}\}$ and $\{S_{V^*}(\tau, \bar{x}, z)\}$ are computationally indistinguishable.*

Definition 2.7 (resettable WI)**.** *Let L be a language in $\mathcal{NP}$ and R_L be the corresponding relation. An interactive argument system (K, P, V) for L is* resettable *witness indistinguishable (rWI) if for every PPT resetting verifier V^* every $t = \mathsf{poly}(\lambda)$, and every pair $(\bar{w}^0 = (w_1^0, \ldots, w_t^0), \bar{w}^1 = (w_1^1, \ldots, w_t^1))$ such that $(x_i, w_i^0) \in R_L$ and $(x_i, w_i^1) \in R_L$ for $i = 1, \ldots, t$, and any auxiliary input z, for $\sigma \leftarrow K(1^\lambda)$, the following ensembles are computationally indistinguishable:*

$$\{\mathtt{view}^{P(\sigma, \bar{w}^0)}_{V^*(\sigma, \bar{x}, z)}\} \quad \textit{and} \quad \{\mathtt{view}^{P(\sigma, \bar{w}^1)}_{V^*(\sigma, \bar{x}, z)}\}.$$

Definition 2.8 (resettably-sound arguments)**.** *A resetting attack of a cheating prover P^* on a resettable verifier V is defined by the following two-step random process, indexed by a security parameter λ.*

1. *Uniformly select and fix $t = \mathsf{poly}(\lambda)$ random-tapes, denoted $r_1, \ldots, r_t$, for V, resulting in deterministic strategies $V^{(j)}(x) = V_{\sigma_V, x, r_j}$ defined by $V_{\sigma_V, x, r_j}(\alpha) = V(\sigma_V, x, r_j, \alpha)$,[3] where $(\sigma_P, \sigma_V) \leftarrow K(1^\lambda)$, $x \in \{0,1\}^\lambda$ and $j \in [t]$. Each $V^{(j)}(\sigma_V, x)$ is called an incarnation of V.*
2. *On input 1^λ, machine P^* is allowed to initiate $\mathsf{poly}(\lambda)$-many interactions with the $V^{(j)}(x)$'s. The activity of P^* proceeds in rounds. In each round P^* chooses $x \in \{0,1\}^\lambda$ and $j \in [t]$, thus defining $V^{(j)}(x)$, and conducts a complete session with it.*

Let (K, P, V) be an interactive argument for a language L. We say that (K, P, V) is a resettably-sound argument *for L if the following condition holds:*

- Resettable-soundness*: For every polynomial-size resetting attack, the probability that in some session the corresponding $V^{(j)}(x)$ has accepted and $x \notin L$ is negligible.*

[3]Here, $V(\sigma_V, x, r, \alpha)$ denotes the message sent by the strategy V on the correlated random string σ_V, common input x, random-tape r, after seeing the message-sequence α.

An interactive argument system that is both resettable zero-knowledge and resettably-sound is called *simultaneous resettable zero-knowledge* argument.

Definition 2.9 (commitment scheme)**.** *Given a security parameter 1^λ, a commitment scheme* com *is a two-phase protocol between two* PPT *interactive algorithms, a sender S and a receiver R. In the commitment phase S on input a message m interacts with R to produce a commitment $c = \mathsf{com}(m)$. In the decommitment phase, S sends to R a decommitment information d such that R accepts m as the decommitment of c.*

Formally, we say that com *is a perfectly binding commitment scheme if the following properties hold:*

Correctness:

- *Commitment phase. Let $c = \mathsf{com}(m)$ be the commitment of the message m given as output of an execution of* com *where S runs on input a message m. Let d be the private output of S in this phase.*
- *Decommitment phase*[4]*. R on input m and d accepts m as decommitment of c.*

Statistical (resp. Computational) Hiding: *for any adversary (resp.* PPT *adversary) $\mathcal{A}$ and a randomly chosen bit $b \in \{0,1\}$, consider the following hiding experiment $\mathsf{ExpHiding}^b_{\mathcal{A},\mathsf{com}}(\lambda)$:*

- *Upon input 1^λ, the adversary $\mathcal{A}$ outputs a pair of messages m_0, m_1 that are of the same length.*
- *S on input the message m_b interacts with $\mathcal{A}$ to produce a commitment of m_b.*
- *$\mathcal{A}$ outputs a bit b' and this is the output of the experiment.*

For any adversary (resp. PPT *adversary) $\mathcal{A}$, there exist a negligible function ν, s.t.:*

$$\left| \Pr[\mathsf{ExpHiding}^0_{\mathcal{A},\mathsf{com}}(\lambda) = 1] - \Pr[\mathsf{ExpHiding}^1_{\mathcal{A},\mathsf{com}}(\lambda) = 1] \right| < \nu(\lambda).$$

Statistical (resp. Computational) Binding: *for every commitment* $\mathtt{com}$ *generated during the commitment phase by a possibly malicious unbounded (resp. malicious* PPT*) sender S^* there exists a negligible function ν such that S^*, with probability at most $\nu(\lambda)$, outputs two decommitments (m_0, d_0) and (m_1, d_1), with $m_0 \neq m_1$, such that R accepts both decommitments.*

We also say that a commitment scheme is perfectly binding *iff $\nu(\lambda) = 0$.*

Definition 2.10 (secret sharing scheme)**.** *Let K be a finite set of secrets. An n-*out-of-n secret sharing scheme *$\mathcal{S}$ consists of a randomized algorithm $\mathsf{share} : K \to K_1 \times \ldots \times K_n$ and a deterministic algorithm $\mathsf{recon} : K_1 \times \ldots \times K_n \to K$ satisfying*

- *Correctness: for any $s \in K$,*

$$\Pr[\mathsf{recon}(\mathsf{share}(s)) = s] = 1;$$

[4]In this paper we consider a non-interactive decommitment phase only.

- *Privacy: for any* $s, s' \in K$, $(s_1, \ldots, s_n) \in K_1 \times \ldots \times K_n$ *and any* $T \subsetneq [n]$,

$$\Pr[\mathsf{share}(s)_T = (s_i)_{i \in T}] = \Pr[\mathsf{share}(s')_T = (s_i)_{i \in T}]$$

where $(s_1, \ldots, s_n)_T = (s_i)_{i \in T}$.

Definition 2.11 (pseudorandom function (PRF))**.** *A family of functions* $\{f_s\}_{s \in \{0,1\}^*}$ *is called* pseudorandom *if for all adversarial PPT machines* $\mathcal{A}$*, for every positive polynomial* $p(\cdot)$*, and sufficiently large* $\lambda \in \mathbb{N}$*, it holds that*

$$|\Pr[\mathcal{A}^{f_s}(1^\lambda) = 1] - \Pr[\mathcal{A}^F(1^\lambda) = 1]| \leq \frac{1}{p(\lambda)}.$$

where $|s| = n$ *and* F *denotes a truly random function.*

2.2 UC Security in the Correlated Randomness Model

The correlated randomness model is an extension of the CRS model where each party has access to a string generated by a trusted third party. Unlike in the CRS model, the strings for parties may be different, but possibly correlated. Also, unlike in the augmented CRS model of [8], honest parties can access their strings privately. Thus, it can be considered as a variant of the key registration (KR) model of [8].

Our correlated randomness model is defined to be consistent with the one of [31], taking into account the UC setting. A protocol ϕ in the correlated randomness model is defined with the corresponding correlated randomness functionality $\mathcal{F}_{corr}^{\phi}$, which generates a correlated random string for each party in the protocol ϕ independently of the parties' input. Each party can access its string (but not other parties' random strings) by invoking $\mathcal{F}_{corr}^{\phi}$.

In the security proof, the ideal world simulator is allowed to obtain the correlated random strings associated to all parties, thereby having an advantage over the real-world adversary.

Let ϕ be n-party protocol in the correlated randomness model. Let $\mathcal{D}$ be a distribution on $S_1 \times \ldots \times S_n$ where S_i is the set of possible random strings for party P_i. The correlated randomness functionality $\mathcal{F}_{corr}^{\phi}$ is defined in Figure 1.

Definition 2.12. *Let* $\mathcal{F}$ *be an ideal functionality and let* ϕ *be a multi-party protocol. Then the protocol* ϕ *UC realizes* $\mathcal{F}$ *in* $\mathcal{F}_{corr}^{\phi}$*-hybrid model if for every PPT hybrid model adversary* $\mathcal{A}$*, there exists a uniform PPT simulator* $\mathcal{S}$ *such that for every non-uniform environment* $\mathcal{Z}$*, the following two ensembles are computationally indistinguishable*

$$\{\mathsf{View}_{\phi,\mathcal{A},\mathcal{Z}}^{\mathcal{F}_{corr}^{\phi}}(\lambda)\}_{\lambda \in \mathbb{N}} \approx^c \{\mathsf{View}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda)\}_{\lambda \in \mathbb{N}}.$$

3 Simultaneous Resettable ZK from OWFs

In this section, we construct a simultaneous resettable ZK argument in the correlated randomness model with straight-line simulation. The security of our construction relies only on the existence of OWFs. The main building block for our construction is a 3-round public-coin ZK argument system in the CRS model with straight-line simulation based on OWFs (such as in [35]). Using this public-coin argument system, we first construct a zero-knowledge *argument of knowledge* (ZKAoK) in the correlated randomness model with straight-line simulation and extraction. We then use this

$\mathcal{F}_{corr}^{\phi}$

When receiving (sid) from P_i:

1. If there is no tuple of the form $(\mathsf{sid}, \star, \ldots, \star)$,
   (a) Generate $(s_1, \ldots, s_n) \leftarrow \mathcal{D}(1^\lambda)$.
   (b) Store $(\mathsf{sid}, s_1, \ldots, s_n)$.

   Otherwise, retrieve the stored $(\mathsf{sid}, s_1, \ldots, s_n)$.
2. Send (sid, s_i) to P_i.

Figure 1: Correlated Randomness Functionality $\mathcal{F}_{corr}^{\phi}$

ZKAoK protocol to construct a simultaneously resettable zero-knowledge protocol in the correlated randomness model, based only on OWFs. We do so, in the following way: As part of the correlated randomness, the prover is given the commitment t, to the seed of a PRF, s, as well as a long random string d. The verifier is given the decommitment information (s and randomness) to this commitment t as well as d. Now, we have the verifier prove, using a simultaneous resettable WI (srWI) argument (based on OWFs [14]), that: either the verifier's random message c in the ZKAoK protocol is the output of a PRF (using seed s) on input the transcript so far, or that d is the output of a PRG on input a short string. The prover verifies the srWI argument and if this is successful, will execute the remainder of the ZKAoK taking c as the verifier's message. More details follow.

3.1 ZKAoK in the Correlated Randomness Model from OWFs

We first show how to convert a 3-round public-coin ZK argument system in the CRS model with straight-line simulation (based on OWFs) into one that is also an argument of knowledge (with straight-line simulation and straight-line witness extractor) in the correlated randomness model. Let $\Pi_{ZK} = (K, P, V)$ be the ZK argument in the CRS model with a straight-line simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ (e.g. [35]). Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure secret key encryption scheme. Define (K', P', V') in the correlated randomness model as in Figure 2.

Lemma 3.1. *Π_{ZKAoK} is ZKAoK with straight-line simulator and witness extractor in the correlated randomness model.*

Proof. Zero-knowledge: we construct a straight-line simulator $\mathcal{S}' = (\mathcal{S}'_1, \mathcal{S}'_2)$ as follows. $\mathcal{S}'_1$ runs $\mathcal{S}_1$ to generate (σ', τ), generates sk, k, γ_0 as in K, and outputs $s'_P = (\sigma', sk, k, \gamma_0)$, $s'_V = (\sigma', k)$ and $\tau' = (sk, k, \gamma_0)$. $\mathcal{S}'_2(\tau')$ sends $e' \leftarrow \mathsf{Enc}(sk, 0^{|w|})$ and runs $\mathcal{S}_2(\tau)$ to generate messages. We show the indistinguishability by considering a hybrid $\mathsf{Hyb}_{\mathcal{S}}^{\mathsf{Enc}}$ where $\mathcal{S}$ is used to generate the CRS σ' and messages, but the prover sends $e \leftarrow \mathsf{Enc}(sk, w)$ as in Π_{ZKAoK}. Finally, $\mathsf{Hyb}_{\mathcal{S}}^{\mathsf{Enc}}$ outputs the verifier's output. This hybrid is indistinguishable from $\mathsf{Exp}_{\Pi_{ZKAoK}} = \langle P'(w, s_P), V'(s_V)\rangle(x)$ by the zero-knowledge property of Π_{ZK}. It is also indistinguishable from the experiment $\mathsf{Exp}_{\mathcal{S}'} = \langle \mathcal{S}'_2(\tau'), V'(s'_V)\rangle(x)$ running the above simulator by the security of the encryption scheme.

$$\Pi_{ZKAoK} = (K', P', V')$$

$K'(1^\lambda)$:

1. $\sigma \leftarrow K(1^\lambda)$, $sk \leftarrow \mathsf{KeyGen}(1^\lambda)$. Let $k = \mathsf{com}(sk)$ and γ_0 be the decommitment information.
2. K' outputs $s_P = (\sigma, sk, k, \gamma_0)$ and $s_V = (\sigma, k)$.

Execution phase: P' on input (x, w) and private string s_P; V' on input x and private string s_V

1. P' parses $s_P = (\sigma, sk, k, \gamma_0)$, computes $e \leftarrow \mathsf{Enc}(sk, w)$ and sends e to V'.
2. V' parses $s_V = (\sigma, k)$.
3. P' and V' run $\langle P(w'), V\rangle(\sigma, x')$ where $x' = (x, e, k)$ and $w' = (w, sk, \gamma_0)$ to prove that there exists w, sk, γ_0 such that $(x, w) \in R_L$ and $w = \mathsf{Dec}(sk, e)$ and k can be decommitted to sk using γ_0.
4. V' outputs the output of V.

Figure 2: ZKAoK argument protocol Π_{ZKAoK} in the correlated randomness model

Argument of knowledge: we construct a straight-line witness extractor $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2)$ as follows. $\mathcal{E}_1$ generates s_P and s_V as in K and also outputs $\tau = sk$. $\mathcal{E}_2$ runs V' honestly, and if V' accepts, it decrypts and outputs $w = \mathsf{Dec}(sk, e)$. Otherwise, it outputs $\perp$. If V' accepts but $(x, w) \notin R_L$, we have (x', w') fails to satisfy the relation proved by Π_{ZK}. By the soundness of Π_{ZK}, this only occurs with negligible probability. □

If the protocol Π_{ZK} is 3-round and public-coin, the resulting protocol Π_{ZKAoK} is also 3-round and public-coin.

3.2 Simultaneous resettable ZK in the correlated randomness model from OWFs

We now construct a simultaneous resettable ZK argument system in the correlated randomness model based on OWFs. Let $\Pi_{ZKAoK} = (K_{ZKAoK}, P_{ZKAoK}, V_{ZKAoK})$ be a 3-round ZK argument of knowledge protocol in the correlated randomness model with transcript (m_1, c, m_2) where $c \in \{0,1\}^\lambda$ is chosen uniformly at random, a straight-line simulator $\mathcal{S}_{ZKAoK} = (\mathcal{S}_1, \mathcal{S}_2)$, and a straight-line witness extractor $\mathcal{E}_{ZKAoK} = (\mathcal{E}_1, \mathcal{E}_2)$ from Lemma 3.1. Let (P_{WI}, V_{WI}) be a srWI argument (e.g. [14]). Let $\{f_s\}_s$ be a family of pseudorandom functions such that for $s \in \{0,1\}^{\ell_0(\lambda)}$, f_s outputs $c \in \{0,1\}^\lambda$. Let $f : \{0,1\}^{\ell_1(\lambda)} \to \{0,1\}^{\ell_2(\lambda)}$ be a PRG. We define Π_{srZK} as in Figure 3.

The proof of resettable soundness goes as follows. We first consider the experiment with an imaginary protocol Π_F where a truly random function is used instead of the PRF, and the verifier uses an alternate witness for the sim-res WI. We will show that Π_F is resettably sound by contradiction. Finally, we show that the probability that any resetting adversary can prove a false

$$\Pi_{srZK} = (K, P, V)$$

$K(1^\lambda)$:

1. $(\sigma_P, \sigma_V) \leftarrow K_{ZKAoK}(1^\lambda)$, $s \leftarrow U_{\ell_0(\lambda)}$, $d \leftarrow U_{\ell_2(\lambda)}$. Let $t = \mathsf{com}(s)$ and γ be the decommitment information.
2. K outputs $s_P = (\sigma_P, t, d)$ and $s_V = (\sigma_V, s, \gamma, t, d)$.

Execution phase: P on input (x, w) and private string s_P; V on input x and private string s_V

1. P parses $s_P = (\sigma_P, t, d)$, runs $P_{ZKAoK}(x, w, \sigma_P)$ to compute m_1, and sends m_1 to V.
2. V parses $s_V = (\sigma_V, s, \gamma, t, d)$, runs $V_{ZKAoK}(x, \sigma_V)$ to sends $c = f_s(x||m_1)$ on behalf of V_{ZKAoK} to P (running P_{ZKAoK}), and runs $P_{WI}(y, (s, \gamma))$, with $y = (t, c, d, x, m_1)$, proving to P running $V_{WI}(y)$ that one of the following statements hold
   - there exists s' and γ such that t can be decommitted to s' using γ and $c = f_{s'}(x||m_1)$.
   - there exists d' such that $d = f(d')$.
3. If V_{WI} accepts, P continues running $P_{ZKAoK}(x, w, \sigma_P)$ to compute m_2 and send it to V.
4. V runs V_{ZKAoK} on (m_1, c, m_2) and outputs the output of V_{ZKAoK}.

Figure 3: Simultaneous resettable ZK argument protocol Π_{srZK} in the correlated randomness model

theorem in Π_{srZK} is negligible close to that of Π_F through a series of hybrids. This implies that Π_{srZK} is also resettably-sound.

Lemma 3.2. *The protocol Π_{srZK} in the correlated randomness model is resettably-sound.*

Proof. Let F be a truly random function. Consider an experiment Π_F in Figure 4.

Note that s_P generated in K_F is computationally indistinguishable from s_P generated in K. Furthermore, P behaves identically in the execution phase of Π_{srZK} and Π_F. We first prove that in Π_F, for any resetting prover $\widetilde{P}$, the probability that $\widetilde{P}$ makes V_F accepts $x \notin L$ in one of the resetting session is negligible.

Assume for contradiction that there exists a PPT resetting prover $\widetilde{P}$ that can prove a false statement $x \notin L$ in one of the resetting session of Π_F with non-negligible probability p. We construct a non-resetting prover $\widetilde{P}_{ZKAoK}$ that can prove a false statement $x \notin L$ in Π_{ZKAoK} as follows. Given σ_P, $\widetilde{P}_{ZKAoK}$ generates the additional parameters d_0, t, γ, d as in K_F. $\widetilde{P}_{ZKAoK}$ internally runs $\widetilde{P}$ on $s_P = (\sigma_P, t, d)$ in Π_F by either sending c from an honest verifier V_{ZKAoK} or generating c itself to $\widetilde{P}$ on behalf of V. At the beginning of the protocol, $\widetilde{P}_{ZKAoK}$ randomly selects $i \in [T]$ where T is the upper bound on the number of resetting sessions determined by $\widetilde{P}$. $\widetilde{P}_{ZKAoK}$ runs a verifier V_F for $\widetilde{P}$ for all but the ith resetting session. In the ith resetting session, $\widetilde{P}_{ZKAoK}$ passes the first message m_1 from $\widetilde{P}$ to V_{ZKAoK} and c from V_{ZKAoK} to $\widetilde{P}$. $\widetilde{P}_{ZKAoK}$ also provides a srWI argument to $\widetilde{P}$ using d_0 it generated as a witness. If $\widetilde{P}$ resets and sends the same m_1, $\widetilde{P}_{ZKAoK}$

$$\Pi_F = (K_F, P, V_F)$$

K_F:

1. $(\sigma_P, \sigma_V) \leftarrow K_{ZKAoK}(1^\lambda)$, $d_0 \leftarrow U_{\ell_1(\lambda)}$. Let $t = \mathsf{com}(0^{\ell_0(\lambda)})$ and γ be the decommitment information, and $d = f(d_0)$.
2. K_F outputs $s_P = (\sigma_P, t, d)$ and $s_V = (\sigma_V, d_0, t, d)$.

Execution phase: P on input (x, w) and private string s_P; V_F on input x and private string s_V

1. P parses $s_P = (\sigma_P, t, d)$, runs $P_{ZKAoK}(x, w, \sigma_P)$ to compute m_1, and sends m_1 to V_F.
2. V_F parses $s_V = (\sigma_V, s, \gamma, t, d)$, runs $V_{ZKAoK}(x, \sigma_V)$ to send $c = F(x||m_1)$ on behalf of V_{ZKAoK} to P (running P_{ZKAoK}), and runs $P_{WI}(y, d_0)$, with $y = (t, c, d, x, m_1)$, proving to P running $V_{WI}(y)$ that one of the following statements hold
   - there exists s' and γ such that t can be decommitted to s' using γ and $c = f_{s'}(x||m_1)$.
   - there exists d' such that $d = f(d')$.
3. If V_{WI} accepts, P continues running $P_{ZKAoK}(x, w, \sigma_P)$ to compute m_2 and send it to V_F.
4. V_F runs V_{ZKAoK} on (m_1, c, m_2) and outputs the output of V_{ZKAoK}.

Figure 4: Experiment Π_F

will send the same c corresponding to the m_1 sent before. If $\widetilde{P}$ resets and sends a new m_1, $\widetilde{P}_{ZKAoK}$ will either pass m_1 to V_{ZKAoK} and relay c from V_{ZKAoK} to $\widetilde{P}$, or generate c itself. Since $\widetilde{P}$ cannot distinguish H_0 and H_3, it will also prove a false statement $x \notin L$ in one of the resetting session of Π_{srZK} with non-negligible probability $p' = p - \mathsf{negl}(\lambda)$. If $\widetilde{P}_{ZKAoK}$ guesses correctly which m_1 to pass to V_{ZKAoK} for $\widetilde{P}$ to complete the protocol for $x \notin L$, then it will convince V_{ZKAoK} to accept a false statement. This happens with probability $1/T$ where T is polynomial in the security parameter. Thus, $\widetilde{P}_{ZKAoK}$ can prove a false statement in Π_{ZKAoK} with probability p'/T which is non-negligible (in the security parameter). This contradicts the soundness of Π_{ZKAoK}. Hence, the protocol Π_F is resettably sound.

We then show that the original protocol Π_{srZK} is also resettably-sound by considering the following hybrid experiments whose outputs come from the joint distribution of the output of the verifier and the witness extractor $\mathcal{E}_2$: let $\widetilde{P}$ be a resetting prover.

H_0: This hybrid experiment runs the protocol Π_{srZK} $\langle \widetilde{P}, V \rangle$ with σ replaced by σ' generated in $(\sigma', \tau) \leftarrow \mathcal{E}_1(1^\lambda)$. If $\widetilde{P}$ convinces V to accept x in a resetting session, take the transcript (m_1, c, m_2) and run $\mathcal{E}_2(x, \tau, (m_1, c, m_2))$ to extract a witness w. Output (x, w) where $w = \perp$ if $\mathcal{E}_2$ fails to extract a witness, or abort if V rejects.

H_1**:** This hybrid is the same as H_0 except that d is generated by first sampling $d_0 \leftarrow U_{\ell_1(\lambda)}$ and computing $d = f(d_0)$, and also giving d_0 to V. By the property of the PRG, this hybrid is indistinguishable from H_0.

H_2**:** This hybrid is the same as H_1 except that V runs $P_{WI}(y, d_0)$ instead of $P_{WI}(y, s)$.

Claim. *Hybrid H_1 and H_2 are indistinguishable.*

Proof. Suppose a distinguisher D can distinguish H_1 and H_2 with non-negligible probability q. We construct $\widetilde{V}^*$ that can distinguish the interaction with $P_{WI}(y, (s, \gamma))$ and $P_{WI}(y, d_0)$ where $y = (t, c, d, x, m_1)$ with probability q as follows. $\widetilde{V}^*$ samples d_0, s and computes t, d, γ, m_1, c as in H_1, H_2. It then runs D while feeding the messages from $P_{WI}(x, (s, \gamma))$ or $P_{WI}(x, d_0)$ after sending c. □

H_3**:** This hybrid is the same as H_2 except that V is no longer given s, γ and instead sends $c \leftarrow F(x||m_1)$, where F is a truly random function. By the property of the PRF, this hybrid is indistinguishable from H_2.

H_4**:** This hybrid is the same as H_3 except that s is no longer generated and t is a commitment of $0^{\ell_0(\lambda)}$ instead of s. By the hiding property of the commitment scheme, this hybrid is indistinguishable from H_3.

Let $\epsilon_i = \epsilon_i(\lambda)$ be the (negligible) probability of distinguishing H_{i-1} from H_i for each i. Note that the Hybrid H_4 is the same as Hybrid H_0 but with protocol Π_F $\langle \widetilde{P}, V_F \rangle$ instead of Π_{srZK}. As we proved above, the probability that V_F accepts x but $\mathcal{E}_2$ cannot extract the witness $\Pr[H_4 = (x, \perp)]$ is negligible $\epsilon_0 = \epsilon_0(\lambda)$ by the property of the witness extractor $\mathcal{E}_{ZKAoK}$. Thus, $\Pr[H_0 = (x, \perp)] \leq \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4 = \epsilon'$ is negligible. Note that by the property of the PRF, the extractor $\mathcal{E}_2$ can fail to extract in the Hybrid H_0 with at most negligible probability $\epsilon_5 = \epsilon_5(\lambda)$. Therefore, for any resetting prover $\widetilde{P}$, the probability that $\widetilde{P}$ convinces V to accept $x \notin L$ is at most $\epsilon_5 + \epsilon'$ which is negligible. □

Lemma 3.3. *The protocol Π_{srZK} is resettable ZK in the correlated randomness model with a straight-line simulator.*

Proof. Now we construct a zero-knowledge simulator $\mathcal{S}$ against a resetting verifier $\widetilde{V}$. $\mathcal{S}$ runs the simulator $\mathcal{S}_1$ for Π_0 to generate $(\sigma'_P, \sigma'_V, \tau)$. It gives σ'_V to $\widetilde{V}$ instead of honestly generated σ_V in the correlated randomness generation phase. $\mathcal{S}$ generates s, γ, t, d honestly. In the execution phase, $\mathcal{S}$ runs $\mathcal{S}_2(\tau)$ to generate m_1. $\mathcal{S}$ receives c from $\widetilde{V}$ and runs V_{WI} honestly. $\mathcal{S}$ continues running $\mathcal{S}_2(\tau)$ to generate m_2, and records the transcript (m_1, c, m_2) for later use. When $\widetilde{V}$ resets, $\mathcal{S}$ sends the recorded m_1. If $\widetilde{V}$ sends the previously seen c, $\mathcal{S}$ will send the corresponding recorded m_2. If $\widetilde{V}$ sends a new c, $\mathcal{S}$ aborts.

Claim. *The views of $\widetilde{V}$ interacting with an honest prover P and with $\mathcal{S}$ are indistinguishable.*

Proof. First consider a hybrid experiment where we replace $(\sigma_P, \sigma_V) \leftarrow K_{ZKAoK}(\lambda)$ with (σ'_P, σ'_V) from $(\sigma'_P, \sigma'_V, \tau) \leftarrow \mathcal{S}_1(\lambda)$, and m_1 and m_2 are generated by $\mathcal{S}_2(\tau)$. By the ZK property of Π_{ZKAoK}, $\widetilde{V}$ cannot distinguish this hybrid experiment from running Π_{ZK}. Now suppose that $\widetilde{V}$ can distinguish the hybrid experiment from running against the simulator $\mathcal{S}$ with non-negligible probability q.

The only difference between the hybrid and the interaction with $\mathcal{S}$ is when $\widetilde{V}$ resets and sends different c. In this case, $\widetilde{V}$ needs to provide a sim-res WI for $y_1 = (t, c_1, d, x, m_1)$ and $y_2 = (t, c_2, d, x, m_1)$ with $c_1 \neq c_2$. We construct a PPT $\widetilde{P}_{WI}$ that can prove a false statement $y = (t, c, d, x, m_1) \notin R_{WI}$ as follows: $\widetilde{P}_{WI}$ generates the setup strings and runs the interaction above (with $(\mathcal{S}_1, \mathcal{S}_2)$). $\widetilde{P}_{WI}$ randomly chooses the session of sim-res WI to pass the WI prover message from $\widetilde{V}$ to V_{WI}. Since d is chosen uniformly at random, except with negligible probability, $d \neq f(d')$ for any $d' \in \{0,1\}^{\ell_1(\lambda)}$. Thus, if $y_1, y_2 \in R_{WI}$, there exists s' such that $t = \mathsf{com}(s')$ and $c_1 = c_2 = f_{s'}(x||m_1)$. At least one of the c's will make $y = (t, c, d, x, m_1) \notin R_{WI}$. The probability that $\widetilde{P}_{WI}$ passes the sim-res WI messages corresponding to such y to V_{WI} is at least $1/T'$, where T' is the number of different c's sent by $\widetilde{V}$. Therefore, the probability that $\widetilde{P}_{WI}$ can prove $y \notin R_{WI}$ is at least q/T', which contradicts the soundness of (P_{WI}, V_{WI}). □

Since $(\mathcal{S}_1, \mathcal{S}_2)$ is straight-line, $\mathcal{S}$ is also straight-line. □

Lemmas 3.2 and 3.3 together gives us the following theorem:

Theorem 3.4. *Assuming the existence of OWFs, there exists a simultaneous resettable ZK argument protocol in the correlated randomness model with a straight-line simulator.*[5]

4 MPC in the Correlated Randomness Model

In this section, we construct a UC-secure MPC protocol in the correlated randomness model based on OWFs. The key ingredients are an MPC protocol in the OT-hybrid model UC-secure against an adversary corrupting all but one party, and a protocol UC-realizing unbounded number of OTs in the correlated randomness model. In [32], Ishai, Prabhakaran and Sahai introduce the IPS compiler which combines an MPC with an honest majority and a protocol secure against semi-honest adversary in the OT-hybrid model to get a protocol UC-secure against malicious adversaries in the setting of no honest majority. One of their main applications, by applying the compiler to a variant of the protocol in [16], gives an MPC protocol in the OT-hybrid model that is UC-secure against a malicious adversary corrupting all but one party, assuming only a PRG. Unlike their main result, however, this MPC protocol requires a large number of OTs, proportional to the circuit size.

To address the number of OTs required, we then construct a UC-secure protocol for unbounded number of OTs in the correlated randomness model. The first attempt is to use Beaver's OT extension [6] from a bounded number of OTs which can be generated using correlated randomness [5]. The problem with this approach is that we can only get polynomial number (in the number of original OTs) of OTs for some fixed polynomial known in advance. However, in our protocol, we would require (an apriori) unknown number of OTs to be generated from the initial OTs – this is because the number of OTs needed depends on the number of times the hardware token is executed.

To get around this problem, we do as follows. We have the sender construct a super-polynomial size Yao's garbled circuit that computes the OTs. Of course, the sender cannot compute this entire garbled circuit. So, instead of sending the garbled circuit to the receiver, the sender commits to the first layer of the garbled circuit and the seed for the PRF that is used to generate the rest of the garbled circuit. When the receiver queries for the ith OT, the sender sends a section of the garbled circuit that suffices to compute the output followed by the ZK argument that it is consistent with committed values. However, this section of the circuit is now of polynomial size. This technique is

[5] Our ZK argument protocol also has a straight-line witness extractor, but it is not necessary for our applications.

similar in spirit to the GGM [24] technique for constructing a PRF. We now present more details of this construction.

4.1 Beaver's OT Extension

Before we construct a UC-secure protocol computing unbounded number of OTs, we first recall Beaver's construction [6]. Beaver considers two notions of OT.

- $\frac{1}{2}$OT: the sender S has x_0 and x_1. At the end of the protocol, the receiver learns (b, x_b) for a random bit b, the sender learns nothing about b.
- $\binom{2}{1}$OT: the sender has x_0 and x_1, the receiver has a bit b. At the end of the protocol, the receiver learns x_b, the sender learns nothing about b.

In [5], Beaver shows that $O(n)$ instances of $\binom{2}{1}$OT can be generated from $O(n^2)$ instances of $\frac{1}{2}$OT.

In [6], the sender constructs a garbled circuit that takes a short input for a PRG, then expands it to a long string. Each bit of the string is used to select one of each pair of the sender's inputs. In order to get a garbled input corresponding to the receiver's seed and the garbled circuit, the sender and the receiver only need to perform a small number of OTs for each bit of the short input. This OT extension technique extends λ $\binom{2}{1}$OTs to $poly(\lambda)$ $\frac{1}{2}$OTs. This small number of OTs can be precomputed [5] as part of the correlated randomness. While this OT extension results in $\frac{1}{2}$OT, smaller number (but still polynomial) of $\binom{2}{1}$OT can also be generated using the same number of starting OTs.

$$\mathsf{OT}^n = (K, S, R)$$

$K(1^\lambda)$:

1. $(\sigma_S, \sigma_R) \leftarrow K_0(1^\lambda)$ and $s \leftarrow \{0,1\}^\lambda$.
2. K outputs $s_S = (\sigma_S)$ and $s_R = (\sigma_R, s)$.

Execution phase: S on input $X = \{x_0^i, x_1^i\}_{i\in[n]}$, $x_b^i \in \{0,1\}$, and private string σ_S; R on input $s \in \{0,1\}^\lambda$, and private string σ_R

1. Let $C_X(s)$ be a circuit that computes $(r_1, \ldots, r_n) = r = G(s)$ and outputs $((r_1, x_{r_1}^1), \ldots, (r_n, x_{r_n}^n))$. Let $(G, \pi) \leftarrow \mathsf{GC}(C)$. S sends G to R.
2. For each $i \in [\lambda]$, R runs $R_0(i, s_i, \sigma_R)$ to query $\pi(i, s_i)$ from S running $S_0(i, \pi(i,0), \pi(i,1))$. Let $\overline{s} = \pi(s) = (\pi(1, s_1), \ldots, \pi(\lambda, s_\lambda))$.
3. R outputs $\mathsf{GE}(C, \pi(s))$.

Figure 5: Beaver's OT Protocol

Let $\mathcal{G} = (\mathsf{GC}, \mathsf{GE})$ be Yao's garbling circuit scheme where each gate and wire are encrypted. For a circuit C, let $(G, \pi) \leftarrow \mathsf{GC}(C)$ consist of a garbled circuit C and garbled input function

π such that $\pi(i, x)$ is a garbled input for ith position input $x \in \{0,1\}$. Let $n = poly(k)$. Let $G : \{0,1\}^\lambda \leftarrow \{0,1\}^n$ be a PRG. Let $\mathsf{OT}_0 = (K_0, S_0, R_0)$ be a protocol for λ $\binom{2}{1}$OTs in the correlated randomness model. The Beaver's OT extension protocol OT^n is described in Figure 5.

4.2 Unbounded Number of OTs

We now construct a UC-secure protocol computing unbounded number of OTs in the correlated randomness model assuming only OWFs. We consider the following modification to the OT extension above. Instead of a PRG, we use a PRF to generate a pseudorandom r_i for any $i \in \{0,1\}^\lambda$ using seed s_1 given to the sender on input $s_2||i$ where s_2 plays the same role as s in Figure 5. Each r_i can be used to select the sender's input in the same way as in Beaver's protocol. However, the entire circuit (for all i) will have exponential size. To get around this problem, for each i, the sender only sends a garbled circuit corresponding to a subcircuit that suffices to compute an output based on the sender's ith input and r_i. We also use UC-secure ZK argument and commitment to ensure that malicious parties cannot deviate from the protocol. Since the whole garbled circuit is fixed given the committed values, the sender cannot change the circuit and still successfully provide the ZK argument. Sender security is proved by arguing that the receiver does not learn more than the intended output by the property of the garbled circuits.

The unbounded OT protocol construction, $\mathsf{OT}^{\mathsf{unbounded}}$, is provided in Figure 6. Let $\{f_s\}_{s\in\mathcal{S}}$ be a family of PRFs with seed space $\mathcal{S}$ and $f_s : \{0,1\}^{2\lambda} \to \{0,1\}^{p_1(\lambda)}$ for some polynomial p_1. Each execution of $\mathsf{OT}^{\mathsf{unbounded}}$ gives $n = p_1(\lambda)$ $\frac{1}{2}$OTs similar to Beaver's, which can be turned into $p_2(\lambda)$ $\binom{2}{1}$OT for a smaller polynomial p_2. A computational-bounded receiver can execute $\mathsf{OT}^{\mathsf{unbounded}}$ polynomially many times for any polynomial not known at construction time (as long as the polynomial is smaller than 2^λ).

Theorem 4.1. *Assuming OWFs, the protocol in Figure 6 is a UC-secure protocol computing unbounded number of OTs in the correlated randomness model.*

Proof. (Sketch) We construct a simulator Sim in the ideal world running Adv as follows. Sim uses the simulator Sim_0 for OT_0 to generate the correlated randomness for OT_0, and the simulator Sim_{ZK} for Π_{ZK} to generate the CRS for Π_{ZK}. In the case that Adv corrupts the receiver, Sim computes $(c_i, \gamma_i) \leftarrow \mathsf{com}(0)$ for $i = 1, 3$. Sim generates s_2 and γ_2 honestly. Sim computes $(G, \pi) = \mathsf{GC}(C^*; r_0^*)$ for zero circuit $C^*(s)$.

When Adv queries the garbled input for $\overline{s}$, Sim runs Sim_0 to extract s^*, γ^*. For each i execution, Sim queries the OT functionality for the jth output $((r_1, x_{r_1}^{i,1}), \ldots, (r_n, x_{r_n}^{i,n}))$. It generates $C_i^*(s)$ that outputs $((r_1, x_{r_1}^{i,1}), \ldots, (r_n, x_{r_n}^{i,n}))$ if $s = s_2$ and $(G_i, \pi) = \mathsf{GC}(C; r_0^*, r_i^*)$ such that π is consistent with G above. Finally, Sim runs Sim_{ZK} instead of the ZK prover.

In the case that Adv corrupts the sender, Sim computes $(c_2, \gamma_2) \leftarrow \mathsf{com}(0)$, and generates s_i, γ_i for $i = 1, 3$ honestly. Sim uses Sim_0 to extract both garbled inputs for each bit of seed s. Sim then uses Sim_{ZK} in the following step. Given the ith garbled input, it can learn both sender's inputs, by repeatedly querying the circuit, and send them to the OT functionality.

The indistinguishability proof is through a series of hybrids. First, we use the simulator Sim_{ZK} for the ZK argument instead of the ZK prover. Then, using the hiding property of com, we replace the commitment of the seed to the commitment of zero. Sim then generates the correlated randomness using Sim_0, thereby learning the private string for both parties. Finally, Sim extracts the sender's garbled inputs and the receiver's seed and proceed as above, using the security of the garbled circuit. □

$$\mathsf{OT}^{\mathsf{unbounded}} = (K, S, R)$$

$K(1^\lambda)$:

1. $(\sigma_S, \sigma_R) \leftarrow K_0(1^\lambda)$, $\sigma_{ZK} \leftarrow K_{ZK}(1^\lambda)$, $s_1, s_2, s_3 \leftarrow S$. For $i = 1, 2, 3$, let $(c_i, \gamma_i) \leftarrow \mathsf{com}(s_i)$ with decommitment information $\gamma_i \in \{0,1\}^{\ell(\lambda)}$.
2. K outputs $s_S = (\sigma_S, \sigma_{ZK}, s_1, \gamma_1, s_3, \gamma_3, c_2)$ and $s_R = (\sigma_R, \sigma_{ZK}, s_2, \gamma_2, c_1, c_3)$.

Before 1st Execution phase: S on private string s_S; R on private string s_R

1. S parses $s_S = (\sigma_S, \sigma_{ZK}, s_1, \gamma_1, s_3, \gamma_3, c_2)$. Let $C(s, i) = C_{X,s_1,c_2}(s, i)$ be a circuit that outputs $(r_1, \ldots, r_n) = F_{s_1}(s||i)$. Let $(G, \pi) = \mathsf{GC}(C; f_{s_3}(0))$.
2. R parses $s_R = (\sigma_R, \sigma_{ZK}, s_2, \gamma_2, c_1, c_3)$. R runs $R_0(s_2, \sigma_R)$ to query $\pi(s_2)$ from S running $S_0(\pi, \sigma_S)$. R records the output $\overline{s}$.
3. R runs $P_{ZK}(\sigma_{ZK}, (s_2, \gamma_2))$ to prove to S running $V_{ZK}(\sigma_{ZK})$ that c_2 can be decommitted to s_2 using γ_2 and R queries for s_2. If V_{ZK} rejects, S aborts.

ith Execution phase: S on input $X_i = \{x_0^{i,j}, x_1^{i,j}\}_{j \in [n]}$, where $x_b^{i,j} \in \{0,1\}$ and private string s_S; R on private string s_R

1. S parses $s_S = (\sigma_S, \sigma_{ZK}, s_1, \gamma_1, s_3, \gamma_3, c_2)$. Let $C_i(s) = C_{i,X,s_1,c_2}(s)$ be a circuit that first computes $r = C(s, i)$, and if $r = (r_1, \ldots, r_n) \neq \perp$, outputs $((r_1, x_{r_1}^{i,1}), \ldots, (r_n, x_{r_n}^{i,n}))$; otherwise, outputs $\perp$. S computes $(G_i, \pi) = \mathsf{GC}(C_i; f_{s_3}(0), f_{s_3}(i))$ such that $f_{s_3}(0)$ is used for input wires (for consistency of π) and $f_{s_3}(i)$ is used for the rest. S sends G_i to R
2. R parses $s_R = (\sigma_R, \sigma_{ZK}, s_2, \gamma_2, c_1, c_3)$; S runs $P_{ZK}(\sigma_{ZK}, (s_1, s_3, \gamma_1, \gamma_3)))$ to prove to R running $V_{ZK}(\sigma_{ZK})$ that G_i is generated using s_1 and s_3, which decommitted to c_1 and c_3 using γ_1 and γ_3, respectively. R aborts if V_{ZK} rejects.
3. R outputs $\mathsf{GE}(G_i, \overline{s})$.

Figure 6: UC-Secure Unbounded OT Protocol

4.3 MPC in the OT-hybrid Model

In [32], Ishai *et al.* construct a compiler that turns an MPC protocol that is secure against adversary corrupting less than half of the parties (honest majority) into a UC-secure MPC protocol in the OT-hybrid model. They apply this transformation to a variant of the MPC protocol from [16] to obtain the following theorem.

Theorem 4.2 (Theorem 3 in [32])**.** *Assuming a PRG, for any $n \geq 2$, there exists an n-party constant-round MPC protocol in the OT-hybrid model that is UC-secure against an active adversary adaptively corrupting at most $n-1$ parties.*

Combining this theorem with our UC-secure protocol for unbounded number of OTs in the correlated randomness model, we get the following corollary.

Corollary 4.3. *Assuming OWFs, for any $n \geq 2$, there exists an n-party constant-round MPC protocol in the correlated randomness model that is UC-secure against an active adversary adaptively corrupting at most $n-1$ parties.*

5 Corrupted Token Model

We consider a generalization of the Katz's tamper-proof token model [33] where tokens can be corrupted by adversaries even when they are created by honest parties. Our model is inspired by the real world application where honest users cannot create tokens themselves. They instead rely on a number of manufacturers, some of whom could be malicious. Thus, the secrets embedded in the token description can be revealed to the adversary. Furthermore, the adversary can replace the tokens with ones of its choice.

$\mathcal{F}_{\mathsf{token}}$

Upon receiving $(\mathsf{create}, \mathsf{sid}, P_j, \Pi)$ from P_i with $i \neq j$:

1. If there is no tuple of the form $(\mathsf{sid}, P_i, P_j, \star)$, store $(\mathsf{sid}, P_i, P_j, \Pi)$.
2. Send $(\mathsf{done}, \mathsf{sid})$ to P_i and send $(\mathsf{create}, \mathsf{sid}, P_i)$ to P_j.

Upon receiving $(\mathsf{execute}, \mathsf{sid}, P_i, inp)$ from P_j with $i \neq j$:

1. Find the unique stored tuple $(\mathsf{sid}, P_i, P_j, \Pi)$. If no such tuple exists, abort.
2. Run $\Pi(inp)$ and let out be the output.
3. Send (sid, out) to P_j.

Figure 7: Token Functionality $\mathcal{F}_{\mathsf{token}}$

5.1 Katz's Stateless Tamper-Proof Token Functionality $\mathcal{F}_{\mathsf{token}}$

Our model is based on the stateless version of Katz's tamper-proof token model [33] defined in Figure 7. In this model, each user can create a stateless token by sending its description to $\mathcal{F}_{\mathsf{token}}$. The token is tamper-proof in the sense that the receiver can only access it through $\mathcal{F}_{\mathsf{token}}$ functionality in a black-box manner. We consider the case of stateless tokens where the tokens do not keep information between each access and use the same random tape. Hence, without loss of generality, we can assume that the function computed by the token is deterministic. In this case, we may represent the function with a circuit.

$\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$

Upon receiving $(\mathsf{create}, \mathsf{sid}, P_j, \Pi)$ from P_i with $i \neq j$:

1. If there is no tuple of the form $(\mathsf{sid}, P_i, P_j, \star)$, store $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{creating})$.
2. Send $(\mathsf{create}, \mathsf{sid}, P_i, P_j)$ to Adv.

Upon receiving $(\mathsf{interrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{creating})$. If no such tuple exists, abort.
2. Send $(\mathsf{interrupted}, P_i)$ to P_j.

Upon receiving $(\mathsf{notinterrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{creating})$. If no such tuple exists, abort.
2. Store $(\mathsf{sid}, P_i, P_j, \Pi)$.
3. Send $(\mathsf{done}, \mathsf{sid})$ to P_i and send $(\mathsf{create}, \mathsf{sid}, P_i)$ to P_j.

Upon receiving $(\mathsf{execute}, \mathsf{sid}, P_i, inp)$ from P_j with $i \neq j$:

1. Find the unique stored tuple $(\mathsf{sid}, P_i, P_j, \Pi)$. If no such tuple exists, abort.
2. Run $\Pi(inp)$ and let out be the output.
3. Send (sid, out) to P_j.

Figure 8: Token with Abort Functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$

Our protocol will UC-realize a variant of $\mathcal{F}_{\mathsf{token}}$, called $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$, described in Figure 8 in which the adversary is notified whenever a party creates a token and can choose to interrupt its delivery. The receiver will not receive the token, but will be notified with the special message $\mathsf{interrupted}$. In such a case, the receiver aborts the protocol. This (otherwise unavoidable) change can be avoided

by restricting the adversary to corrupt less than half of the corruptible tokens, which will allow the receiver to compute the output using the remaining uncorrupted tokens, but will weaken the threshold of corruptions tolerated.

$\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$

Upon receiving $(\mathsf{create}, \mathsf{sid}, P_j, \Pi)$ from P_i with $i \neq j$:

1. If there is no tuple of the form $(\mathsf{sid}, P_i, P_j, \star, \star)$, store $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{creating})$.
2. Send $(\mathsf{create}, \mathsf{sid}, P_i, P_j)$ to Adv.

Upon receiving $(\mathsf{corrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find the stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, \mathsf{creating})$. If no such tuple exists, abort.
2. Send Π to Adv.

Upon receiving $(\mathsf{replace}, \mathsf{sid}, P_i, P_j, \Pi^*, \mathsf{state}_0)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, \mathsf{creating})$. If no such tuple exists, abort.
2. Store $(\mathsf{sid}, P_i, P_j, \Pi^*, \mathsf{state}_0)$, send $(\mathsf{done}, \mathsf{sid})$ to P_i, send $(\mathsf{create}, \mathsf{sid}, P_i)$ to P_j.

Upon receiving $(\mathsf{notcorrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, \mathsf{creating})$. If no such tuple exists, abort.
2. Store $(\mathsf{sid}, P_i, P_j, \Pi, \bot)$, send $(\mathsf{done}, \mathsf{sid})$ to P_i and send $(\mathsf{create}, \mathsf{sid}, P_i)$ to P_j.

Upon receiving $(\mathsf{execute}, \mathsf{sid}, P_i, inp)$ from P_j with $i \neq j$:

1. Find the stored tuple $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{state})$. If no such tuple exists, abort.
2. Run $\Pi(\mathsf{state}, inp)$ and let (state', out) be the output. If $\mathsf{state} \neq \bot$, set $\mathsf{state} = \mathsf{state}'$.
3. Send (sid, out) to P_j.

Upon receiving $(\mathsf{read}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find the unique stored tuple $(\mathsf{sid}, P_j, P_i, P, \mathsf{state})$. If no such tuple exists, abort.
2. Send $(\mathsf{sid}, \mathsf{state})$ to Adv.

Figure 9: Token Functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$

5.2 Corruptible Tamper-Proof Token Functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$

We generalize the tamper-proof token model to accommodate such a scenario by allowing an adversary to corrupt each token upon its creation. We define corruptible tamper-proof token functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ in Figure 9 by modifying $\mathcal{F}_{\mathsf{token}}$ as follows. Every time a user sends create command to the functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$, it first notifies the adversary and waits for one of two possible responses. The adversary may choose to learn the description of the token, and replace it with another (possibly stateful) token of its choice. We call the token chosen by the adversary a *corrupted* token. Alternately, the adversary may ignore the creation of that token, and therefore, that token creation is completed successfully and in this case, the adversary will not learn the description of the token. After uncorrupted tokens are created, they are tamper-proof in the same sense as in Katz's model. The stateful program for the corrupted token can be represented by a Turing machine.

In the case that the adversary chooses not to corrupt any token created by honest users, our model is identical to the model of Katz. Thus, our model generalizes the standard tamper-proof token model. In this work, we show that we can achieve UC-secure 2PC/MPC in the corrupted token model allowing the adversary to corrupt one party and all but one token generated by every honest party.

6 A Compiler to the Corrupted Token Model

6.1 Protocol for Corruptible Tokens

In this section, we describe a multi-party protocol that the n corruptible tokens will run in order to emulate the Katz' stateless token functionality.

Let $(K_{srZK}, P_{srZK}, V_{srZK})$ be a simultaneous resettable ZK argument in the correlated randomness model with straight-line simulator. Let $\mathcal{S} = (\mathsf{share}, \mathsf{recon})$ be an n out of n secret sharing scheme. Let Γ_0 be a UC-secure MPC protocol in the correlated randomness model for functionality $\mathcal{F}$ described in Figure 10.

$\mathcal{F}$

On input $((\Pi_1, x_1), \ldots, (\Pi_n, x_n))$

1. If $x_i \neq x_j$ for some $i \neq j$, output $\perp$. Otherwise, let x be the common input.
2. $\Pi = \mathsf{recon}(\Pi_1, \ldots, \Pi_n)$.
3. Output $\Pi(x)$.

Figure 10: Function $\mathcal{F}$

We define a multi-party protocol $\Gamma = \Gamma(\Pi)$ on input $(x_1, \ldots, x_n)$ to compute $\Pi(x)$ when $x = x_i$ for all $i \in [n]$ in Figure 11.

$\Gamma(\Pi)$

Setup:

1. Let $(\Pi_1, \ldots, \Pi_n) \leftarrow \mathsf{share}(\Pi)$. Let $c_i = \mathsf{com}(\Pi_i)$ with decommitment information d_i for $i \in [n]$.
2. For $i \neq j \in [n]$, let $(\sigma_{i,j,P}, \sigma_{i,j,V}) \leftarrow K_{srZK}(1^\lambda)$ be the correlated randomness for the simultaneous resettable ZK argument with prover P_i and verifier P_j.
3. Let $(\sigma_1, \ldots, \sigma_n) \leftarrow \mathsf{SetUp}_{\Gamma_0}(1^\lambda)$ be the correlated randomness for Γ_0.
4. For $i \in [n]$, send $\Sigma_i = (\Pi_i, \sigma_i, c_i, d_i, \{c_j, \sigma_{i,j,P}, \sigma_{j,i,V}\}_{j \in [n] \setminus \{i\}})$ to Party P_i.

Execution: Party P_i on input x_i, correlated string Σ_i and a random tape $R_i = (s_i, r_i)$

1. For each $i \in [n]$, P_i commits to its input x_i and its PRF seed s_i using randomness $r_{i,x} = PRF_{s_i}(x_i)$ and r_i, respectively. P_i sends its *determining message* $M_i = (\mathsf{com}(x_i; r_{i,x}) || \mathsf{com}(s_i; r_i))$ *to all* P_j, $j \neq i$.
2. *For each* $i \in [n]$, P_i *computes* $R'_i = PRF_{s_i}(M_1 || \ldots || M_n)$ *consisting of* $R'_i[0]$ *for running* Γ_0 *and* $R'_i[1]$ *for running* (P_{srZK}, V_{srZK}).
3. *For each* $i \in [n]$, P_i *executes as the* i*th party in* Γ_0 *where* P_i *follows* k*th round message* $m_{i,k}$ *by running* P_{srZK} *to prove that there exists* $\alpha_i = (x_i, s_i)$ *and* Π_i *such that*
   (a) $M_i = (M_i[0], M_i[1])$ *where* $M_i[0]$ *can be decommitted to* x_i *and* $M_i[1]$ *can be decommitted to* s_i*;*
   (b) c_i *can be decommitted to* Π_i*;*
   (c) $m_{i,k}$ *is correctly computed using* $R'_i[0]$ *in* Γ_0 *with* (Π_i, x_i) *as an input where* $R'_i = PRF_{s_i}(M_1 || \ldots || M_n)$.

Figure 11: Multi-party protocol $\Gamma(\Pi)$ computing Π

6.2 Realizing a Tamper-Proof Token with Corruptible Tokens

Now we are ready to describe our protocol realizing the tamper-proof token with n corruptible tokens. To compute Π, the corruptible tokens are given the setup parameters for the MPC protocol $\Gamma(\Pi)$. Up on execution with input x_i, they will run $\Gamma(\Pi)$ to compute $\Pi(x)$ only if $x = x_i$ for all $i \in [n]$. We let Γ_i be the Turing machine computing messages Party P_i in Γ sends to other P_j, $j \in [n] \setminus \{i\}$ in each round of Γ. Since our token are stateless, Γ_i takes as input a state state_{k-1} which stores internal memory of P_i in round $k-1$ together with in which consists of all incoming messages P_i receives in round $k-1$. Γ_i outputs a new state state_k and outgoing messages for round k.

Formally, $\Gamma_i(\mathsf{state}_{k-1}, in_{k-1}) = (\mathsf{state}_k, out_k)$ where state_k is the internal state of P_i in round k

and

- $in_{k-1} = \{m_{j,i,k-1}\}_{j\in[n]\setminus\{i\}}$ where $m_{j,i,k-1}$ is the incoming message from Party P_j to Party P_i in round $k-1$. $m_{j,i,k-1} = \perp$ if P_i does not receive a message from P_j in round $k-1$.
- $out_k = \{m_{i,j,k}\}_{j\in[n]\setminus\{i\}}$ where $m_{i,j,k}$ is the outgoing message from Party P_i to Party P_j in round k. $m_{i,j,k} = \perp$ if Party P_i does not send a message to Party P_j in round k.
- Let $\mathsf{state}_0 = \perp$ and in_0 be P_i's input for Γ.
- If P_i terminates at the end of round t, $\Gamma_i(\mathsf{state}_t, in_t) = (\mathsf{done}, output)$ where $output$ is P_i's output for Γ.

In order to protect Γ_i's state state_k when a token is sent to a malicious party, we use a symmetric key encryption scheme with a secret key embedded in the token to encrypt a state state before outputting. Let $\mathbb{S} = (\mathsf{SetUp}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric key encryption scheme. Let $\overline{\mathsf{state}}$ denote an encryption of state using $\mathbb{S}$. Let s_i consists of all information embedded in the ith token. Formally, we define $T_i = T(s_i)$ in Figure 12.

On input $(\mathsf{Initialize}, x_i)$

1. Parse $s_i = (i, sk_i, R_i, \Pi_i, \sigma_i, c_i, d_i, \{c_j, \sigma_{i,j,P}, \sigma_{j,i,V}\}_{j\in[n]\setminus\{i\}})$.
2. Initiate Γ_i on setup parameters $(\Pi_i, \sigma_i, c_i, d_i, \{c_j, \sigma_{i,j,P}, \sigma_{j,i,V})\}_{j\in[n]\setminus\{i\}})$, random tape R_i
3. Let $\Gamma_i(\perp, x_i) = (\mathsf{state}, out)$.
4. Output $(\mathsf{Enc}(sk_i, \mathsf{state}), out)$.

On input $(\overline{\mathsf{state}}, in)$

1. Parse $s_i = (i, sk_i, R_i, \Pi_i, \sigma_i, c_i, d_i, \{c_j, \sigma_{i,j,P}, \sigma_{j,i,V}\}_{j\in[n]\setminus\{i\}})$.
2. Initiate Γ_i on setup parameters $(\Pi_i, \sigma_i, c_i, d_i, \{c_j, \sigma_{i,j,P}, \sigma_{j,i,V})\}_{j\in[n]\setminus\{i\}})$, random tape R_i.
3. Decrypt $\mathsf{state} = \mathsf{Dec}(sk_i, \overline{\mathsf{state}})$, abort if fail.
4. Let $\Gamma_i(\mathsf{state}, in) = (\mathsf{state}', out)$
5. Output $(\mathsf{Enc}(sk_i, \mathsf{state}'), out)$.

Figure 12: Token $T_i = T(s_i)$

Finally, we define the protocol Λ in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$-hybrid model realizing $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ in Figure 13.

6.3 Proof of Security

Let $\mathcal{S}_{srZK} = (\mathcal{S}_1, \mathcal{S}_2)$ be the straight-line simulator for the simultaneous resettable ZK, and Sim_{MPC} be the UC simulator for the UC-secure MPC.

Λ

To create a token running Π for P_j, P_i does the following:

1. Generate the setup parameters for $\Gamma(\Pi)$: $(\Pi_k, \sigma_k, c_k, d_k, \{c_l, \sigma_{k,l,P}, \sigma_{l,k,V})\}_{l\in[n]})$ for $k \in [n]$ as defined in Figure 11.
2. Generate secret keys for decrypting share/state $sk_k \leftarrow \mathsf{KeyGen}(1^\lambda)$ for all $k \in [n]$.
3. Let $s_k = (k, sk_k, R_k, \Pi_k, \sigma_k, c_k, d_k, \{c_l, \sigma_{k,l,P}, \sigma_{l,k,V}\}_{l\in[n]})$.
4. Send $(\mathsf{create}, \mathsf{sid}_k, P_j, T_k)$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ where $T_k = T(s_k)$ for $k \in [n]$.

To execute a token running Π sent by P_i, P_j does the following:

1. For $k \in [n]$, initialize T_k by sending $(\mathsf{execute}, \mathsf{sid}_k, S, (\mathsf{initialize}, inp))$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ to compute $T_k(\mathsf{initialize}, inp) = (\overline{\mathsf{state}}_k, out_k)$.
2. While $\overline{\mathsf{state}}_k \neq \mathsf{done}$ for all $k \in [n]$, for $k \in [n]$
   (a) Parse $out_k = \{m_{k,l}\}_{l\neq k}$. Let $in_k = \{m_{l,k}\}_{l\neq k}$.
   (b) Send $(\mathsf{execute}, \mathsf{sid}_k, P_i, (\overline{\mathsf{state}}_k, in_k))$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ to compute $T_k(\overline{\mathsf{state}}_k, in_k) = (\overline{\mathsf{state}}'_k, out'_k)$.
   (c) Replace $\overline{\mathsf{state}}_k$ by $\overline{\mathsf{state}}'_k$ and out_k by out'_k.
3. Let $out = out_k$ for $k \in [n]$ such that $\overline{\mathsf{state}}_k = \mathsf{done}$.

Figure 13: Protocol Λ in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$-hybrid model UC realizing $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$

Let Adv be an adversary corrupting up to $n-1$ tokens. Let n_c be the number of corrupted tokens, and $n_h = n - n_c$ be the number of honest (uncorrupted) tokens. We construct a UC simulator Sim in Figure 14 internally running Adv such that any environment $\mathcal{E}$ cannot distinguish between interacting with Adv running Λ in the real world and interacting with Sim running $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ in the ideal world. Now consider the series of hybrids:

Hybrid H_0: This hybrid is the real world execution.

Hybrid H_1: This hybrid is similar to H_0 except that every message to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ goes to Sim, and Sim acts honestly on behalf of $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ while recording the messages. This hybrid is identical to H_0.

Let t be the maximum number of call from Adv to execute uncorrupted tokens. Let Hybrid $H_{2.0} = H_1$. For $k = 1, \ldots, t$

Sim

Whenever Sim receives $(\mathsf{create}, \mathsf{sid}, P_i, P_j)$ from $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$, Sim does the followings:

1. For each $k \in [n]$, send $(\mathsf{create}, \mathsf{sid}_k, P_i, P_j)$ to Adv.
2. If Adv replies with $(\mathsf{corrupt}, \mathsf{sid}_k, P_i, P_j)$ for any $k \in [n]$ or P_j is corrupted,
   (a) Follow the protocol of Λ for creating a token, except that Sim uses zero string $0^{|\Pi|}$ instead of the actual token Π to create secret shares $\Pi_1, \ldots, \Pi_n$, and uses $\mathcal{S}_1$ to generate $(\sigma_{i,j,P}, \sigma_{i,j,V}, \tau_{i,j})$ instead of K_{srZK} and Sim_{MPC} to generate σ_i instead of $\mathsf{SetUp}_{\Gamma_0}$ for the setup parameters of Γ for the corruptible tokens. Sim stores the secret shares for later comparison.
   (b) Send T_k to Adv for each k Adv chose to corrupt.
   (c) Store $(\mathsf{replace}, \mathsf{sid}_k, P_i, P_j, T'_k, \mathsf{state}_k)$ from Adv.
   (d) Send $(\mathsf{interrupt}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$.
3. Otherwise, send $(\mathsf{notinterrupt}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$.

Whenever Adv runs the protocol for execution that involves both corrupted and uncorrupted tokens, Sim does the following:

1. Sim generates the Γ messages for uncorrupted T_k using $\mathcal{S}_2(\tau_k)$ and Sim_{MPC} as follows:
   (a) Sim generates and commits to α_k honestly as in Γ.
   (b) Sim runs Sim_{MPC} to generate messages for Γ_0.
   (c) Sim For each message generated by Sim_{MPC}, runs $\mathcal{S}_2(\tau_k)$ to generate messages for the following sim-res ZK argument.
   (d) When Sim_{MPC} queries the functionality of the function F on input $((\Pi'_1, x'_1), \ldots, (\Pi'_n, x'_n))$, if x'_k's are all equal to x', send $(\mathsf{execute}, \mathsf{sid}, P_i, x)$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ and passes the output to Sim_{MPC}. Otherwise, Sim aborts.
2. Sim records and replaces every encrypted state $\overline{\mathsf{state}}$ from uncorrupted T_k with $\overline{\mathsf{state}}^* = \mathsf{Enc}(sk_l, 0^{|\mathsf{state}|})$ before sending it to Adv and replaces $\overline{\mathsf{state}}^*$ with $\overline{\mathsf{state}}$ before applying T_k.
3. Sim records all inputs/outputs to the tokens. If Adv queries with the same input (state and incoming messages), Sim returns the recorded output (new state and outgoing messages).

Figure 14: UC Simulator Sim for Λ

Hybrid $H_{2.k}$**:** This hybrid is similar to $H_{2.(k-1)}$ except that Sim records and replaces the kth encrypted state $\overline{\mathsf{state}}$ from uncorrupted T_l with $\overline{\mathsf{state}}^* = \mathsf{Enc}(sk_l, 0^{|\mathsf{state}|})$ before sending it to Adv and replaces $\overline{\mathsf{state}}^*$ with $\overline{\mathsf{state}}$ before applying T_l. Hybrid $H_{2.(k-1)}$ and $H_{2.k}$ are indistinguishable by the security of the symmetric key encryption $\mathbb{S}$.

Let Hybrid $H_{3.0} = H_{2.t}$. For $k = 1, \ldots, n_h \cdot n_c$,

Hybrid $H_{3.k}$**:** This hybrid is similar to $H_{3.(k-1)}$ except that Sim uses $\mathcal{S}_1$ to generate $(\sigma_{i,j,P}, \sigma_{i,j,V}, \tau_{i,j})$ instead of K_{srZK} for honest token i and corrupted token j with $k = i(n_c - 1) + j$, and runs $\mathcal{S}_2(\tau_{i,j})$ to generate the sim-res ZK messages for token i by feeding the sim-res ZK messages from corrupted tokens. Sim records the transcript leading to each sim-res ZK session. By the GUC-security of the rZK, this hybrid is indistinguishable from H_1.

Lemma 6.1. *Hybrid $H_{3.(k-1)}$ and $H_{3.k}$ are indistinguishable.*

Proof. Suppose there exists a poly-time D that can distinguish $H_3.(k-1)$ and $H_3.k$ with non-negligible probability. We construct a distinguisher D' that can distinguish an interaction of P_{srZK} with a resetting verifier V^*_{srZK} and $\mathcal{S}_2(\tau_{i,j})$ for the sim-res ZK as follows. Given setup strings for the sim-res ZK, D' generates the setup for other pairs of tokens and the inputs for $\Gamma(\Pi)$. D' then runs $H_3.(k-1)$ or $H_3.k$ until Adv queries the honest token to prove a statement x using the sim-res ZK. D' runs the interaction and passes the messages from and to Adv as V^*_{srZK}'s messages. When V^*_{srZK} resets P_{srZK} or $\mathcal{S}_2(\tau_{i,j})$, D' queries the token using the saved state of the earlier round in the sim-res ZK. Finally, D' outputs the output of D. □

Claim. *Fixed a combined determining message $M = M_1 || \ldots || M_n$, any polynomial-time resetting machine Adv can find only one transcript of Γ_0 in $\Gamma(\Pi)$ that every following sim-res ZK argument convinces the verifier to accept.*

Proof. Suppose not. Let $tr = (\ldots, m_{i,k})$ and $tr' = (\ldots, m'_{i,k})$ be the partial transcripts of Γ_0 generated by Adv up to the differing messages $m_{i,k}, m'_{i,k}$ with accepting sim-res ZK argument. Note that we cannot have both $(c_i, M_i, M, tr), (c_i, M_i, M, tr') \in R_{rsZK}$. Otherwise, either M_i or c_i can be decommitted to two different values, and thus can be reduced to the security of the commitment scheme. Hence, either $(c_i, M_i, M, tr) \notin R_{rsZK}$ or $(c_i, M_i, M, tr') \notin R_{rsZK}$. Thus, we can construct a resetting prover P^*_{srZK} that can prove a false statement. □

Let Hybrid $H_{4.0} = H_{3.(n_h \cdot n_c)}$. Let m be the number of distinct sessions of Γ_0 based on combined determining message $M_1 || \ldots || M_n$ generated through Adv querying the tokens. For $k = 1, \ldots, m$,

Hybrid $H_{4.k}$**:** This hybrid is similar to $H_{4.(k-1)}$ except that Sim runs Sim_{MPC} to generate the MPC messages for uncorrupted tokens by feeding the MPC messages from corrupted tokens in the execution of $\Gamma(\Pi)$ following kth combined determining message.

Lemma 6.2. *Hybrid $H_{4.(k-1)}$ and $H_{4.k}$ are indistinguishable.*

Proof. Suppose there exists a poly-time D that can distinguish $H_{4.(k-1)}$ and $H_{4.k}$ with non-negligible probability. We construct a distinguisher D' for Sim_{MPC} as follows. Given the correlated randomness for the MPC, D' generates the rest of the setup parameters for $\Sigma(\Pi)$ as in the experiment. D' then passes the MPC messages from Adv to D followed by the srZK messages from Sim_{ZK}. Since the accepting transcript is unique by the claim above, Adv cannot change the messages. D' outputs the output of D. □

Let Hybrid $H_{5.0} = H_{4.m}$. For $k = 1, \ldots, n$,

Hybrid $H_{5.k}$: This hybrid is similar to $H_{5.(k-1)}$ except that if the kth token is uncorrupted, Sim replaces the PRF in $\Gamma(\Pi)$ with truly random function F.

Lemma 6.3. *Hybrid $H_{5.(k-1)}$ and $H_{5.k}$ are indistinguishable.*

Proof. Suppose there exists a PPT distinguisher D that can distinguish $H_{5.(k-1)}$ and $H_{5.k}$ with non-negligible probability p. We construct a PPT D' that given function f, it runs $H_{5.(k-1)}$ and outputs the output of D when PRF_{s_i} is replaced by f. By the property of the PRF, p is negligible. □

Let Hybrid $H_{6.0} = H_{5.n}$. For $k = 1, \ldots, n$,

Hybrid $H_{6.k}$: This hybrid is similar to $H_{6.(k-1)}$ except that if the kth token is uncorrupted, Sim replaces the second half of the determining message M_i in $\Gamma(\Pi)$ with $\mathsf{com}(0^{|s_i|}; r_i)$ where s_i is the PRF seed in $\Gamma(\Pi)$.

Lemma 6.4. *Hybrid $H_{6.(k-1)}$ and $H_{6.k}$ are indistinguishable.*

Proof. Suppose there exists a PPT distinguisher D that can distinguish $H_{6.(k-1)}$ and $H_{6.k}$ with non-negligible probability p. We construct a PPT D' that given a commitment C of s_i or $0^{|s_i|}$, it runs $H_{6.(k-1)}$ or $H_{6.k}$ and outputs the output of D when the second half of M_i is replaced by C. Since s_i is not used as a witness nor as a PRF seed in $H_{6.(k-1)}$ or $H_{6.k}$, D' can generate the input for D. By the hiding property of com, p is negligible. □

Hybrid H_7: This hybrid is similar to $H_{6.n}$ except that Sim checks if inputs x_k, $k \in [n]$, are the same. If not, Sim records x_k's and replaces outputs of $\Gamma(\Pi)$ by $\perp$.

Lemma 6.5. *Hybrid $H_{6.n}$ and H_7 are indistinguishable.*

Proof. By the binding of com, Adv cannot find x'_k that $\mathsf{com}(x_k; r)$ (where r is an output of the truly random function f) decommitted to x_k except with negligible probability. In this case, by the soundness of rsZK, the output of $\Gamma(\Pi)$ is $\perp$. □

Hybrid H_8: This hybrid is similar to H_7 except that Sim passes token creation request from honest parties to $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ and uses it to compute the output for Sim_{MPC}. By the soundness of the rZK argument, the input to the MPC from Adv are the correctly generated secret share. Thus, this hybrid is identical to H_3 except with negligible probability.

Lemma 6.6. *Hybrid H_7 and H_8 are indistinguishable.*

Proof. Note that if Adv generates messages for the MPC honestly using the same input x_i and the share Π_i given in the setup, then the output from $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ must be the same as the output of the MPC by the correctness of the MPC. Suppose there exists a poly-time D that can distinguish H_3 and H_4 with non-negligible probability p. There must be at least one MPC message m^* from Adv that is not generated honestly. Thus, we construct a resetting prover P^*_{srZK} for the sim-res ZK argument following m^* by randomly choosing a Γ_0 message and passing the following prover messages to V. When Adv sends a different message using the same token state, P^*_{srZK} resets the verifier. It has at least $1/T$ probability of choosing m^* where T is the number of Γ_0 messages sent by Adv. Thus, it has at least p/T probability of proving a false statement, contradicting the resettable soundness of the sim-res ZK. □

Hybrid H_9: This hybrid is similar to H_8 except that Sim generates secret share of zero string $0^{|\Pi|}$ instead of the one received from an honest party. By the security of the secret sharing scheme, this hybrid is indistinguishable from H_4.

Lemma 6.7. *Hybrid H_8 and H_9 are indistinguishable.*

Proof. Suppose there exists a poly-time D that can distinguish H_8 and H_9 with non-negligible probability. We construct a distinguisher D' for the secret sharing scheme $\mathcal{S}$ as follows. D' runs the experiment for D until it is given Adv shares consisting of less than n shares. D' then continues the experiment and D distinguishes between H_8 and H_9. Using the result of D, D' can distinguish between less than n shares of 0 and some program Π, contradicting the security of $\mathcal{S}$. □

Let Hybrid $H_{10.0} = H_9$. For $k = 1, \ldots, n$,

Hybrid $H_{10.k}$: This hybrid is similar to $H_{10.(k-1)}$ except that if the kth token is uncorrupted, Sim replaces the second half of the determining message M_i in $\Gamma(\Pi)$ with $\mathsf{com}(s_i; r_i)$ where s_i is the PRF seed in R_i. Hybrid $H_{10.(k-1)}$ and $H_{10.k}$ are indistinguishable by similar argument as Lemma 6.4.

Let Hybrid $H_{11.0} = H_{10.n}$. For $k = 1, \ldots, n$,

Hybrid $H_{11.k}$: This hybrid is similar to $H_{11.(k-1)}$ except that if the kth token is uncorrupted, Sim replaces the truly random function F in $\Gamma(\Pi)$ with PRF_{s_i}. Hybrid $H_{10.(k-1)}$ and $H_{10.k}$ are indistinguishable by similar argument as Lemma 6.3. This hybrid is the ideal world execution.

Using these, we prove our main theorem below.

Theorem 6.8. *Assuming an existence of OWFs, there exists a protocol with n corruptible tokens in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$-hybrid model UC-realizing $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$.*

7 RAM Obfuscation and Tokens with Bounded Memory

RAM obfuscation. We now describe how to obtain program obfuscation with stateless hardware tokens solely from OWFs. This improves the assumption from the work of Nayak *et al.* [37], who additionally also assumed collision-resistant hash functions. At a very high level, the protocol of Nayak *et al.* makes use of OWFs and CRHFs. First, they make use of CRHFs for the authenticated ORAM structure. We observe that we can replace the authenticated ORAM used in Nayak *et al.* with an authenticated ORAM based on OWF (that can be built from the work of Ostrovsky and Goldreich, Ostrovsky). Next, in order to obtain a single starting seed for randomness that depends on the specific execution of the program and input, Nayak *et al.* require the user to first feed in a hash of the input to the token and then use this hash to derive all randomness (along with a unique program id). This gives them a unique execution id. We derive a unique value based on the input and program by having the user feed the input one-by-one to the token. Upon receiving one input, the token will authenticate it and provide an authentication tag (this process is deterministic) that will then allow the user to input the next input. This process continues until the last input is inserted into the token, upon which the authentication tag produced at this stage is a unique id that can be used (in combination with the program id) to derive all randomness needed by the token for program execution. This process is similar in spirit to the GGM construction of deriving a PRF from a PRG.

7.1 High level description of the protocol

Program Authentication. At a high level, the program creation by the sender works as follows. Let the program to be obfuscated be $\mathsf{RAM} := (\mathsf{cpustate}, \mathsf{mem})$ where mem is a list of program instructions and $\mathsf{cpustate}$ is the initial cpu state. Let the program comprise of t instructions. The sender first creates the token containing a hardwired secret key K where $K := (K_e, K_{\text{prf}})$. K_e is used as the encryption key for encrypting state, K_{prf} is used as the key to a pseudorandom function used by the token to generate all randomness needed for executing the ORAM, creating ciphertexts, and so on. The sender creates a unique execution identity $\mathsf{id}_{\text{exec}}$, which is unique for every program created. The sender then encrypts $\mathsf{mem}||\mathsf{id}_{\text{exec}}||\mathsf{id}_{\text{instr}}$ (one instruction at a time) to obtain $\overline{\mathsf{mem}}$ ($\overline{\mathsf{mem}}_i$ denotes the ciphertext obtained upon encrypting $\mathsf{mem}_i||\mathsf{id}_{\text{exec}}||i$). The sender also computes a "tag" of the start ciphertext, $\overline{\mathsf{mem}}_1$, as $\tau_1 = \mathsf{PRF}_{K_{\text{prf}}}(\mathtt{start}, \overline{\mathsf{mem}}_1)$. The sender creates an encrypted program header $\overline{\mathsf{Header}} := \mathsf{Enc}_{K_e}(\mathsf{cpustate}, \mathsf{id}_{\text{exec}}, t)$. The receiver is sent $\overline{\mathsf{mem}}, \overline{\mathsf{mem}}_1^*$ and $\overline{\mathsf{Header}}$ as the obfuscated program.

Program Feed. At a very high level, the receiver will feed in the program, one instruction at a time, to the token, as follows:

1. As the first message, the token receives $(\mathtt{programauth}, 1, \overline{\mathsf{mem}}_1, \tau_1, \overline{\mathsf{mem}}_2, \overline{\mathsf{Header}})$. It will check that $\mathsf{PRF}_{K_{\text{prf}}}(\mathtt{start}, \overline{\mathsf{mem}}_1, \overline{\mathsf{Header}}) = \tau_1$ and output $\perp$ otherwise. Similarly, for all $2 \leq i < t-1$, it receives $(\mathtt{programauth}, i, \tau_{i-1}, \overline{\mathsf{mem}}_i, \tau_i, \overline{\mathsf{mem}}_{i+1}, \overline{\mathsf{Header}})$. It will check that $\mathsf{PRF}_{K_{\text{prf}}}(\tau_{i-1}, \overline{\mathsf{mem}}_i, \overline{\mathsf{Header}}) = \tau_i$ and output $\perp$ otherwise.

2. Next, it decrypts $\overline{\mathsf{mem}}_i$ and $\overline{\mathsf{mem}}_{i+1}$ to get mem_i and mem_{i+1}, and $\mathsf{id}_{\text{exec}}$ as well as decrypt $\overline{\mathsf{Header}}$ to get $\mathsf{id}_{\text{exec}}$ and t. It will check that $\mathsf{id}_{\text{instr}} = i$ and $i+1$ respectively (also that these values are $\leq t$) and that the two $\mathsf{id}_{\text{exec}}$ values are the same and equal to the $\mathsf{id}_{\text{exec}}$ value in $\overline{\mathsf{Header}}$. If these checks do not pass, it will respond with $\perp$. If the checks pass, the token will output $\tau_{i+1} = \mathsf{PRF}_{K_{\text{prf}}}(\tau_i, \overline{\mathsf{mem}}_{i+1}, \overline{\mathsf{Header}})$.

Input Feed. Let the input to the program be denoted by $x_1, \cdots, x_n$. The receiver will send the following instructions, step-by-step, for every input, to the token.

1. On input, $(\mathtt{inputauth}, 1, \tau_{t-1}, \overline{\mathsf{mem}}_t, \tau_t, x_1, n, \overline{\mathsf{Header}})$, it checks that $\mathsf{PRF}_{K_{\text{prf}}}(\tau_{t-1}, \overline{\mathsf{mem}}_t, \overline{\mathsf{Header}}) = \tau_t$, that $\overline{\mathsf{mem}}_t$ is the t^{th} program instruction (by decrypting $\overline{\mathsf{mem}}_t$ to get $\mathsf{id}_{\text{instr}}$ and $\overline{\mathsf{Header}}$ to get t and comparing) and output $\perp$ otherwise. It outputs $\tau_{t+1} = \mathsf{PRF}_{K_{\text{prf}}}(\tau_t, 1, x_1, n, \overline{\mathsf{Header}})$.

2. On input, $(\mathtt{inputauth}, j, \tau_{t+j-2}, x_{j-1}, \tau_{t+j-1}, x_j, n, \overline{\mathsf{Header}})$, for $2 \leq j \leq n$, it checks that $\mathsf{PRF}_{K_{\text{prf}}}(\tau_{t+j-2}, j-1, x_{j-1}, n, \overline{\mathsf{Header}}) = \tau_{t+j-1}$ and outputs $\perp$ otherwise. It then outputs $\tau_{t+j} = \mathsf{PRF}_{K_{\text{prf}}}(\tau_{t+j-1}, j, x_j, n, \overline{\mathsf{Header}})$.

Program/Input ORAM Insertion. Once the program and input authentication is done, the program and input, henceforth collectively referred to as memory, must be inserted into the Authenticated Oblivious RAM structure. There are t program instructions and n inputs that must be inserted. Let $\ell = t + n$ be the total memory requirement of the program (we can assume this without loss of generality as any additional memory needed by the program can be thought of as

dummy program instructions). First, a set of ℓ "zeroes" are inserted into the ORAM structure (i.e., the values of memory in all locations is set to 0)[6]. The insertion of a set of ℓ "zeroes" into the ORAM structure is done as follows:

1. For every memory location $1 \leq i \leq \ell$, the user prepares the message $(\texttt{ORAMsetup}, i, \ell, \tau_{i-1}^{\text{oraminit}}, n, x_n, \overline{\mathsf{Header}}, \tau_i^{\text{oraminit}}, \tau_\ell)$ and gives it to the token, with $\tau_1^{\text{oraminit}} = \tau_\ell$ and $\tau_0^{\text{oraminit}} = \tau_{\ell-1}$. The token checks that $\mathsf{PRF}_{K_{\text{prf}}}(\tau_0^{\text{oraminit}}, n, x_n, n, \overline{\mathsf{Header}}) = \tau_1^{\text{oraminit}}$ (for $i = 1$) and $\mathsf{PRF}_{K_{\text{prf}}}(\text{ORAMsetup}, i, \ell, \overline{\mathsf{Header}}, \tau_{i-1}^{\text{oraminit}}, \tau_\ell) = \tau_i^{\text{oraminit}}$ (for all other i) and outputs $\perp$ otherwise.

2. Otherwise, the token derives a key for the ORAM structure − this ORAM key is derived as $K_{\text{oram}} = \mathsf{PRF}_{K_{\text{prf}}}(\texttt{ORAMKey}, \tau_\ell)$.

   (a) It creates an ORAM initialization structure (that is, creates an initial random mapping of all virtual addresses to their real address); this initialization is done using randomness from the ORAM key K_{oram}.

   (b) In this map let address a_j have been mapped to address i. In this case, the token creates an authenticated encryption of $(a_j, 0)$ (again using keys and randomness derived from K_{oram}) to be inserted into the ORAM structure at virtual address i.

   (c) The token then outputs $\tau_{i+1}^{\text{oraminit}} = \mathsf{PRF}_{K_{\text{prf}}}(\text{ORAMsetup}, i, \ell, \overline{\mathsf{Header}}, \tau_i^{\text{oraminit}}, \tau_\ell)$.

Once all ℓ memory locations have been inserted with 0 values, the user then inserts the real input and program into the ORAM structure. This is done as follows: in the reverse order, starting with the n^{th} input to the first input, and then the t^{th} to the first program instruction. We now describe this process at a high level. For ease of exposition, we shall assume that every ORAM operation is a single step denoted as $\mathsf{oram}_{\sigma,K_{\text{oram}}}(i, v_i, \texttt{read/write}, \perp/v_i^*)$ (this can be easily extended to the case when the ORAM read/write is a set of operations, similar to Nayak *et al.* [37]). The protocol is as follows:

1. The user will insert the i^{th} memory location ($\ell \geq i \geq 1$, which is either an input or a program instruction) by sending the message $(\text{MemORAMInsert}, i, \ell, \tau_{2\ell-i}^{\text{oraminit}}, n, x_n, n, \overline{\mathsf{Header}}, \tau_{2\ell-i+1}^{\text{oraminit}}, \tau_i, \tau_{i-1}, w_i)$, where $w_i = (i - t, x_{i-t}, n)$ if the i^{th} location has an input (i.e., $\ell \geq i \geq t + 1$) and $w_i = \overline{\mathsf{mem}}_i$ if the i^{th} location has a program instruction (i.e., $t \geq i \geq 1$).

2. If the i^{th} location has an input:

   (a) The token will check that $\tau_i = \mathsf{PRF}_{K_{\text{prf}}}(\tau_{i-1}, i - t, x_{i-t}, n, \overline{\mathsf{Header}})$ and that $\tau_{2\ell-i+1}^{\text{oraminit}} = \mathsf{PRF}_{K_{\text{prf}}}(\text{ORAMsetup}, i, \ell, \overline{\mathsf{Header}}, \tau_{2\ell-i}^{\text{oraminit}}, \tau_\ell)$.

   (b) The token will then execute the ORAM instruction $\mathsf{oram}_{\sigma,K_{\text{oram}}}(i, 0, \texttt{write}, x_{i-t})$.

   (c) The token then outputs $\tau_{2\ell-i+2}^{\text{oraminit}} = \mathsf{PRF}_{K_{\text{prf}}}(\text{ORAMsetup}, i - 1, \ell, \overline{\mathsf{Header}}, \tau_{2\ell-i+1}^{\text{oraminit}}, \tau_\ell)$.

3. If the i^{th} location has a program instruction:

[6]Whenever, the state of the program (ORAM or otherwise) needs to be modified, this is done by appending encrypted `state` with $\overline{\mathsf{Header}}$ and then authenticating, similar to Nayak *et al.* [37]

(a) The token will check that $\tau_i = \mathsf{PRF}_{K_{\mathrm{prf}}}(\tau_{i-1}, \overline{\mathsf{mem}}_i, \overline{\mathsf{Header}})$ and that $\tau_{2\ell-i+1}^{\mathrm{oraminit}} = \mathsf{PRF}_{K_{\mathrm{prf}}}(\mathrm{ORAMsetup}, i, \ell, \overline{\mathsf{Header}}, \tau_{2\ell-i}^{\mathrm{oraminit}}, \tau_\ell)$.

(b) The token decrypts $\overline{\mathsf{mem}}_i$ to get $\mathsf{mem}_i||\mathsf{id}_{\mathrm{exec}}||i$ and executes the ORAM instruction $\mathsf{oram}_{\sigma,K_{\mathrm{oram}}}(i, 0, \mathtt{write}, \mathsf{mem}_i||\mathsf{id}_{\mathrm{exec}}||i)$.

(c) The token then outputs $\tau_{2\ell-i+2}^{\mathrm{oraminit}} = \mathsf{PRF}_{K_{\mathrm{prf}}}(\mathrm{ORAMsetup}, i-1, \ell, \overline{\mathsf{Header}}, \tau_{2\ell-i+1}^{\mathrm{oraminit}}, \tau_\ell)$.

Program Execution. The program execution works in a similar manner to that of Nayak *et al.*

8 Tokens with small memory

We consider a variant of $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ in Figure 9, called $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short},L_1,L_2}$ in Figure 15 where create and execute only take Π and *inp* of short size. We also allow a token sender to send a message along with the token created through the functionality. This allows the adversary to intercept the message when it chooses to corrupt a token without neither sender nor receiver knowledge. This is unavoidable as we represent a token in the standard corruptible model with both a token and an additional auxiliary string from the sender. We use $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$ when L_1 and L_2 are clear from the context.

In theory, we would like L_1 and L_2 be of constant size in security parameter. Though [37] suggests using logarithmic size in practice for better performance.

We define an implementation Token of $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$-hybrid model in Figure 16.

Theorem 8.1. *The protocol* Token *UC-realizes* $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ *in* $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$*-hybrid model.*

Proof. (Sketch) The proof follows closely to the proof of Theorem 1 in [37] when the adversary Adv does not corrupt a token. We construct a simulator Sim as follows: Whenever Sim receives a create message from $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$, it passes the message to Adv. If Adv chooses to corrupt the token, Sim sends corrupt to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ and receives Π. It creates RAM from Π and generates $m = (\mathsf{mem}, \mathsf{header}, \tau_1)$ honestly as in Token and sends them to Adv. When Sim receives $\overline{T}_K^*$ and m^* from Adv, it parses $K^* = (K_e^*, K_{prf}^*)$ and $m^* = (\mathsf{mem}^*, \mathsf{header}^*, \tau_1^*)$. Sim then uses K_e^* to decrypt to get $\mathsf{RAM}^* = (\mathsf{cpustate}^*, \mathsf{mem}^*)$ representing a program Π^*. If Sim fails to get Π^* from this steps, it aborts. Otherwise, it sends Π^* to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$. Sim runs Token honestly for this token, and sends execution message to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ whenever Adv complete the execution phase.

If Adv chooses not to corrupt the token created by an honest party, Sim passes notcorrupt to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$, and creates a token for $\mathsf{RAM}_0 = (0^{|\mathsf{cpustate}|}, 0^{|\mathsf{mem}_0|})$ instead of the actual program RAM representing Π. Sim maintains a memory mem' to keep track of $\mathsf{mem} = \overline{\mathsf{mem}}_0||inp||0$.

Sim runs input and program authentication phases as described in Token, and also records the input authenticated. If Adv resets and sends different input, Sim aborts.

Whenever $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$ sends an encrypted state $\overline{\mathsf{state}}$ to Adv, Sim instead sends $\mathsf{Enc}_K(0^{|\mathsf{state}|})$.

To simulate ORAM algorithm, it uses the ORAM simulator and the PRF is replaced by a truly random function. Sim records all transcripts with Adv and sends the recorded response when Adv resets and sends the recorded messages. It aborts if it receives messages different from recorded ones.

Upon completion of the execution phase, Sim sends the execution message to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ with the recorded input. Sim sends the output it receives from $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible}}$ to Adv.

We show the indistinguishability through a series of hybrids:

$\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible},\mathsf{short},L_1,L_2}$

Upon receiving $(\mathsf{create}, \mathsf{sid}, P_j, \Pi, m)$ from P_i with $i \neq j$:

1. If $|\Pi| > L_1$, ignore.
2. Otherwise, if there is no tuple of the form $(\mathsf{sid}, P_i, P_j, \star, \star)$, store $(\mathsf{sid}, P_i, P_j, \Pi, m, \mathsf{creating})$.
3. Send $(\mathsf{create}, \mathsf{sid}, P_i, P_j)$ to Adv.

Upon receiving $(\mathsf{corrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find the unique stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, m, \mathsf{creating})$. If no such tuple exists, abort.
2. Send (Π, m) to Adv.

Upon receiving $(\mathsf{replace}, \mathsf{sid}, P_i, P_j, \Pi^*, m^*, \mathsf{state}_0)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, m, \mathsf{creating})$. If no such tuple exists, abort.
2. Store $(\mathsf{sid}, P_i, P_j, \Pi^*, \mathsf{state}_0)$, send $(\mathsf{done}, \mathsf{sid})$ to P_i and send $(\mathsf{create}, \mathsf{sid}, P_i, m^*)$ to P_j.

Upon receiving $(\mathsf{notcorrupt}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find and remove the unique stored tuple $(\mathsf{sid}, P_j, P_i, \Pi, m, \mathsf{creating})$. If no such tuple exists, abort.
2. Store $(\mathsf{sid}, P_i, P_j, \Pi, \bot)$, send $(\mathsf{done}, \mathsf{sid})$ to P_i and send $(\mathsf{create}, \mathsf{sid}, P_i, m)$ to P_j.

Upon receiving $(\mathsf{execute}, \mathsf{sid}, P_i, inp)$ from P_j with $i \neq j$:

1. If $|inp| > L_2$, ignore.
2. Otherwise, find the unique stored tuple $(\mathsf{sid}, P_i, P_j, \Pi, \mathsf{state})$. If no such tuple exists, abort.
3. Run $\Pi(\mathsf{state}, inp)$ and let (state', out) be the output. If $\mathsf{state} \neq \bot$, set $\mathsf{state} = \mathsf{state}'$.
4. Send (sid, out) to P_j.

Upon receiving $(\mathsf{read}, \mathsf{sid}, P_i, P_j)$ from Adv:

1. Find the unique stored tuple $(\mathsf{sid}, P_j, P_i, P, \mathsf{state})$. If no such tuple exists, abort.
2. Send $(\mathsf{sid}, \mathsf{state})$ to Adv.

Figure 15: Token with Short Input Functionality $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible},\mathsf{short},L_1,L_2}$

Token

To create a token running Π for P_j, P_i does the following:

1. Construct a RAM program $\mathsf{RAM} = (\mathsf{cpustate}, \mathsf{mem}_0)$ that runs Π.
2. Sample $K_e \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $K_{prf} \leftarrow U_\ell$.
3. Let $\tau_1 = \mathsf{PRF}_{K_{prf}}(\mathsf{start}, \overline{\mathsf{mem}}_1)$ where $\overline{\mathsf{mem}}_k = \mathsf{Enc}_{K_e}(\mathsf{mem}_0[k]||\mathsf{id}||k)$, and $\overline{\mathsf{header}} = \mathsf{Enc}_{K_e}(\mathsf{cpustate}||\mathsf{id}||t)$.
4. Send $(\mathsf{create}, \mathsf{sid}, P_j, \overline{T}_K, (\overline{\mathsf{mem}}_0, \overline{\mathsf{header}}, \tau_1))$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$ where $K = (K_e, K_{prf})$ and $\overline{\mathsf{mem}}_0 = \{\overline{\mathsf{mem}}_k\}_{k\in[t]}$.

To execute a token on input $inp = \{inp_k\}_{k\in[n]}$, P_j, given $(\overline{\mathsf{mem}}_0, \overline{\mathsf{header}}, \tau_1)$, does the following:

1. Authenticate the program feed and input feed as described in Section 7 to get $\tau_2, \ldots, \tau_{t+n}$ by sending $(\mathsf{execute}, \mathsf{sid}, P_i, -)$ to $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$ with appropriate command.
2. Execute as described in Section 7.

Figure 16: Token Implementation Token in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$-hybrid model

Hybrid H_0: This hybrid is the real world execution. Sim runs Token honestly in place of honest parties given their inputs using $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible,short}}$ functionality.

Hybrid H_1: This hybrid is similar to Hybrid H_0 except that when Adv chooses to corrupt a token created by an honest party, Sim maintains a memory mem' to keep track of mem. Sim records all transcripts with Adv and sends the recorded response when Adv resets and sends the recorded messages. This hybrid is identical to H_1.

Hybrid H_2: This hybrid is similar to Hybrid H_1 except that when Adv chooses to corrupt a token created by an honest party, Sim uses a truly random function instead of the PRF. This hybrid is computationally indistinguishable from H_1 by the property of the PRF.

Hybrid H_3: This hybrid is similar to Hybrid H_2 except that when Adv chooses to corrupt a token created by an honest party, Sim checks the authenticity of the program in $\overline{\mathsf{header}}$ and input in the execution phase directly from its recorded memory mem' instead of checking tags $\tau_1, \ldots, \tau_{t+n}$. When Adv initializes the program and the input, Sim records the pairing of tags and the program and the input. If Adv restarts the program and input feed, Sim compares with the recorded pairs and outputs the same tags for the same program and input. Since the tags are generated uniformly at random, the probability that poly-time Adv can find a different program or input with the same tag is negligible. This hybrid is indistinguishable from H_2 by the randomness of the truly random function.

Hybrid H_4: This hybrid is similar to Hybrid H_3 except that when Adv chooses to corrupt a token created by an honest party, *simu* compares state from Adv with the recorded state previously outputed on behalf of $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible},\mathsf{short}}$ instead of using the authentication of Dec. This hybrid is computationally indistinguishable from H_3 by the authenticity of the encryption scheme.

Hybrid H_5: This hybrid is similar to Hybrid H_4 except that when Adv chooses to corrupt a token created by an honest party, Sim replaces all encryptions from $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible},\mathsf{short}}$ to Adv with encryption of zero strings, This hybrid is computationally indistinguishable from H_4 by the security of the encryption scheme.

Hybrid H_6: This hybrid is similar to Hybrid H_5 except that when Adv chooses to corrupt a token created by an honest party, Sim uses the ORAM simulator instead of the actual ORAM algorithm. This hybrid is statistically indistinguishable from H_6 by the property of ORAM simulator. This is the ideal world interaction between Sim and Adv. □ □

Combining the above result with the result in Section 6 gives the following corollary.

Corollary 8.2. *Assuming OWFs, there exists a protocol that UC realizes $\mathcal{F}_{\mathsf{token}}^{\mathsf{abort}}$ functionality in $\mathcal{F}_{\mathsf{token}}^{\mathsf{corruptible},\mathsf{short}}$-hybrid model using n corruptible tokens with short inputs and small size against an adversary corrupting up to $n-1$ tokens.*

References

[1] S. Agrawal, V. Goyal, A. Jain, M. Prabhakaran, and A. Sahai. New impossibility results for concurrent composition and a non-interactive completeness theorem for secure computation. In *CRYPTO 2012* , pages 443–460, 2012.

[2] G. Ateniese, A. Kiayias, B. Magri, Y. Tselekounis, and D. Venturi. Secure outsourcing of circuit manufacturing. In *ProvSec* 2018.

[3] B. Barak, O. Goldreich, S. Goldwasser, and Y. Lindell. Resettably-sound zero-knowledge and its applications. In *FOCS '02*, pages 116–125, 2001.

[4] B. Barak, M. Prabhakaran, and A. Sahai. Concurrent non-malleable zero knowledge. In *FOCS '06*, pages 345–354, 2006.

[5] D. Beaver. Precomputing oblivious transfer. *CRYPTO '95*, pages 97–109, 1995.

[6] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC '96*, pages 479–488. ACM, 1996.

[7] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01*, pages 136–145. IEEE, 2001.

[8] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC '07*, pages 61–85. Springer, 2007.

[9] R. Canetti and M. Fischlin. Universally composable commitments. In *Crypto '01*, pages 19–40. Springer, 2001.

[10] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali. Resettable zero-knowledge (extended abstract). In *STOC '00*, pages 235–244, 2000.

[11] R. Canetti, E. Kushilevitz, and Y. Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *Journal of Cryptology*, 19(2):135–167, 2006.

[12] N. Chandran, V. Goyal, and A. Sahai. New constructions for uc secure computation using tamper-proof hardware. In *Eurocrypt '08*, pages 545–562. 2008.

[13] S. G. Choi, J. Katz, D. Schröder, A. Yerukhimovich, and H.-S. Zhou. (efficient) universally composable oblivious transfer using a minimal number of stateless tokens. In *TCC '14*, pages 638–662. Springer, 2014.

[14] K.-M. Chung, R. Ostrovsky, R. Pass, and I. Visconti. Simultaneous resettability from one-way functions. In *FOCS '13*, pages 60–69. IEEE, 2013.

[15] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Venkitasubramaniam. Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In *ASIACRYPT '13*, pages 316–336. Springer, 2013.

[16] I. Damgård and Y. Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO '05*, pages 378–394. Springer, 2005.

[17] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust non-interactive zero knowledge. In *CRYPTO '01*, pages 566–598. Springer, 2001.

[18] N. Döttling, D. Kraschewski, J. Müller-Quade, and T. Nilges. General statistically secure computation with bounded-resettable hardware tokens. In *TCC (1)*, pages 319–344, 2015.

[19] S. Dziembowski, S. Faust, and F.-X. Standaert. Private circuits iii: Hardware trojan-resilience via testing amplification. In *CCS '16*, pages 142–153. ACM, 2016.

[20] U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In *STOC '90*, pages 416–426, 1990.

[21] M. Fischlin, B. Pinkas, A. Sadeghi, T. Schneider, and I. Visconti. Secure set intersection with untrusted hardware tokens. In *Topics in Cryptology - CT-RSA 2011*, volume 6558 of *LNCS*, pages 1–16. Springer, 2011.

[22] J. A. Garay, P. MacKenzie, and K. Yang. Strengthening zero-knowledge protocols using signatures. In *Eurocrypt '03*, volume 2656, pages 177–194. Springer, 2003.

[23] S. Garg, A. Kumarasubramanian, R. Ostrovsky, and I. Visconti. Impossibility results for static input secure computation. In *CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 424–442. Springer, 2012.

[24] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions In *Journal of the ACM*, 33(4):792-807, 1986.

[25] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC '87*, pages 218–229, 1987.

[26] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[27] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC '10*, pages 308–326. 2010.

[28] J. Groth and R. Ostrovsky. Cryptography in the multi-string model. In *CRYPTO '07*, pages 323–341, 2007. Springer-Verlag.

[29] C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In *TCC '16*, pages 367–399. Springer, 2016.

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

[31] Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC'13*, pages 600–620. Springer, 2013.

[32] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer–efficiently. *CRYPTO '08*, 5157:572–592, 2008.

[33] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT '07*, pages 115–128. Springer, 2007.

[34] H. Lin, R. Pass, and M. Venkitasubramaniam. A unified framework for concurrent security: universal composability from stand-alone non-malleability. In *STOC '09*, pages 179–188. ACM, 2009.

[35] P. MacKenzie and K. Yang. On simulation-sound trapdoor commitments. In *EUROCRYPT '04*, pages 382–400. Springer, 2004.

[36] V. Mavroudis, A. Cerulli, P. Svenda, D. Cvrcek, D. Klinec and G. Danezis. A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components. In *CCS '17*, pages 1583–1600. ACM, 2017.

[37] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal. Hop: Hardware makes obfuscation practical. In *NDSS '17*, 2017.