# UC-Secure Multiparty Computation from One-Way Functions using Stateless Tokens

Saikrishna Badrinarayanan[1⋆], Abhishek Jain[2 ⋆⋆], Rafail Ostrovsky[1 ⋆⋆⋆], and Ivan Visconti[3†]

[1] UCLA
{saikrishna,rafail}@cs.ucla.edu
[2] JHU
abhishek@cs.jhu.edu
[3] University of Salerno
visconti@unisa.it

**Abstract.** We revisit the problem of universally composable (UC) secure multiparty computation in the stateless hardware token model.

- We construct a three round multi-party computation protocol for general functions based on one-way functions where each party sends two tokens to every other party. Relaxing to the two-party case, we also construct a two round protocol based on one-way functions where each party sends a single token to the other party, and at the end of the protocol, both parties learn the output.
- One of the key components in the above constructions is a new two-round oblivious transfer protocol based on one-way functions using only one token, which can be reused an unbounded polynomial number of times.

All prior constructions required either stronger complexity assumptions, or larger number of rounds, or a larger number of tokens.

**Keywords:** Secure Computation, Hardware Tokens.

## 1 Introduction

**Hardware Token Model.** The seminal work of Katz [Kat07] initiated the study of Universally Composable (UC) [Can01] protocols using tamper-proof

⋆ Research supported in part by the IBM PhD Fellowship.
⋆⋆ Research supported in part by NSF SaTC grant 1814919 and Darpa Safeware grant W911NF-15-C-0213.
⋆⋆⋆ Research supported in part by NSF-BSF Grant 1619348, DARPA/SPAWAR N66001-15-C-4065, ODNI/IARPA 2019-1902070008 US-Israel BSF grant 2012366, JP Morgan Faculty Award, 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.
† Research supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No 780477 (project PRIViLEDGE).

hardware tokens. In this model, each party can create hardware tokens that compute functions of its choice such that an adversary that has access to these tokens does not learn anything more than their input/output behavior. The main appeal of this model is that its security relies on a physical assumption and does not require all the players to trust a common entity. Instead each player can construct its own tokens or rely on its own token manufacturer.

Over the years, two different versions of the hardware token model have been studied: stateful tokens, and stateless (a.k.a. resettable) tokens. The latter model is more realistic, and, in practice, places weaker requirements on the token manufacturer. This makes it appealing from both theoretical and practical viewpoints. In this work, we focus on the *stateless* hardware token model.

**Minimizing Complexity.** There are three main parameters in the study of UC secure multiparty computation (MPC) in the stateless hardware token model: complexity assumption, number of rounds in the protocol, and the number of tokens. Since the introduction of the stateless hardware token model [CGS08], several works [CGS08,GIS+10,CKS+14,DKMN15a,HPV16] have investigated various trade-offs between these three parameters (see Section 1.2 for details). However, in the *multiparty* setting, the best known protocols based on the minimal assumption of one-way functions[4] require $O(d)$ rounds [HPV16], where $d$ is the depth of the circuit being computed. This leaves open the following question (w.r.t. any polynomial number of tokens):

*Does there exist a constant round UC secure multiparty computation protocol for general functions based on one-way functions in the stateless hardware token model?*

*Token Reusability.* Since tamper-proof hardware tokens can be expensive to manufacture, it is very desirable to allow *reuse* of tokens across multiple sessions. Indeed, for this reason, the reusable token model was put forth by [CKS+14], where a set of tokens can be reused across multiple protocol executions (for different function evaluations, on different set of inputs) between the same set of parties, a.k.a. concurrent *self composition*. While the ability to reuse a setup for concurrent self composition typically comes for free in setup models such as the common reference string model, it is not the case for the hardware token model. As such, it was put forth as an explicit goal by [CKS+14].

In the setting of *two-party* computation, [HPV16] constructed round-optimal (i.e., two-round) protocols based on one-way functions, with unlimited token reusability (even in the stronger Global UC model [CDPW07,CJS14]). However, their protocol requires a polynomial number of tokens. The concurrent work of [DKMN15a] requires only one token, but does not support unlimited token reuse. This leaves open the following question:

[4] One-way function is a necessary assumption in the stateless hardware token model since an unbounded adversary can simply "learn" a stateless token [GIS+10].

*Does there exist a two round UC secure two-party computation protocol for general functions based on one-way functions in the reusable stateless token model?*

In this work, we resolve both of the aforementioned questions in the affirmative.

### 1.1 Our Results

We continue the study of UC secure computation in the stateless hardware token model.

**Multiparty Computation.** Our first result is a three-round UC-secure multiparty computation protocol based on the minimal assumption of one-way functions. Our protocol requires each party to send two tokens to every other party.

**Theorem 1 (Informal).** *Assuming one-way functions, there exists a three-round UC multiparty computation protocol in the stateless hardware token model.*

If we restrict our attention to the case of two parties, where the parties communicate over simultaneous broadcast channels[5], we can further reduce the round-complexity of our protocol to *two-rounds*. We emphasize that at the end of the protocol, *both* parties learn the output. Our protocol requires each party to send only one token to the other party. Prior to our work, no such two-party computation protocol was known in the literature.

**Theorem 2 (Informal).** *Assuming one-way functions, there exists a two-round UC two-party computation protocol over simultaneous broadcast channels, in the stateless hardware token model.*

We emphasize that the protocols in Theorem 1 and 2 allow for unlimited token reuse across multiple sessions between the same set of parties.

**Oblivious Transfer.** A key component in our constructions is a new two-round UC oblivious transfer protocol in the stateless hardware token model based on one-way functions, and relying upon a single token. Crucially, unlike [DKMN15a] who support an a priori bounded number of uses of the token, our protocol supports *unlimited* token reuse.

**Theorem 3 (Informal).** *Assuming one-way functions, there exists a two-round UC oblivious transfer protocol in the reusable stateless hardware token model, using a single token.*

[5] This is the standard model for multiparty computation, where in each round, every party simultaneously broadcasts a message to the other parties. However, a rushing adversary may wait to receive the honest party's message in any round before deciding its own message.

By combining the above theorem with the work of Ishai et al. [IKO+11], we can obtain a two-round secure two-party computation protocol in the unidirectional-message model based on one-way functions with one reusable token. Unlike Theorem 2, however, only one of the two parties learns the output at the end of the protocol.

**Discussion and Future Work.** The work of Hazay et al. [HPV16] puts forth GUC security as a more desirable notion of security for protocols in the hardware token model. Our protocols do not achieve GUC security, and it is an interesting open problem to extend our results to the stronger model of [HPV16].

Further, unlike the work of [HPV16], who construct black-box protocols, our protocols make non-black-box use of one-way functions due to the use of ZK arguments. It is an interesting open problem to construct optimal black-box protocols in the hardware token model.

### 1.2 Related Work

Katz established the first feasibility results for UC secure multiparty computation (MPC) using stateful hardware tokens. Subsequently, this model was extensively explored in several directions with the purpose of improving upon the complexity assumptions, round-complexity of protocols and the number of required tokens [MS08,GKR08,Kol10,DKM11,DKM12].

The study of UC-secure protocols in the stateless hardware token model was initiated by Chandran et al. [CGS08]. They constructed a polynomial round protocol for multi-commitment functionality where each party exchanges one token with the other party, based on enhanced trapdoor permutations. Subsequent to their work, Goyal et al. [GIS+10] constructed constant-round protocols assuming collision-resistant hash functions (CRHFs). However, these improvements were achieved at the cost of requiring a polynomial number of tokens. Choi et al. [CKS+14] subsequently improved upon their result by decreasing the number of required tokens to only one, while still using only constant rounds and CRHFs.

Recently, Hazay et al. [HPV16] constructed two rounds two-party computation protocols based on one-way functions, and three-round MPC protocols based on oblivious transfer in the Global UC model. They also construct a multiparty protocol from one-way functions where the round complexity is linear in the depth of the circuit being computed. All of their protocols require a polynomial number of tokens. In a concurrent work, Döttling et al. [DKMN15a] construct two-round oblivious transfer from stateless tokens based on one-way functions, but their protocol does not support unbounded token reuse. Badrinarayanan et al. [BJOV18] constructed a non-interactive UC-secure two party computation protocol in the stateless hardware token model based on one way functions.

Döttling et al. [DKMN15b] construct information-theoretic UC-secure protocols in a model where the tokens can be reset only a bounded number of times. In a different work, Döttling et al. [DMMN13] construct UC-secure protocols for resettable functionalities using stateless tokens. In contrast, we focus on securely computing general functionalities using stateless tokens in this work.

The work of Agrawal et al. [AAG+14] proves lower bounds on the number of token queries necessary for secure computation in the stateless hardware token model. We do not seek to optimize the query complexity of tokens in this work.

Niles [Nil15] and Mechler et al. [MMN16] provide a new formulation for tamper-proof hardware tokens that can be reused across different protocol executions. Their security definition is different from GUC security studied in [HPV16].

Recently, Hazay et al. [HPV17] constructed constant round adaptively secure protocols in the stateless token model. In this work, we focus on static corruptions.

## 2 Technical Overview

We first describe the techniques used in our new two round oblivious transfer protocol in the next subsection. In the subsequent subsection, we describe the techniques for the two party computation protocol. We then build upon these techniques to construct the MPC protocol and discuss this in the final subsection.

### 2.1 Two-Round Oblivious Transfer (OT)

We design a new two-round OT protocol based on one-way functions where the sender $\mathcal{S}$ sends a single token $\mathbf{T}$ to the receiver $\mathcal{R}$. Our protocol combines multiple ideas from prior works to address some standard issues that arise when dealing with *stateless* tokens, together with our new ideas for improving upon the parameters achieved in prior works. Below, we discuss our approach for the case where the token is only used for a single execution. However, our approach easily extends to allow for resuability of token.

Our starting approach is to divide the computation into two parts: in the first part, the receiver $\mathcal{R}$ performs a *random* OT execution with the token $\mathbf{T}$. In the second part, $\mathcal{R}$ interacts with the sender $\mathcal{S}$ to perform standard OT using the random OT instance. In more detail, the sender embeds two random strings $(\mathsf{r}_0, \mathsf{r}_1)$ in the token $\mathbf{T}$ and sends it to $\mathcal{R}$. The receiver secret-shares its input bit $\mathsf{b}$ into two parts $(\mathsf{s}, \mathsf{z})$ s.t. $\mathsf{s} \oplus \mathsf{z} = \mathsf{b}$, and then uses $\mathsf{z}$ to learn $\mathsf{r}_\mathsf{z}$ from $\mathbf{T}$. At the same time, $\mathcal{R}$ sends $\mathsf{s} = \mathsf{b} \oplus \mathsf{z}$ to the sender $\mathcal{S}$ to obtain $(M_0, M_1)$ s.t. $M_0 = (\mathsf{m}_0 \oplus \mathsf{r}_0)$, $M_1 = (\mathsf{m}_1 \oplus \mathsf{r}_1)$ if $\mathsf{s} = 0$ and $M_0 = (\mathsf{m}_0 \oplus \mathsf{r}_1)$, $M_1 = (\mathsf{m}_1 \oplus \mathsf{r}_0)$ otherwise. Using the mask $\mathsf{r}_\mathsf{z}$ learned from the token, $\mathcal{R}$ appropriately unmasks one of the two values $(M_0, M_1)$ to learn $\mathsf{m}_\mathsf{b}$.

The immediate problem with this naive approach is that an adversarial receiver can simply reset the stateless token and run it two times on different inputs to learn both $\mathsf{r}_0$ and $\mathsf{r}_1$. Using these masks, the receiver can then recover both $(\mathsf{m}_0, \mathsf{m}_1)$, completely breaking the security.

We address this issue in a similar manner as many prior works such as [GIS+10,CKS+14,DKMN15a]. The basic idea is to require $\mathcal{S}$ to *authenticate* $\mathcal{R}$'s query to the token. Namely, $\mathcal{R}$ commits to its query $\mathsf{z}$ and then obtains a

signature $\sigma$ on the commitment from $\mathcal{S}$.[6] In order to query $\mathbf{T}$ on $\mathsf{z}$, the receiver must provide $\sigma$ and the appropriate decommitment information. The unforgeability of the signature scheme ensures that an adversarial receiver cannot query the token more than once.

**Input-dependent aborts.** Unfortunately, this modification introduces a subtle problem: a malicious sender can subliminally communicate $\mathsf{s}$ to the token by embedding bit $\mathsf{s}$ into the signature value $\sigma$. This allows the token to learn the receiver's input bit $\mathsf{b}$. It can now decide whether or not to abort based on this input bit, which effectively signals the bit $\mathsf{b}$ back to the sender, breaking the security of the protocol.

Similar to [DKMN15a], we address this problem by hiding the signature from the token. Specifically, instead of sending $\sigma$ to $\mathbf{T}$, $\mathcal{R}$ proves knowledge of $\sigma$ via a zero-knowledge argument of knowledge. Since $\mathbf{T}$ is stateless, we require this argument of knowledge to be resettably sound [BGGL01]. Recent works have constructed such argument systems based on one-way functions [CPS13,BP13,COPV13,COP$^+$14,BP15,CPS16].

While this modification prevents subliminal communication from the sender to the token, unfortunately, the protocol still remains susceptible to input-dependent aborts. In particular, an adversarial token can simply decide to abort or not based on the random bit $\mathsf{z}$. This effectively signals $\mathsf{z}$ back to the sender, who combines it with $\mathsf{s}$ to learn the receiver's input $\mathsf{b}$.

A natural idea to prevent such an attack is to secret-share $\mathsf{z}$ into two parts and query the token on each part separately. The hope here would be that the adversarial token can only signal back one of the two secret shares of $\mathsf{z}$ back to the sender, which does not suffice for learning $\mathsf{b}$. Unfortunately, this idea immediately fails since an adversarial token may be *stateful*, and therefore have a joint view of all the queries made by the receiver.

**Leakage-resilient secret-sharing.** Our first step to address the problem of input-dependent aborts is to employ leakage-resilient secret-sharing sharing. Roughly, $\mathcal{R}$ secret-shares its input $\mathsf{b}$ into $2n$ random bits $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$ s.t. $\mathsf{b}$ is the inner product of $(\mathsf{b}_1, \ldots, \mathsf{b}_n)$ with $(\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})$. Each bit $\mathsf{b}_i$ is further secret shared into $(\mathsf{s}_i, \mathsf{z}_i)$ s.t. $\mathsf{b}_i = (\mathsf{s}_i \oplus \mathsf{z}_i)$.

The main idea is that due to the leakage-resilient properties of inner product, even given all of the bits $(\mathsf{z}_1, \ldots, \mathsf{z}_{2n})$, an adversarial token cannot signal back any one bit of information to $\mathcal{S}$ that is sufficient for learning $\mathsf{b}$.

Unfortunately, however, it is not immediately clear how to integrate the above secret-sharing scheme with the rest of the protocol. In particular, while our strategy of performing OT via a random OT is compatible with the XOR-based secret sharing, it does not seem to be compatible with inner-product based secret shar-

[6] For simplicity, here we assume a non-interactive commitment scheme. In order to use a two-round commitment scheme based on one-way functions, we use the token $\mathbf{T}$ to generate the first commitment message.

ing.

**Delegation of Computation.** Towards building a solution, let us first assume that we have a trusted third party that computes the following function $G$: it takes as input the bits $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$, recomputes $\mathsf{b}$ and then outputs one of the two hardcoded values $(\mathsf{m}_0, \mathsf{m}_1)$, depending upon $\mathsf{b}$.

Clearly, given access to such a third party, performing OT is straightforward. Our main idea is to implement such a party via *garbled circuits*. Namely, we augment the functionality of the token $\mathbf{T}$ to compute a garbled circuit for $G$ and send it to the receiver $\mathcal{R}$ so that it can evaluate it on its own. In other words, the token delegates the computation of $G$ to $\mathcal{R}$.

Note, however, that to evaluate the garbled circuit, the receiver needs to obtain input wire labels via OT. Thus, it may seem that we come back in a full circle and not made any progress at all.

The crucial observation is that the input wire labels for the garbled circuit can be obtained in the same manner as earlier, without leaking any information about $\mathsf{b}$. In particular, the receiver uses the same process as described earlier to obtain one of the two wire labels for each bit $\mathsf{b}_i$. Namely, it first queries the token on a random bit $\mathsf{z}_i$ to learn a random mask $\mathsf{r}_i$. At the same time, it obtains the masked input wire labels for the $i$th input from $\mathcal{S}$. It then uses the mask $\mathsf{r}_i$ to recover the wire label corresponding to bit $\mathsf{b}_i$. This process is repeated in parallel for every position $i$.

Since the garbled circuit gets a full view of the input of the receiver, we require the token to prove its well-formedness via a resettable zero-knowledge argument of knowledge [CGGM00,COPV13]. This ensures that the garbled circuit cannot do an input-dependent abort and signal the bit $\mathsf{b}$ back to $\mathcal{S}$. Note that a similar proof could not have been given by the sender $\mathcal{S}$ about the physical token.

*Trapdoor Mechanism.* A crucial issue that arises while proving UC security of the protocol is the following: when $\mathcal{R}$ is corrupted, the proof of well-formedness of the garbled circuit given by the token $\mathbf{T}$ must be simulated in order to "force" the correct output. However, the UC simulator cannot rewind the adversary, nor does it have access to its code! To get around this issue, we implement the following trapdoor generation mechanism that allows the simulator to recover a trapdoor that can then be used to perform straight-line simulation. In the first round, along with the other messages, $\mathcal{R}$ also sends a random string $x$. The sender $S$ then sends $y = \mathrm{OWF}(x)$, where OWF is a one-way function, and a signature on $y$ along with the other messages in the second round. Upon being queried with $y$, the token proves, via a resettable witness-indistinguishable argument of knowledge[7] (RWIAOK), that either the garbled circuit is well-formed or it knows an inverse of $y$. Here, we crucially rely on the asymmetry between the simulator and the adversary: since an honest sender's token is implemented by the simulator, it already knows the trapdoor $x$ that was sent by the adversarial

[7] Such argument systems can be constructed from one-way functions [COPV13].

receiver during the protocol. In contrast, once an adversarial sender learns $x$ in the protocol execution, it has no way of signaling it to its token. At this juncture, we remark that this trapdoor mechanism is crucially used in all our other secure computation protocols as well.

We refer the reader to the technical sections for further details about our OT protocol.

### 2.2 Two-Party Computation

**A Cloning Strategy.** Consider parties $\mathsf{P}_1$ and $\mathsf{P}_2$ with inputs $\mathsf{x}_1$ and $\mathsf{x}_2$ respectively who wish to securely evaluate a function $f$ in two rounds such that both parties learn the output. The main idea at the heart of our protocol is the following: instead of running a two-party computation protocol between remote players that would require several rounds of interaction, we ask a player to construct a clone of itself in the form of a stateless token that can then be remotely activated by its creator to perform the actual two-party computation. We explain this in more detail below.

$\mathsf{P}_1$ creates a clone of itself in the form of a stateless token $\mathbf{T}_1$ and sends it to $\mathsf{P}_2$. $\mathsf{P}_2$ does the same thing by sending a token $\mathbf{T}_2$ to $\mathsf{P}_1$. Then, $\mathsf{P}_2$ can simply execute a secure two-party computation protocol $\Pi$ for $f$ *locally* with $\mathbf{T}_1$, while $\mathsf{P}_1$ can do the same with $\mathbf{T}_2$. An immediate problem with this idea is that since the tokens are stateless, an adversarial $\mathsf{P}_2$, for example, can simply reset the token $\mathbf{T}_1$ during the execution of $\Pi$, which may completely break its security.

**Input Authentication.** We solve this issue by allowing the sender to remotely activate the token only for one input of the other party, in a similar manner as in our OT protocol. We describe the strategy for $\mathsf{P}_1$. Upon being activated, $\mathsf{P}_2$'s token first outputs a commitment to $\mathsf{P}_2$'s input, and proves the knowledge of the committed value using a resettable zero-knowledge argument of knowledge (RZKAOK). Party $\mathsf{P}_1$ then signs this commitment and sends it to $\mathsf{P}_2$. In the course of the two-party computation with token $\mathbf{T}_1$, $\mathsf{P}_2$ proves that its behavior is correct with respect to the input inside the signed commitment.

**Implementing Two-Party Computation.** We implement the actual two-party computation between the token $\mathbf{T}_1$ and $\mathsf{P}_2$ via garbled circuits and OT. (The two-party computation between $\mathbf{T}_2$ and $\mathsf{P}_1$ is implemented in a symmetric manner.) In more detail, the token $\mathbf{T}_1$ prepares and outputs a garbled circuit for the functionality $f(\mathsf{x}_1, \cdot)$ and proves its well-formedness via a RZKAOK. In order to evaluate this garbled circuit, $\mathsf{P}_2$ needs the wire labels corresponding to its input $x_2$, which in turn requires the use of OT.

Instead of performing OTs with the token $\mathbf{T}_1$, $\mathsf{P}_2$ runs multiple parallel executions of OT with $\mathsf{P}_1$, where $\mathsf{P}_2$ plays the receiver and $\mathsf{P}_1$ plays the sender. The role of the OT token in this protocol is played by $\mathbf{T}_1$. By using our new two-round OT protocol, we are able to ensure that the communication between

$\mathsf{P}_1$ and $\mathsf{P}_2$ only requires two rounds. Barring several details concerning the security proof (see below), this already yields a two-round two-party computation protocol.

We now briefly discuss some of the steps in the security proofs of the above protocol. Consider a malicious $\mathsf{P}_2^*$. The simulator can extract its input using the non-black-box extractor of RZKAOK given by token $\mathbf{T}_2$. Here, the extractor requires the code of the token $\mathbf{T}_2$ which is not an issue since $\mathbf{T}_2$ is disconnected from its creator $\mathsf{P}_2^*$ and the environment. Once it obtains the output $\mathsf{y}_2$ from the ideal functionality, it can simulate the two-party computation protocol between token $\mathbf{T}_1$ and $\mathsf{P}_2^*$. Here, we will need to simulate the proof of the well-formedness of the garbled circuit and we will rely on the trapdoor generation mechanism used in the OT protocol to achieve this task.

**Simulating Input Commitment.** Another issue that arises is how does the simulator generate the proofs for the input commitment? That is, in the ideal world, consider the setting where the party $\mathsf{P}_1$ is corrupted. Now, the simulator's token $\mathbf{T}_2$ on behalf of honest party $\mathsf{P}_2$, while interacting with $\mathsf{P}_1$ will have to prove via a RZKAOK that it indeed knows the honest party's input inside the commitment. In the ideal world, $\mathsf{P}_2$ does not know the honest party's input and so the commitment will be just to some random string. However, we can not simulate the RZKAOK argument given by the token as we don't have the code of the environment that it is interacting with in the setting of UC security. Neither can we use the trapdoor mechanism as the trapdoor is generated only much later after interacting with the adversary's token. We overcome this issue by noting that we actually don't need the full power of zero knowledge here and instead, all we require is a resettable strong witness indistinguishable argument of knowledge. That is, we just require that the input commitment being used is changed honestly to an indistinguishable one (a commitment to a random string) and simultaneously can change the proof to prove knowledge of this committed value. As a result, we do the following: in the reduction, we first simulate the RZKAOK argument. Here, we crucially use the fact that this happens only inside the security reduction and the *final UC simulator does not need the environment's code*. We then switch the input commitment to be a commitment of a random string by relying on the hiding property and finally, switch the proof back to honestly prove knowledge of the committed value.

The above discussion ignores several subtleties that arise in the proof. A more detailed explanation of our protocol and proof can be found in Section 5.

### 2.3 Multiparty Computation

We now describe the techniques used in our MPC protocol. At a high level, we follow the same recipe as in the two party case: that is, each party creates a clone of itself that can then be remotely activated by the creator to perform the actual MPC. As in the two party setting, we will also use the trapdoor mechanism described in the OT section to help simulate the resettable arguments in the

security proof. However, there are several additional challenges that arise in the context of MPC and we describe them below.

First, lets describe the approach in more detail. Consider a set of $n$ parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. Now, in order to offload the heavy computation of an actual MPC protocol to remote players, we would require each party $\mathsf{P}_i$ to interact with a set of $(n-1)$ tokens - one each from every other party, in an actual MPC protocol. Unlike the two party setting where we essentially performed a two party computation between a party and a token, here, the tokens can not talk to each other! Therefore, each party $\mathsf{P}_i$ has to facilitate as the channel through which the messages are exchanged between all these tokens taking party in the MPC.

The next question is what sort of MPC protocol do we run amongst $\mathsf{P}_i$ and the $(n-1)$ tokens? Recall that our goal is to base the security of our entire protocol only on the existence of one way functions. In the two party setting, we overcame this issue by running a semi-honest two party protocol based on garbled circuits and oblivious transfer (OT) and composing it with appropriately resettable arguments. We then used our new two round OT protocol to compute the OTs required by the semi-honest construction. Taking a similar approach, we would need to run a semi-honest secure MPC protocol that can be based on just OT and one way functions. While there are several such protocols in literature, a crucial issue that arises is that we would need to instantiate it with an MPC protocol where all the OT executions can be made in parallel once before the execution of the rest of the protocol. We know of protocols in the OT hybrid model[Bea96,Kil88,IPS08,IKO$^+$11] assuming just one way functions that satisfy this structure and we use such an MPC protocol and use our two round OT protocol to run the OT executions. As in the two party setting, we perform the input authentication and trapdoor generation before running the MPC protocol and this help facilitates the proof.

*Extra Round.* The description so far seems to suggest that the protocol runs in only two rounds. However, unlike the two party setting, we need an extra round for the following reason. Let's recall how the actual MPC is computed. Consider party $\mathsf{P}_i$. In order to run the underlying MPC protocol, the $(n-1)$ tokens in possession of $\mathsf{P}_i$ do act as the OT receiver in some executions of the initial parallel OT calls. As a result, the tokens need to know the output of the OT invocations before proceeding with the rest of the computation. However, its not at all clear how to deliver this output to the tokens. To illustrate the issue more clearly with an example, consider two tokens $\mathbf{T}_1$ and $\mathbf{T}_2$ in the presence of party $\mathsf{P}_i$. Let's suppose that in some OT invocation, $\mathbf{T}_1$ is the sender and $\mathbf{T}_2$ is the receiver. Now, clearly, the OT has to be performed "externally" via their token creators as the respective sender and receiver and not amongst the tokens themselves using $\mathsf{P}_i$ as the channel because our OT protocol is not resettably secure. Therefore, lets suppose we perform the OT protocol amongst their respective creators $\mathsf{P}_1$ and $\mathsf{P}_2$. At the end of this OT, the party $\mathsf{P}_2$ only learns the output. However, we need to transmit this to its token $\mathbf{T}_2$. To solve this, in the third round, we

have $\mathsf{P}_2$ send the OT outputs in an encrypted (and signed) form which is then relayed to $\mathbf{T}_2$ via the party $\mathsf{P}_i$.

At this point, we stop and reflect why this was not an issue in the two party setting. There, recall that the only OT to be performed involved the party $\mathsf{P}_i$ as the receiver and the corresponding token (of the other party) as the sender. Therefore, by just running the two round OT protocol, $\mathsf{P}_i$ learns the output and we avoid this extra round.

Finally, to ease the exposition and simplify the proof, unlike in the two party setting, we treat the token that computes the MPC different from the one that takes part in the OT protocol. Hence, we require every party to send two tokens to every other party. We refer the reader to the technical section for more details.

## 3 Preliminaries

**UC Secure Computation.** The UC framework, introduced by [Can01] offers advanced security guarantees since it deals with the security of protocols that may be arbitrarily composed. We include the formal definitions in the full version.

**OT.** Ideal 2-choose-1 oblivious transfer (OT) is a two-party functionality that takes two inputs $m_0, m_1$ from a sender and a bit $b$ from a receiver. It outputs $m_b$ to the receiver and $\perp$ to the sender. We use $\mathcal{F}_{\mathsf{ot}}$ to denote this functionality. The ideal oblivious transfer(OT) functionality $\mathcal{F}_{\mathsf{ot}}$ is formally defined in the supplementary material. Given UC oblivious transfer, it is possible to obtain UC secure two-party computation of any functionality [IPS08,IKO$^+$11].

**Token functionality.** We model a tamper-proof hardware token as an ideal functionality $\mathcal{F}_{\mathsf{WRAP}}$ in the UC framework, following Katz[Kat07]. A formal definition of this functionality can be found in the full version. Note that our ideal functionality models stateful tokens. Although all our protocols use stateless tokens, an adversarially generated token may be stateful.

**Cryptographic primitives.** We use the following primitives all of which can be constructed from one way functions: pseudorandom functions, digital signatures, commitments, garbled circuits.[GGM86,Yao86,Rom90,Nao91]. Additionally, we use the following advanced primitives recently constructed based on one way functions: resettable zero knowledge argument of knowledge, resettably sound zero knowledge argument of knowledge, resettable witness indistinguishable argument of knowledge and resettably sound witness indistinguishable argument of knowledge. [CGGM00,BGGL01,CPS13,BP13,COPV13,COP$^+$14,BP15,CPS16].

**Interactive proofs for a "stateless" player.** We consider the notion of an interactive proof system for a "stateless" prover/verifier. By "stateless", we mean that the verifier has no extra memory that can be used to remember the transcript of the proof so far. Consider a stateless verifier. To get around the issue

of not knowing the transcript, the verifier signs the transcript at each step and sends it back to the prover. In the next round, the prover is required to send this signed transcript back to the verifier and the verifier first checks the signature and then uses the transcript to continue with the protocol execution. Without loss of generality, we can also include the statement to be proved as part of the transcript. It is easy to see that such a scenario arises in our setting if the stateless token acts as the verifier in an interactive proof with another party.

## 4 Oblivious Transfer

In this section, we construct a two round UC oblivious transfer protocol with unbounded reusability based on one-way functions using only one stateless hardware token. The token is sent by the OT sender to the OT receiver in an initial token transfer phase.

We first describe our protocol for the case where the token sent by the OT sender can only be used for a single OT protocol execution. We then describe a modification to make the token *reusable*, such that it can be used to perform an unbounded polynomial number of OT executions between the same pair of parties, with different pairs of inputs.

Formally, we show the following theorem:

**Theorem 4.** *Assuming one-way functions exist, there exists a two round UC secure unbounded OT protocol in the stateless hardware token model.*

Combining this with the result of Ishai et al. [IKO+11], we obtain the following corollary:

**Corollary 5** *Assuming one-way functions exist, there exists a two round UC secure two-party computation protocol using one stateless hardware token where only one party learns the output.*

### 4.1 Overview

Consider a sender $\mathcal{S}$ with inputs $(\mathsf{m}_0, \mathsf{m}_1)$ and a receiver $\mathcal{R}$ with choice bit $\mathsf{b}$ who wish to run an OT protocol.

**Token transfer phase.** Initially, as part of the token transfer phase, $\mathcal{S}$ creates a stateless token $\mathbf{T}$ that has a prf key $\mathsf{k}_{\mathcal{S}}$ and a signing key and verification key pair $(\mathsf{sk}, \mathsf{vk})$ for a signature scheme hardwired into it. Additionally, $\mathcal{S}$ chooses two random strings $(\mathsf{r}_0, \mathsf{r}_1)$ and creates a circuit $\mathcal{C}$ that, given input $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$, outputs $\mathsf{r}_\mathsf{b}$ where $\mathsf{b} = \langle(\mathsf{b}_1, \ldots, \mathsf{b}_n), (\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})\rangle$. (Here, $\langle x, y\rangle$ denotes the inner product of $x$ and $y$.) The sender creates a garbled version of this circuit $\tilde{\mathcal{C}}$ and hardwires it into the token, together with the randomness used to create the garbled circuit. $\mathcal{S}$ sends the token $\mathbf{T}$ to $\mathcal{R}$.

**Round 1.** $\mathcal{R}$ picks a key $\mathsf{k}_{\mathcal{R}}$ for a pseudorandom function and sends $\mathsf{c}$ which is a commitment to this key. Also, $\mathcal{R}$ picks $2n$ bits $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$ uniformly at random

such that $\langle \mathsf{B}_1, \mathsf{B}_2 \rangle = \mathsf{b}$ where $\mathsf{B}_1 = (\mathsf{b}_1, \ldots, \mathsf{b}_n)$ and $\mathsf{B}_2 = (\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})$. Then, for each $i \in [2n]$, $\mathcal{R}$ sends $\mathsf{s}_i = (\mathsf{b}_i \oplus \mathsf{z}_i)$ where $\mathsf{z}_i = \mathsf{PRF}(\mathsf{k}_{\mathcal{R}}, i)$.

**Round 2.** $\mathcal{S}$ computes a signature $\sigma = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{c})$. Also, for each $i \in [2n]$, $\mathcal{S}$ computes $\mathsf{A}_{i,0} = \mathsf{PRF}(\mathsf{k}_{\mathcal{S}}, i, 0)$ and $\mathsf{A}_{i,1} = \mathsf{PRF}(\mathsf{k}_{\mathcal{S}}, i, 1)$. Looking ahead, $\mathsf{A}_{i,0}$ and $\mathsf{A}_{i,1}$ will be the token's output when queried with $\mathsf{z}_i = 0$ or $\mathsf{z}_i = 1$ respectively. Let the pair of labels for the $i^{th}$ input wire to the garbled circuit be $\mathsf{L}_{i,0}$ and $\mathsf{L}_{i,1}$. If $\mathsf{s}_i = 0$, $\mathcal{S}$ computes $\mathsf{Z}_{i,0} = (\mathsf{L}_{i,0} \oplus \mathsf{A}_{i,0})$ and $\mathsf{Z}_{i,1} = (\mathsf{L}_{i,1} \oplus \mathsf{A}_{i,1})$. On the other hand, if $\mathsf{s}_i = 1$, $\mathcal{S}$ computes $\mathsf{Z}_{i,0} = (\mathsf{L}_{i,1} \oplus \mathsf{A}_{i,0})$ and $\mathsf{Z}_{i,1} = (\mathsf{L}_{i,0} \oplus \mathsf{A}_{i,1})$. Also, $\mathcal{S}$ computes $\alpha_0 = (\mathsf{m}_0 \oplus \mathsf{r}_0)$ and $\alpha_1 = (\mathsf{m}_1 \oplus \mathsf{r}_1)$. $\mathcal{S}$ sends $(\mathsf{Z}_{i,0}, \mathsf{Z}_{i,1})$ for each $i \in [2n]$ along with $(\alpha_0, \alpha_1, \sigma)$.

**Output Computation.** First, $\mathcal{R}$ aborts if $\mathsf{Verify}_{\mathsf{vk}}(\mathsf{c}, \sigma) = 0$. Now, for each $i \in [2n]$, $\mathcal{R}$ queries the token $\mathbf{T}$ using input $(\mathsf{z}_i, \mathsf{c}, i)$ along with a resettably sound zero-knowledge argument of knowledge (RSZKAOK) for the following NP statement:

$$\text{There exists } (\mathsf{k}_{\mathcal{R}}, \sigma) \text{ such that } \mathsf{c} = \mathsf{Commit}(\mathsf{k}_{\mathcal{R}}), \mathsf{Verify}_{\mathsf{vk}}(\mathsf{c}, \sigma) = 1 \text{ and } \mathsf{z}_i = \mathsf{PRF}(\mathsf{k}_{\mathcal{R}}, i).$$

$\mathbf{T}$ first verifies the proof. It aborts if the proof doesn't verify. Then, $\mathbf{T}$ outputs $\mathsf{A}_{i,\mathsf{z}_i} = \mathsf{PRF}(\mathsf{k}_{\mathcal{S}}, i, \mathsf{z}_i)$ and $\sigma_i = \mathsf{Sign}_{\mathsf{sk}}(i)$. Now, for each $i$, $\mathcal{R}$ computes the label value as $\mathsf{L}_{i,\mathsf{b}_i} = \mathsf{Z}_{i,\mathsf{z}_i} \oplus \mathsf{A}_{i,\mathsf{z}_i}$.

After this, $\mathcal{R}$ queries the token with the $2n$ signatures - $\sigma_1, \ldots, \sigma_{2n}$ and receives a garbled circuit $\tilde{\mathcal{C}}$ in response along with a resettable zero knowledge (RZK) argument that it was generated correctly. In order to facilitate simulation of this proof, we actually implement it via a resettable witness indistinguishable argument of knowledge (RWIAOK) which can be proven by using a "trapdoor witness" that is generated as follows: $\mathcal{R}$, in the first round of the protocol, picks a random $\bar{\mathsf{x}}$ and sends it to $\mathcal{S}$. In the second round, $\mathcal{S}$ computes $\mathsf{y} = \mathsf{OWF}(\bar{\mathsf{x}})$ and a signature $\sigma_{\mathsf{y}} = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{y})$. Now, when $\mathcal{R}$ queries the token to get the garbled circuit, he also sends $\mathsf{y}$ and gives a RSZKAOK that he knows a signature on $\mathsf{y}$ with respect to the verification key $\mathsf{vk}$. The token, via the RWIAOK proves that either the garbled circuit was correctly generated or that it knows a pre-image of $\mathsf{y}$. Using the corresponding label values, $\mathcal{R}$ evaluates the garbled circuit to recover its output $\mathsf{r}_{\mathsf{b}}$. $\mathcal{R}$ then uses this value along with $\alpha_{\mathsf{b}}$ to recover $\mathsf{m}_{\mathsf{b}}$.

The correctness of the protocol follows by inspection. Below, we provide a brief overview of the security proofs against malicious receivers and malicious senders.

*Security Against a Malicious Receiver.* Consider a malicious receiver $\mathcal{R}^*$. For each $i$, let's suppose $\mathcal{R}^*$ queries $\mathbf{T}$ with $(\mathsf{z}_i, \mathsf{c}, i)$ and a valid RSZKAOK argument. First, from the security of the pseudorandom function, observe that the output of $\mathbf{T}$ for a query containing index $i'$ gives no information at all about its output for index $i \neq i'$. Therefore, we now need to argue that $\mathcal{R}^*$ can not query the token with $(1 - \mathsf{z}_i, \mathsf{c}', i)$ and receive a valid output. If $\mathcal{R}^*$ produces a different $(\mathsf{c}')$ that would break the security of the signature scheme. Fixing $(\mathsf{c}') = (\mathsf{c})$,

we observe that, from the statistical binding property of the commitment, there is a unique $\mathsf{k}_{\mathcal{R}}$ and hence a unique value of $\mathsf{z}_i = \mathsf{PRF}(\mathsf{k}_{\mathcal{R}}, i)$. Therefore, if $\mathcal{R}^*$ produces a valid argument for $\mathsf{z}'_i \neq \mathsf{z}_i$, then that would violate the soundness of the RSZKAOK system.

The simulation strategy in the ideal world is as follows: the simulator first retrieves all the $\mathsf{z}_i$ values by observing the queries to $\mathbf{T}$. It then extracts the receiver's input $\mathsf{b}$ from the set of $\mathsf{z}_i$ and $\mathsf{s}_i$ values. The simulator forwards $\mathsf{b}$ to the ideal OT functionality to receive $\mathsf{m}_{\mathsf{b}}$. It then computes a simulated garbled circuit as output of the token. Note that by using additional signatures on each output of the token, we force the receiver to query for the garbled circuit from the token only after it gets all the label keys and messages from the sender. This ensures that the simulator has enough time to extract the adversary's input and produce a simulated garbled circuit. Further, the simulator observes the query $\bar{\mathsf{x}}$ from the receiver in the first round and uses that as the witness in the RWIAOK given by the token.

*Security Against a Malicious Sender.* To prove security against a malicious sender $\mathcal{S}^*$, the simulator, which receives the token's code $\mathsf{M}$ from the ideal functionality when the token is created, runs the code $\mathsf{M}$ on both $\mathsf{z}_i = 0$ and $\mathsf{z}_i = 1$ for every $i$ by producing simulated RSZKAOK arguments as input. Note that in order to produce simulated RSZKAOK arguments, the simulator requires the code of the verifier which in our case is the token. Observe that this does not violate UC security since the simulator only needs the code of the token (which it does receive as per the model) and not the environment's code. In its interaction with the sender $\mathcal{S}^*$, the simulator picks $\sigma_i$ uniformly at random and not as the output of a PRF. Using the sender's responses $\mathsf{A}_{i,0}, \mathsf{A}_{i,1}$ along with the outputs from the token - $(\mathsf{Z}_{i,0}, \mathsf{Z}_{i,1})$ on both $\mathsf{z}_i = 0$ and $\mathsf{z}_i = 1$, the simulator can compute both the label values for each input wire to the garbled circuit. Further, the simulator sends a random $\mathsf{y}$ as input to receive the garbled circuit $\tilde{\mathcal{C}}$ and produces a simulated RSZKAOK of the signature. Then, from the soundness of the RWIAOK, the simulator is guaranteed that $\tilde{\mathcal{C}}$ was indeed garbled correctly using two messages $(\mathsf{r}_0, \mathsf{r}_1)$. Finally, the simulator can extract both $\mathsf{m}_0$ and $\mathsf{m}_1$ using the garbled circuit $\tilde{\mathcal{C}}$, all the labels and the messages $\alpha_0, \alpha_1$.[8]

Further, note that due to the "leakage resilience" of the inner product, $\mathcal{S}^*$ doesn't learn anything about $\mathsf{b}$ even if the malicious token selectively aborts. That is, $\mathcal{S}^*$ can't learn $\mathsf{b}$ unless it learns all the $\mathsf{b}_i$ values. For this, the token has to signal information about each $\mathsf{z}_i$ by selectively aborting to help $\mathcal{S}^*$ recover the respective $\mathsf{b}_i$ and this can happen only with negligible probability since the $\mathsf{z}_i$'s are essentially picked uniformly at random. That is, in the proof, the situation where the simulator fails to extract both messages while the honest party doesn't abort happens only with negligible probability.

In the above protocol description, we treated the RSZKAOK and RWIAOK argument systems as non-interactive protocols, but in reality they are interactive

[8] An alternate proof strategy is for the simulator to directly extract the values $\mathsf{r}_0$ and $\mathsf{r}_1$ using the extractor of the RWIAOK but we won't delve further into that.

proofs. This doesn't increase the round complexity of our protocol since these protocols are only executed between the receiver and the token. However, since the token is stateless, it can't "remember" anything about the proof. We fix this by simply having the token sign the statement and transcript along with its message in every round.

### 4.2 Construction

*Notation.* Let $n$ denote the security parameter. Let $\mathsf{OWF} : \{0,1\}^n \to \{0,1\}^{2n}$ be a one-way function, $\mathsf{PRF} : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^n$ and $\mathsf{PRF}_1 : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be two pseudorandom functions, $\mathsf{Com} = (\mathsf{Commit}, \mathsf{Decommit})$ be a non-interactive [9]computationally hiding and statistically binding commitment scheme that uses $n$ bits of randomness to commit to one bit, let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme, $(\mathsf{RSZKAOK.Prove}, \mathsf{RSZKAOK.Verify})$ be a resettably-sound zero-knowledge(RSZKAOK) argument of knowledge system for a "stateless verifier" and $(\mathsf{RWIAOK.Prove}, \mathsf{RWIAOK.Verify})$ be a resettable witness indistinguishable (RWIAOK) argument of knowledge system for a "stateless prover" as defined in Section 3. Let $(\mathsf{Garble}, \mathsf{Eval})$ be a garbling scheme for poly sized circuits that take inputs of length $(2n)$ bits and produces an output of length $n$ bits. Let's say the sender $\mathcal{S}$ has private inputs $(\mathsf{m}_0, \mathsf{m}_1) \in \{0,1\}^{2n}$ and receiver $\mathcal{R}$ has private input $\mathsf{b} \in \{0,1\}$.

Note that all these primitives can be constructed assuming the existence of one-way functions.

*NP languages.* We will use the following NP languages in our OT protocol.

1. NP language $L_1^{OT}$ characterized by the following relation $R_1^{OT}$.
   Statement : $\mathsf{st} = (\mathsf{z}, i, \mathsf{c})$
   Witness : $\mathsf{w} = (\mathsf{k}_{\mathcal{R}}, \mathsf{r}, \sigma)$
   $R_1^{OT}(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
   - $\mathsf{z} = \mathsf{PRF}_1(\mathsf{k}_{\mathcal{R}}, i)$ AND
   - $\mathsf{Verify}_{\mathsf{vk}}(\mathsf{c}, \sigma) = 1$
   - $\mathsf{c} = \mathsf{Commit}(\mathsf{k}_{\mathcal{R}}; \mathsf{r})$
2. NP language $L_2^{OT}$ characterized by the following relation $R_2^{OT}$.
   Statement : $\mathsf{st} = (\mathsf{y})$
   Witness : $\mathsf{w} = (\sigma_{\mathsf{y}})$
   $R_2^{OT}(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
   - $\mathsf{Verify}_{\mathsf{vk}}(\mathsf{y}, \sigma_{\mathsf{y}}) = 1$
3. NP language $L_3^{OT}$ characterized by the following relation $R_3^{OT}$.
   Statement : $\mathsf{st} = (\tilde{\mathcal{C}}, \mathsf{y})$
   Witness : $\mathsf{w} = (\mathcal{C}, \mathsf{k}, \mathsf{r}_0, \mathsf{r}_1 \bar{\mathsf{x}})$
   $R_3^{OT}(\mathsf{st}, \mathsf{w}) = 1$ if and only if :

[9] To ease the exposition, we use non-interactive commitments that are based on injective one-way functions. We describe later how the protocol can be modified to use a two-round commitment scheme that relies only on one-way functions without increasing the round complexity of the protocol.

– Either
  - $\tilde{\mathcal{C}} = \mathsf{Garble}(\mathcal{C}, \mathsf{k})$ (AND)
  - circuit $\mathcal{C}$ on input $(\mathsf{b}_1, \ldots, \mathsf{b}_{2n})$, outputs $\mathsf{r}_\mathsf{b}$ where $\mathsf{b} = \langle(\mathsf{b}_1, \ldots, \mathsf{b}_n), (\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})\rangle$.

    (OR)
– $\mathsf{y} = \mathsf{OWF}(\overline{\mathsf{x}})$.

**OT Protocol.** We now describe our two round OT protocol $\pi^{OT}$.

– **Token Exchange Phase:**
$\mathcal{S}$ picks two random keys $\{\mathsf{k}_\mathcal{S}, \mathsf{k}_\mathcal{V}\} \xleftarrow{\$} \{0,1\}^{2n}$ for the function $\mathsf{PRF}$ and computes $(\mathsf{sk}, \mathsf{vk}) \leftarrow (\mathsf{Gen}(n))$. Then, $\mathcal{S}$ creates a single token $\mathbf{T}_\mathcal{S}$ containing the codes in Figure 1 and Figure 2. $\mathcal{S}$ picks two random values $\mathsf{r}_0, \mathsf{r}_1$. Consider a circuit $\mathcal{C}$ that, given input $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$, outputs $\mathsf{r}_\mathsf{b}$ where $\mathsf{b} = \langle(\mathsf{b}_1, \ldots, \mathsf{b}_n), (\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})\rangle$. $\mathcal{S}$ creates a garbled version of this circuit - $\tilde{\mathcal{C}}$ using keys $\{\mathsf{L}_{i,\mathsf{b}}\}$ for all $i \in [2n]$ and $\mathsf{b} \in \{0,1\}$. This is hardwired into the token. $\mathcal{S}$ sends $\mathsf{vk}$ and $\mathbf{T}_\mathcal{S}$ to the receiver $\mathcal{R}$.

– **Oblivious Transfer Phase:**
1. **Round 1:** $\mathcal{R}$ does the following:
   - Choose a random key $\mathsf{k}_\mathcal{R} \xleftarrow{\$} \{0,1\}^n$ for the function $\mathsf{PRF}_1$ and a random string $\overline{\mathsf{x}} \xleftarrow{\$} \{0,1\}^n$. Compute $\mathsf{y} = \mathsf{OWF}(\overline{\mathsf{x}})$.
   - Compute $\mathsf{c} = \mathsf{Commit}(\mathsf{k}_\mathcal{R}; \mathsf{r})$ using a random string $\mathsf{r} \xleftarrow{\$} \{0,1\}^{n^2}$.
   - Pick $2n$ bits $\mathsf{b}_1, \ldots, \mathsf{b}_{2n}$ uniformly at random such that $\langle \mathsf{B}_1, \mathsf{B}_2\rangle = \mathsf{b}$ where $\mathsf{B}_1 = (\mathsf{b}_1, \ldots, \mathsf{b}_n)$ and $\mathsf{B}_2 = (\mathsf{b}_{n+1}, \ldots, \mathsf{b}_{2n})$. Then, for each $i \in [2n]$, compute $\mathsf{s}_i = (\mathsf{b}_i \oplus \mathsf{z}_i)$ where $\mathsf{z}_i = \mathsf{PRF}_1(\mathsf{k}_\mathcal{R}, i)$.
   - Send $\mathsf{c}, \overline{\mathsf{x}}$ and $\{\mathsf{s}_i\}_{i=1}^{2n}$ to $\mathcal{S}$.
2. **Round 2:** $\mathcal{S}$ does the following:
   - Compute $\mathsf{y} = \mathsf{OWF}(\overline{\mathsf{x}})$, $\sigma = \mathsf{Sign}_\mathsf{sk}(\mathsf{c}; \mathsf{r}')$ and $\sigma_\mathsf{y} = \mathsf{Sign}_\mathsf{sk}(\mathsf{y})$.
   - For each $i \in [2n]$, compute $\mathsf{A}_{i,\mathsf{s}_i} = \mathsf{PRF}(\mathsf{k}_\mathcal{S}, i, \mathsf{s}_i)$.
   - Compute $\alpha_0 = (\mathsf{m}_0 \oplus \mathsf{r}_0)$ and $\alpha_1 = (\mathsf{m}_1 \oplus \mathsf{r}_1)$.
   - Send $(\{\mathsf{A}_{i,\mathsf{s}_i}\}_{i=1}^{2n}, \sigma, \sigma_\mathsf{y})$ along with $(\alpha_0, \alpha_1)$ to $\mathcal{R}$.

– **Output Computation Phase:** $\mathcal{R}$ does the following:
- Abort if $\mathsf{Verify}_\mathsf{vk}(\mathsf{c}, \sigma) = 0$ or $\mathsf{Verify}_\mathsf{vk}(\mathsf{y}, \sigma_\mathsf{y}) = 0$.
- For each $i \in [2n]$, query $\mathbf{T}_\mathcal{S}$ with input $(\mathsf{z}_i, i, \mathsf{c}, \text{“}\mathsf{prove}\text{”})$. Using the prover algorithm ($\mathsf{RSZKAOK.Prove}$), engage in an execution of an RSZKAOK argument with $\mathbf{T}_\mathcal{S}$ (who acts as the verifier) for the statement $\mathsf{st}_1 = (\mathsf{z}_i, i, \mathsf{c}) \in L_1^{OT}$ using witness $\mathsf{w}_1 = (\mathsf{k}_\mathcal{R}, \mathsf{r}, \sigma)$. That is, as part of the RSZKAOK, if the next message of the prover is $\mathsf{msg}$, query $\mathbf{T}_\mathcal{S}$ with input $(\mathsf{z}_i, i, \mathsf{c}, \mathsf{msg})$ in that round.
- Let $\{(\mathsf{Z}_{i,\mathsf{z}_i}, \sigma_i)\}_{i=1}^{2n}$ be the outputs received from $\mathbf{T}_\mathcal{S}$. For each $i$, compute $\mathsf{L}_{i,\mathsf{b}_i} = (\mathsf{Z}_{i,\mathsf{s}_i} \oplus \mathsf{A}_{i,\mathsf{s}_i})$.
- Query $\mathbf{T}_\mathcal{S}$ with input $(\sigma_1, \ldots, \sigma_{2n}, \mathsf{y}, \text{“}\mathsf{prove}_1\text{”})$. Using the prover algorithm ($\mathsf{RSZKAOK.Prove}$), engage in an execution of an RSZKAOK argument with $\mathbf{T}_\mathcal{S}$ (who acts as the verifier) for statement $(\mathsf{st}_2 = \mathsf{y}) \in L_2^{OT}$ using witness $\mathsf{w}_2 = (\sigma_\mathsf{y})$. That is, as part of the RSZKAOK, if the prover's

next message is $\mathsf{msg}$, query $\mathbf{T}_{\mathcal{S}}$ with input $(\sigma_1, \ldots, \sigma_{2n}, \mathsf{y}, \mathsf{msg})$ in that round.

- Let $(\tilde{\mathcal{C}}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}})$ be the output of $\mathbf{T}_{\mathcal{S}}$. Using the algorithm ($\mathsf{RWIAOK.Verify}$), engage in an execution of a RWIAOK with $\mathbf{T}_{\mathcal{S}}$ (who acts as the prover) for the statement $\mathsf{st}_3 = (\tilde{\mathcal{C}}, \mathsf{y}) \in L_3^{OT}$. As part of the RWIAOK, if the next message of the verifier is $\mathsf{msg}$, query $\mathbf{T}_{\mathcal{S}}$ with input $(\tilde{\mathcal{C}}, \mathsf{y}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}}, \mathsf{msg})$ in that round. Initially, query with $(\tilde{\mathcal{C}}, \mathsf{y}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}}, \text{“}\mathsf{prove}\text{”})$. Abort if the argument doesn't verify.
- Using the keys $\{\mathsf{L}_{i,\mathsf{b}_i}\}_{i=1}^{2n}$ and the garbled circuit $\tilde{\mathcal{C}}$, run the algorithm $\mathsf{Eval}$ to recover the value $\mathsf{r}_\mathsf{b}$.
- Then, compute $\mathsf{m}_\mathsf{b} = (\alpha_\mathsf{b} \oplus \mathsf{r}_\mathsf{b})$

**Remarks:**

1. To be more precise, we use a 2-round commitment scheme where the first message is actually sent by the token (acting on behalf of the receiver of the commitment) independent of the value being committed to. This has been abstracted out as part of the commitment scheme.
2. The verification key $\mathsf{vk}$ can be output by the token itself instead of being sent by $\mathcal{S}$ along with the token. This would then strictly imply that the token exchange phase has no communication messages.

### 4.3 Token Reusability

Observe that the sender's input messages $(\mathsf{m}_0, \mathsf{m}_1)$ don't appear in the token at all. For each execution, the token just evaluates a garbled circuit $\tilde{\mathcal{C}}$ generated using a circuit $C$ that contains two random strings $(\mathsf{r}_0, \mathsf{r}_1)$. In the current construction, the strings $(\mathsf{r}_0, \mathsf{r}_1)$ and the garbled circuit $\tilde{\mathcal{C}}$ were hardwired into the token. Instead, we can just hardwire two PRF keys - $\mathsf{k}_\mathsf{r}$ and $\mathsf{k}_{\tilde{\mathcal{C}}}$ in the token. Then, the token can use the first key $\mathsf{k}_\mathsf{r}$ to generate the pair of random strings $(\mathsf{r}_0, \mathsf{r}_1)$ and thereby the circuit $C$ for each execution. Similarly, the second key $\mathsf{k}_{\tilde{\mathcal{C}}}$ can be used to generate the randomness required to garble the circuit for that execution. Thus, the same token can be re-used to run an unbounded number of oblivious transfer executions between the same pair of sender and receiver.

### 4.4 Security

We defer the formal proof of security to the full version of the paper.

## 5 Two Round Two-Party Computation

In this section, we study two-party computation in the simultaneous broadcast channel using stateless hardware tokens. We first construct a two round UC secure two-party computation protocol for general functions in this model based on one-way functions using two tokens. Specifically, each party sends a single token to the other party in a token exchange phase prior to the protocol communication phase. Formally, we show the following theorem:

**Constants:** $(\mathsf{k}_{\mathcal{V}}, \mathsf{k}_{\mathcal{S}}, \mathsf{vk}, \mathsf{sk}, \tilde{\mathcal{C}}, \{\mathsf{L}_{i,\mathsf{b}}\})$
**Case 1:** If Input $=(\mathsf{z}_i, i, \mathsf{c}, \mathsf{msg})$:

- If $\mathsf{msg} =$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, i, 0)$.
  2. Using the above randomness and the verifier algorithm (RSZKAOK.Verify), initiate an execution of a RSZK argument with the querying party playing the role of the prover for the statement $\mathsf{st}_1 = (\mathsf{z}_i, i, \mathsf{c}) \in L_1^{OT}$. Output the first message of the verifier.
- If $\mathsf{msg} \neq$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, i, 0)$.
  2. Using the above randomness and msg as the message sent by the prover, run the verifier algorithm (RSZKAOK.Verify) to continue an execution of a RSZK argument with the querying party playing the prover's role for the statement $\mathsf{st}_1 = (\mathsf{z}_i, i, \mathsf{c}) \in L_1^{OT}$.
  3. Compute the next message $\mathsf{msg}'$ of the verifier.
  4. If $\mathsf{msg}' \notin \{\mathsf{accept}, \mathsf{reject}\}$, output $\mathsf{msg}'$. If $\mathsf{msg}' = \mathsf{reject}$, abort.
  5. If $\mathsf{msg}' = \mathsf{accept}$:
     If $\mathsf{z}_i = 0$, compute $\mathsf{Z}_{i,0} = (\mathsf{L}_{i,0} \oplus \mathsf{A}_{i,0})$ and $\mathsf{Z}_{i,1} = (\mathsf{L}_{i,1} \oplus \mathsf{A}_{i,1})$. If $\mathsf{z}_i = 1$, compute $\mathsf{Z}_{i,0} = (\mathsf{L}_{i,1} \oplus \mathsf{A}_{i,0})$ and $\mathsf{Z}_{i,1} = (\mathsf{L}_{i,0} \oplus \mathsf{A}_{i,1})$. Output $(\mathsf{Z}_{i,0}, \mathsf{Z}_{i,1}, \sigma_i = \mathsf{Sign}_{\mathsf{sk}}(i))$.

**Case 2:** If Input $=(\sigma_1, \ldots, \sigma_{2n}, \mathsf{y}, \mathsf{msg})$:

- Abort if $\mathsf{Verify}_{\mathsf{vk}}(i, \sigma_i) = 0$ for any $i \in [2n]$.
- If $\mathsf{msg} =$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, 1^{n+1})$.
  2. Using the above randomness and the verifier algorithm (RSZKAOK.Verify), initiate an execution of a RSZKAOK with the querying party playing the role of the prover for the statement $\mathsf{st}_2 = \mathsf{y} \in L_2^{OT}$. Output the first message of the verifier.
- If $\mathsf{msg} \neq$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, 1^{n+1})$.
  2. Using the above randomness and msg as the message sent by the prover, run the verifier algorithm (RSZKAOK.Verify) to continue an execution of a RSZKAOK with the querying party playing the prover's role for the statement $\mathsf{st}_2 = \mathsf{y} \in L_2^{OT}$.
  3. Compute the next message $\mathsf{msg}'$ of the verifier.
  4. If $\mathsf{msg}' \notin \{\mathsf{accept}, \mathsf{reject}\}$, output $\mathsf{msg}'$. If $\mathsf{msg}' = \mathsf{reject}$, abort.
  5. If $\mathsf{msg}' = \mathsf{accept}$, output $(\tilde{\mathcal{C}}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}} = \mathsf{Sign}_{\mathsf{sk}}(\tilde{\mathcal{C}}, \mathsf{y}))$.

Continues in Figure 2.

Fig. 1: Code of token $\mathbf{T}_{\mathcal{S}}$

**Theorem 6.** *Assuming one-way functions exist, there exists a two round UC-secure two-party computation protocol over simultaneous broadcast channels for*

Continuing from Figure 1.
**Case 3:** If Input $=(\tilde{\mathcal{C}}, \mathsf{y}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}}, \mathsf{msg})$:

- Abort if $\mathsf{Verify}_{\mathsf{vk}}(\tilde{\mathcal{C}}, \mathsf{y}, \sigma_{\tilde{\mathcal{C}},\mathsf{y}}) = 0$.
- If $\mathsf{msg} =$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, 0^{n+1})$.
  2. Using the above randomness and the prover algorithm (RWIAOK.Prove), initiate an execution of a RWIAOK with the querying party playing the role of the verifier for the statement $\mathsf{st}_3 = (\tilde{\mathcal{C}}, \mathsf{y}) \in L_3^{OT}$ using witness $\mathsf{w}_3 = (\mathcal{C}, \{\mathsf{L}_{i,0}, \mathsf{L}_{i,1}\}, \mathsf{r}_0, \mathsf{r}_1, \perp)$ where $i \in [2n], \mathsf{b} \in \{0,1\}$. Output the first message of the prover.
- If $\mathsf{msg} \neq$ "prove", the token does the following:
  1. Consider a random tape defined by $\mathsf{PRF}(\mathsf{k}_{\mathcal{V}}, 0^{n+1})$.
  2. Using the above randomness and msg as the message sent by the prover, run the prover algorithm (RWIAOK.Prove) to continue an execution of a RWIAOK with the querying party playing the verifier's role for the statement $\mathsf{st}_3 \in L_3^{OT}$.

Fig. 2: Continuing code of token $\mathbf{T}_{\mathcal{S}}$

*any functionality $f$ in the stateless hardware token model where each party sends one token.*

### 5.1 Construction

Let $f$ be any two-party functionality. Consider two parties $\mathsf{P}_1$ and $\mathsf{P}_2$ with inputs $\mathsf{x}_1 \in \{0,1\}^n$ and $\mathsf{x}_2 \in \{0,1\}^n$ respectively who wish to compute $f$ on their joint inputs. Below, we describe a two round protocol $\Pi^{2pc}$ for securely computing $f$.

*Notation.* Let $n$ denote the security parameter and $\mathsf{OWF} : \{0,1\}^n \to \{0,1\}^{\mathsf{poly}(n)}$ be a one-way function. Let $\mathsf{PRF} : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^n$ be a pseudorandom function, $\mathsf{Com} = (\mathsf{Commit}, \mathsf{Decommit})$ be a non-interactive [10] statistically binding commitment scheme that uses $n$ bits of randomness to commit to one bit and $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme. Let $\mathsf{RZKAOK} = (\mathsf{RZKAOK.Prove}, \mathsf{RZKAOK.Verify})$ be a resettable zero-knowledge argument of knowledge for a "stateless prover", $\mathsf{RWIAOK} = (\mathsf{RWIAOK.Prove}, \mathsf{RWIAOK.Verify})$ be a resettable witness indistinguishable argument of knowledge for a "stateless prover" and $\mathsf{RSZKAOK} = (\mathsf{RSZKAOK.Prove}, \mathsf{RSZKAOK.Verify})$ be a resettably-sound zero-knowledge argument of knowledge for a "stateless verifier" as defined in Section 3. Let $(\mathsf{Garble}, \mathsf{Eval})$ be a garbling scheme for poly sized circuits that take inputs of length $(n)$ bits and produces outputs of length $n$ bits.

[10] To ease the exposition, we use non-interactive commitments that are based on injective one-way functions. We describe later how the protocol can be modified to use a two-round commitment scheme that relies only on one-way functions without increasing the round complexity of the protocol.

Let $(\mathsf{OT}_1, \mathsf{OT}_2, \mathsf{OT}_3)$ denote the 2-message oblivious transfer protocol from Section 4. That is, the algorithm $\mathsf{OT}_1$ is used by the receiver to compute the first message $\mathsf{ot}_1$. The algorithm $\mathsf{OT}_2$ is used by the sender to compute the second message $\mathsf{ot}_2$ and the algorithm $\mathsf{OT}_3$ is used by the receiver to compute the output.

*NP languages.* We will use the following NP languages in our protocol.

- Language $L_1$ characterized by the following relation $R_1$. Statement : $\mathsf{st} = (\mathsf{b}, \mathsf{c})$
  Witness : $\mathsf{w} = (\mathsf{a}, \mathsf{x}, \mathsf{r})$
  $R_1(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
  - $\mathsf{b} = \mathsf{OWF}(\mathsf{a})$ AND
  - $\mathsf{c} = \mathsf{Commit}(\mathsf{x}; \mathsf{r})$
- Language $L_2$ characterized by the following relation $R_2$:
  Statement : $\mathsf{st} = (\tilde{\mathcal{C}}, \mathsf{b}, \mathsf{c})$
  $\mathsf{w} = (\mathsf{x}, \mathsf{r_c}, \mathcal{C}, \mathsf{k}, \mathsf{a})$
  $R_2(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
  - Either
    - $\mathsf{c} = \mathsf{Commit}(\mathsf{x}; \mathsf{r_c})$ AND
    - $\tilde{\mathcal{C}} = \mathsf{Garble}(\mathcal{C}, \mathsf{k})$ AND
    - circuit $C$, on any input $\alpha$, outputs $f(\mathsf{x}, \alpha)$.

    (OR)
  - $\mathsf{b} = \mathsf{OWF}(\mathsf{a})$
- NP language $L_3$ characterized by the following relation $R_3$.
  Statement : $\mathsf{st} = (\mathsf{c}, \mathsf{vk}_1, \mathsf{b}, \mathsf{vk}_2)$
  Witness : $\mathsf{w} = (\sigma_\mathsf{c}, \sigma_\mathsf{b})$
  $R_3(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
  - $\mathsf{Verify}_{\mathsf{vk}}(\mathsf{c}, \sigma_\mathsf{c}) = 1$ AND
  - $\mathsf{Verify}_{\mathsf{vk}}(\mathsf{b}, \sigma_\mathsf{b}) = 1$

**The Protocol.** We now describe protocol $\Pi^{2pc}$ for UC secure two-party computation in the stateless hardware token model. Let party $\mathsf{P}_1$ have input $\mathsf{x}_1$ and $\mathsf{P}_2$ have input $\bar{\mathsf{x}}_2$. Recall that function to be computed is denoted by $f$.

- **Token Exchange Phase:**
  $\mathsf{P}_1$ does the following:
  - Compute $(\mathsf{sk}_{\mathsf{c}_2}, \mathsf{vk}_{\mathsf{c}_2}) \leftarrow \mathsf{Gen}(n)$, $(\mathsf{sk}_{\mathsf{b}_2}, \mathsf{vk}_{\mathsf{b}_2}) \leftarrow \mathsf{Gen}(n)$, $(\mathsf{sk}_1, \mathsf{vk}_1) \leftarrow \mathsf{Gen}(n)$. Pick keys $\{\mathsf{k}_1^{rzk}, \mathsf{k}_1^{rszk}, \mathsf{k}_1^{rwi}\} \xleftarrow{\$} \{0,1\}^{3n}$ for the $\mathsf{PRF}$.
  - Choose a random string $\mathsf{a}_1$ and compute $\mathsf{b}_1 = \mathsf{OWF}(\mathsf{a}_1)$. Also, compute $\mathsf{c}_1 = \mathsf{Commit}(\mathsf{x}_1; \mathsf{r}_{\mathsf{c}_1})$ using a random string $\mathsf{r}_{\mathsf{c}_1}$.
  - Consider a circuit $\mathcal{C}^1$ that, given an $n$-bit input $(\alpha)$, outputs $f(\mathsf{x}_1, \alpha)$. Create a garbled version of this circuit - $\tilde{\mathcal{C}}^1$ using keys $\{\mathsf{L}_{i,\mathsf{b}}^1\}$ for all $i \in [n]$ and $\mathsf{b} \in \{0,1\}$. This is hardwired into the token. Compute $\sigma_{\tilde{\mathcal{C}}^1} = \mathsf{Sign}_{\mathsf{sk}_1}(\tilde{\mathcal{C}}^1; \mathsf{r}_{\tilde{\mathcal{C}}^1})$ using a random string $\mathsf{r}_{\tilde{\mathcal{C}}^1}$.

- Create a token $\mathbf{T}_1^{2pc}$ containing the codes in Figure 3 and Figure 4. Note that this involves performing steps carried out in the token exchange phase of the OT protocol in Section 4.
- $\mathsf{P}_1$ sends the token $\mathbf{T}_1^{2pc}$ to $\mathsf{P}_2$.

The protocol is symmetric. That is, $\mathsf{P}_2$ sends $\mathbf{T}_2^{2pc}$ to $\mathsf{P}_1$.

– **Communication Phase:**

1. **Round 1:**
   $\mathsf{P}_1$ does the following:
   - For each $i \in [n]$, compute $\mathsf{ot}_{1,i}^{1\to 2} = \mathsf{OT}_1(\mathsf{x}_{1,i})$ where $\mathsf{x}_{1,i}$ denotes the $i^{th}$ bit of $\mathsf{x}_i$. Here the superscript denotes that its sent from $\mathsf{P}_1$ to $\mathsf{P}_2$.
   - Send $(\mathsf{b}_1, \mathsf{c}_1, \mathsf{vk}_{\mathsf{b}_2}, \mathsf{vk}_{\mathsf{c}_2}, \{\mathsf{ot}_{1,i}^{1\to 2}\}_{i=1}^n)$ to $\mathsf{P}_2$ where $\mathsf{b}_1, \mathsf{c}_1$ were computed in the token exchange phase.

   $\mathsf{P}_2$ performs the same operations symmetrically and sends $(\mathsf{b}_2, \mathsf{c}_2, \mathsf{vk}_{\mathsf{b}_1}, \mathsf{vk}_{\mathsf{c}_1}, \{\mathsf{ot}_{1,i}^{2\to 1}\}_{i=1}^n)$ to $\mathsf{P}_1$.
2. **Round 2:**
   $\mathsf{P}_1$ does the following:
   - Using the verifier algorithm ($\mathsf{RZKAOK.Verify}$), engage in an execution of a RZKAOK with the token $\mathbf{T}_2^{2pc}$ (who acts as the prover) for the statement that $(\mathsf{st}_1) \in L_1$ where $\mathsf{st}_2 = (\mathsf{b}_2, \mathsf{c}_2)$. This is done by querying token $\mathbf{T}_2^{2pc}$ with input ("$\mathsf{activate}$"). As part of the RZKAOK, if the next message of the verifier is $\mathsf{msg}$, query the token with input ($\mathsf{msg}$) in that round.
   - Abort if the above argument doesn't verify.
   - Compute $\sigma_{\mathsf{c}_2} = \mathsf{Sign}_{\mathsf{sk}_{\mathsf{c}_2}}(\mathsf{c}_2; \mathsf{r}_{\mathsf{c}_2})$ and $\sigma_{\mathsf{b}_2} = \mathsf{Sign}_{\mathsf{sk}_{\mathsf{b}_2}}(\mathsf{b}_2; \mathsf{r}_{\mathsf{b}_2})$ using random strings $\mathsf{r}_{\mathsf{c}_2}$ and $\mathsf{r}_{\mathsf{b}_2}$.
   - For each $i \in [n]$, compute $\mathsf{ot}_{2,i}^{1\to 2} = \mathsf{OT}_2(\mathsf{L}_{i,0}^1, \mathsf{L}_{i,1}^1, \mathsf{ot}_{1,i}^{2\to 1})$ where $\mathsf{L}_{i,0}^1$ and $\mathsf{L}_{i,1}^1$ are the labels of the garbled circuit $\tilde{\mathcal{C}}^1$.
   - Send $(\sigma_{\mathsf{c}_2}, \sigma_{\mathsf{b}_2}, \mathsf{ot}_{2,1}^{1\to 2}, \ldots, \mathsf{ot}_{2,n}^{1\to 2})$ to $\mathsf{P}_2$.

   $\mathsf{P}_2$ symmetrically sends $(\sigma_{\mathsf{c}_1}, \sigma_{\mathsf{b}_1}, \mathsf{ot}_{2,1}^{2\to 1}, \ldots, \mathsf{ot}_{2,n}^{2\to 1})$ to $\mathsf{P}_1$.

– **Output Computation:**

$\mathsf{P}_1$, does the following :

- For each $i \in [n]$, run the "Output computation phase" of the OT protocol using input $\mathsf{x}_{1,i}$ and $\mathsf{ot}_{2,i}^{2\to 1}$ as the messages from the sender. For any query $\mathsf{msg}$ to be made to the token in the OT protocol, query token $\mathbf{T}_2^{2pc}$ using input ("$OT$", $\mathsf{msg}$). Compute output $\mathsf{L}_{i,\mathsf{x}_{1,i}}^2$ for each $i \in [n]$.
- Query $\mathbf{T}_2^{2pc}$ using input $(\mathsf{c}_1, \mathsf{b}_1, \text{"}\mathsf{2pc}\text{"})$. Using the prover algorithm ($\mathsf{RSZKAOK.Prove}$), engage in an execution of an RSZKAOK argument with $\mathbf{T}_2^{2pc}$ (who acts as the verifier) for statement $\mathsf{st}_3 = (\mathsf{c}_1, \mathsf{b}_1, \mathsf{vk}_{\mathsf{c}_1}, \mathsf{vk}_{\mathsf{b}_1}) \in L_3$ using witness $\mathsf{w}_3 = (\sigma_{\mathsf{c}_1}, \sigma_{\mathsf{b}_1})$. That is, as part of the RSZKAOK, if the next message of the prover is $\mathsf{msg}$, query $\mathbf{T}_2^{2pc}$ with input $(\mathsf{c}_1, \mathsf{b}_1, \mathsf{msg})$ in that round.
- Let $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2})$ be the output of $\mathbf{T}_2^{2pc}$. Then, using the verifier algorithm ($\mathsf{RWIAOK.Verify}$), engage in an execution of a RWIAOK with $\mathbf{T}_2^{2pc}$ (who acts as the prover) for the statement $\mathsf{st}_2 = (\tilde{\mathcal{C}}^2, \mathsf{b}_1, \mathsf{c}_1) \in L_2$. As part

of the RWIAOK, if the verifier's next message is $\mathsf{msg}$, query $\mathbf{T}_2^{2pc}$ with input $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2}, \mathsf{msg})$ in that round. Initially, query with $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2}, \text{“}\mathsf{prove}\text{”})$. Abort if the argument doesn't verify.

- Using the keys $\{\mathsf{L}^2_{i,\mathsf{x}_{1,i}}\}_{i=1}^n$, and the garbled circuit $\tilde{\mathcal{C}}^2$, run the algorithm $\mathsf{Eval}$ to recover the output $\mathsf{y}_1$.

$\mathsf{P}_2$ performs the same operations symmetrically to receive output $\mathsf{y}_2$.

**Note:** For better understanding of the rest of the protocol, this figure actually describes the code of token $\mathbf{T}_2^{2pc}$ created by $\mathsf{P}_2$. The code of $\mathbf{T}_1^{2pc}$ is symmetrical.
**Constants**: $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2}, \{\mathsf{L}^2_{i,0}, \mathsf{L}^2_{i,1}\}_{i=1}^n, \mathsf{x}_2, \mathsf{c}_2, \mathsf{r}_{\mathsf{x}_2}, \mathsf{a}_2, \mathsf{b}_2, \mathsf{k}_2^{rzk}, \mathsf{k}_2^{rszk}, \mathsf{k}_2^{rwi}, \mathsf{PRF}$
$(\mathsf{sk}_{\mathsf{c}_1}, \mathsf{vk}_{\mathsf{c}_1}), (\mathsf{sk}_{\mathsf{b}_1}, \mathsf{vk}_{\mathsf{b}_1}), (\mathsf{sk}_2, \mathsf{vk}_2))$

1. If input = (“$OT$”, $\mathsf{msg}$), respond as done by the token in Section 4 using input as $\mathsf{msg}$.
2. If input = (“$\mathsf{activate}$”), do the following: using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{rzk}, 0^{n+1})$ and the prover algorithm ($\mathsf{RZKAOK.Prove}$), initiate an execution of a RZKAOK with the querying party playing the role of the verifier for the statement $(\mathsf{st}_1) \in L_1$ where $\mathsf{st}_1 = (\mathsf{b}_2, \mathsf{c}_2)$ using witness $(\mathsf{a}_2, \mathsf{x}_2, \mathsf{r}_{\mathsf{x}_2})$. Output the first message of the prover.
3. If input = ($\mathsf{msg}$), do the following: using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{rzk}, 0^{n+1})$ and the prover algorithm ($\mathsf{RZKAOK.Prove}$), continue an execution of a RZKAOK with the querying party playing the role of the verifier for the statement $(\mathsf{st}_1) \in L_1$ where $\mathsf{st}_2 = (\mathsf{b}_2, \mathsf{c}_2)$ using witness $(\mathsf{a}_2, \mathsf{x}_2, \mathsf{r}_{\mathsf{x}_2})$. Output the next message of the prover.

Continues in Figure 4.

Fig. 3: Code of token $\mathbf{T}_2^{2pc}$

**Remark:** In the above description, we were assuming non-interactive commitments (which require injective one way functions) to ease the exposition. In order to rely on just one way functions, we switch our commitment protocol to a two-round one where the receiver sends the first message. Now, we tweak our protocol as follows: after receiving the token, $\mathsf{P}_1$ receives the first round of the commitment from the token $\mathbf{T}_2$ and uses that to compute $\mathsf{c}_1$. $\mathsf{P}_2$ does the same thing symmetrically after interacting with $\mathbf{T}_1$.

**Reusability:** If we want to allow our tokens to be reused an unbounded number of times for performing multiple two party computation protocols between the same pair of parties, we can tweak the protocol as follows: instead of hardwiring $\mathsf{P}_1$'s input $\mathsf{x}_1$ and the garbled circuit $\tilde{\mathcal{C}}_1$ inside the token $\mathbf{T}_1^{2pc}$, we can just hardwire an encryption key $\mathsf{ek}_1$ for a symmetric encryption scheme. Now, in this setting, the tokens are exchanged apriori in an initial token exchange phase. Then, in the first round, when $\mathsf{P}_1$ sends $\mathsf{c}_1 = \mathsf{Commit}(\mathsf{x}_1)$ and message $\mathsf{b}_1$ to

Continuing from Figure 3.

4. If input = $(\mathsf{c}_1, \mathsf{b}_1, \text{“2pc”})$, do the following:
   - If $\mathsf{msg} = \text{“prove”}$, the token does the following: Using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{\mathsf{RSZK}}, 1^{n+1})$ and the verifier algorithm ($\mathsf{RSZKAOK.Verify}$), initiate an execution of a RSZKAOK with the querying party playing the role of the prover for the statement $\mathsf{st}_3 = (\mathsf{c}_1, \mathsf{b}_1, \mathsf{vk}_{\mathsf{c}_1}, \mathsf{vk}_{\mathsf{b}_1}) \in L_3$. Output the first message of the verifier.
   - If $\mathsf{msg} \neq \text{“prove”}$, the token does the following:
     (a) Using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{\mathsf{RSZK}}, 1^{n+1})$ and $\mathsf{msg}$ as the message sent by the prover, run the verifier algorithm ($\mathsf{RSZKAOK.Verify}$) to continue an execution of a RSZKAOK with the querying party playing the prover's role for the statement $\mathsf{st}_3 = (\mathsf{c}_1, \mathsf{b}_1) \in L_3$.
     (b) Compute the next message $\mathsf{msg}'$ of the verifier.
     (c) If $\mathsf{msg}' \notin \{\mathsf{accept}, \mathsf{reject}\}$, output $\mathsf{msg}'$. If $\mathsf{msg}' = \mathsf{reject}$, abort.
     (d) If $\mathsf{msg}' = \mathsf{accept}$, output $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2})$.
5. if input = $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2}, \text{“prove”})$
   - Abort if the signature $\sigma_{\tilde{\mathcal{C}}^2}$ doesn't verify.
   - using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{rwi}, 0^{n+1})$ and the prover algorithm ($\mathsf{RWIAOK.Prove}$), initiate an execution of a RWIAOK with the querying party playing the role of the verifier for the statement $\mathsf{st}_2 = (\tilde{\mathcal{C}}^2, \mathsf{b}_1, \mathsf{c}_1) \in L_2$ using witness $(\mathsf{x}_2, \mathsf{r}_{\mathsf{c}_2}, \mathcal{C}^2, \{\mathsf{L}_{i,0}^2, \mathsf{L}_{i,1}^2\}_{i=1}^n, \perp)$ .
   - Output the first message of the prover.
6. If input = $(\tilde{\mathcal{C}}^2, \sigma_{\tilde{\mathcal{C}}^2}, \mathsf{msg})$, do:
   - Abort if the signature $\sigma_{\tilde{\mathcal{C}}^2}$ doesn't verify.
   - Using a random tape defined by $\mathsf{PRF}(\mathsf{k}_2^{rwi}, 0^{n+1})$ and the prover algorithm ($\mathsf{RWIAOK.Prove}$), continue an execution of a RWIAOK with the querying party playing the role of the verifier for the statement $\mathsf{st}_2 = (\tilde{\mathcal{C}}^2, \mathsf{b}_1, \mathsf{c}_1) \in L_2$ using witness $(\mathsf{x}_2, \mathsf{r}_{\mathsf{c}_2}, \mathcal{C}^2, \{\mathsf{L}_{i,0}^2, \mathsf{L}_{i,1}^2\}_{i=1}^n, \perp)$.
   - Output the next message of the prover.

Fig. 4: Code of token $\mathbf{T}_2^{2pc}$

party $\mathsf{P}_2$, it also sends $\mathsf{ct}_1 = \mathsf{enc}_{\mathsf{ek}_1}(\mathsf{x}_1)$ and $\sigma_{\mathsf{ct}_1} = \mathsf{Sign}(\mathsf{ct}_1)$. That is, it sends an encryption of its input and a signature on this encryption. Party $\mathsf{P}_2$ is now expected to also query the token with $(\mathsf{ct}_1, \sigma_{\mathsf{ct}_1})$ along with $\mathsf{c}_1, \mathsf{b}_1$. The token $\mathbf{T}_1^{2pc}$ verifies the signature, decrypts the message to learn the input $\mathsf{x}_1$ and then proceeds to use it for the rest of the computation as before. A similar procedure is also performed with respect to $\mathsf{P}_2$'s initial messages and $\mathbf{T}_2^{2pc}$'s token responses.

### 5.2 Security

We formally prove security in the full version of the paper.

## 6 Three Round MPC

In this section, we use our unbounded reusable OT protocol to construct a three round UC secure MPC protocol for general functions in the stateless hardware token model amongst $n$ parties based on one-way functions. In this protocol, each party sends two tokens to every other party in a token exchange phase prior to the protocol communication phase. Formally, we show the following theorem:

**Theorem 7.** *Assuming one-way functions exist, there exists a three round protocol that UC-securely realizes any n-party functionality $f$ in the stateless hardware token model where each party sends two tokens to every other party.*

### 6.1 Construction

Let $f$ be any functionality. Consider $n$ parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ with inputs $\mathsf{inp}_1, \ldots, \mathsf{inp}_n$ respectively who wish to compute $f$ on their joint inputs. Below, we describe a three round protocol $\Pi^{mpc}$ for securely computing $f$.

*Notation.* We first list some notation and the primitives used.

- Let $\lambda$ denote the security parameter.
- Let $\mathsf{OWF} : \{0,1\}^\lambda \to \{0,1\}^{\mathsf{poly}(\lambda)}$ be a one-way function. Let $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$ be a pseudorandom function, $\mathsf{Com} = (\mathsf{Commit}, \mathsf{Decommit})$ be a non-interactive statistically binding commitment scheme that uses $\lambda$ bits of randomness to commit to one bit, $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme and $(\mathsf{setup}, \mathsf{enc}, \mathsf{dec})$ be a private key encryption scheme.
- Let $\mathsf{RWIAOK} = (\mathsf{RWIAOK.Prove}, \mathsf{RWIAOK.Verify})$ be a resettable witness indistinguishable argument of knowledge for a "stateless prover" and $\mathsf{RZKAOK} = (\mathsf{RZKAOK.Prove}, \mathsf{RZKAOK.Verify})$ be a resettable zero-knowledge argument of knowledge for a "stateless prover" as defined in Section 3.
- Let $(\mathsf{OT}_1, \mathsf{OT}_2, \mathsf{OT}_3)$ denote the 2-message oblivious transfer protocol from Section 4. That is, the algorithm $\mathsf{OT}_1$ is used by the receiver to compute the first message $\mathsf{ot}_1$. The algorithm $\mathsf{OT}_2$ is used by the sender to compute the second message $\mathsf{ot}_2$ and the algorithm $\mathsf{OT}_3$ is used by the receiver to compute the output.
- Let $\pi$ denote a semi-malicious secure MPC protocol in the correlated randomness model where the correlated randomness is the following: between every pair of parties, there exists an OT channel. That is: between every pair of parties $\mathsf{P}_i$, $\mathsf{P}_j$, there exists a set of tuples $\{s_{0,k}, s_{1,k}, b_k\}_{k \in [p(\lambda)]}$ for some fixed polynomial $p$ such that $\mathsf{P}_i$ knows $\{s_{0,k}, s_{1,k}\}_{k \in [\mathsf{poly}(\lambda)]}$ and $\mathsf{P}_j$ knows $\{b_k\}_{k \in [\mathsf{poly}(\lambda)]}$. We know that such a protocol can be constructed assuming just the existence of one way functions [Bea96,Kil88,IPS08,IKO+11]. Lets say its an $\ell$ round protocol. Let $\pi.\mathsf{Round}_i$ denote the algorithm used by any party to generate the message in round $i$ and let $\pi.\mathsf{Out}$ denote the algorithm used to compute the final output. Let $\mathsf{Sim}^\pi$ denote the simulator for this protocol. We require that $\mathsf{Sim}^\pi$ can generate simulated correlated randomness without knowing the output of the protocol or the input and randomness of the corrupted parties.

*NP languages.* We will use the following NP languages in our protocol.

- Language $L_1$ characterized by the following relation $R_1$. Statement : $\mathsf{st} = (\mathsf{b}, \mathsf{c})$
  Witness : $\mathsf{w} = (\mathsf{a}, \mathsf{x}, \mathsf{r})$
  $R_1(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
  - $\mathsf{b} = \mathsf{OWF}(\mathsf{a})$ AND
  - $\mathsf{c} = \mathsf{Commit}(\mathsf{x}; \mathsf{r})$
- Language $L_2$ characterized by the following relation $R_2$:
  Statement : $\mathsf{st} = (\mathsf{b}, \mathsf{c}, \mathsf{Trans}, \mathsf{msg}, i)$
  $\mathsf{w} = (\mathsf{x}, \mathsf{r_c}, \mathsf{cor.rand}, \mathsf{a})$
  $R_2(\mathsf{st}, \mathsf{w}) = 1$ if and only if :
  - Either
    - $\mathsf{c} = \mathsf{Commit}(\mathsf{x}; \mathsf{r_c})$ AND
    - $\mathsf{msg} = \pi.\mathsf{Round}_i(\mathsf{x}, \mathsf{Trans}, \mathsf{cor.rand})$

    (OR)
  - $\mathsf{b} = \mathsf{OWF}(\mathsf{a})$

**The Protocol.** We now describe protocol $\Pi^{mpc}$ in the stateless hardware token model. Recall that each party $\mathsf{P}_i$ has input $\mathsf{inp}_i$ and the function to be computed is denoted by $f$.

- **Token Exchange Phase:**
  Each party $\mathsf{P}_i$ does the following:
  1. For each party $\mathsf{P}_j$, create token $\mathbf{T}_{\mathsf{ot}}^{i \to j}$ as done in Section 4.
  2. Compute $(\mathsf{sk}_i, \mathsf{vk}_i) \leftarrow \mathsf{Gen}(\lambda)$, $\mathsf{ek}_i \leftarrow \mathsf{setup}(\lambda)$. Pick keys $\{\mathsf{k}_i^{rzk}, \mathsf{k}_i^{rwi}\} \xleftarrow{\$} \{0,1\}^{2n}$ for the $\mathsf{PRF}$.
  3. Choose a random string $\mathsf{a}_i$ and compute $\mathsf{b}_i = \mathsf{OWF}(\mathsf{a}_i)$.
  4. Pick a random string $\mathsf{r}_i$ to run the MPC protocol $\pi$. Set $\mathsf{x}_i = (\mathsf{inp}_i || \mathsf{r}_i)$. Compute $\mathsf{c}_i = \mathsf{Commit}(\mathsf{x}_i; \mathsf{r}_{\mathsf{c}_i})$ using a random string $\mathsf{r}_{\mathsf{c}_i}$.
  5. For each party $\mathsf{P}_k$, create a token $\mathbf{T}_{mpc}^{i \to k}$ containing the code in Figure 5.
  6. $\mathsf{P}_i$ broadcasts all the tokens created above.
- **OT Phase:**
  1. For each $k \in [n]$, every pair of parties $\mathsf{P}_i$ and $\mathsf{P}_j$ with $i > j$ perform a set of $p(\lambda)$ executions of the Oblivious Transfer protocol from Section 4 using the token $\mathbf{T}_{\mathsf{ot}}^{i \to j}$. Here, $\mathsf{P}_i$ picks random inputs $(s_0, s_1)$ for each execution independently and $\mathsf{P}_j$ picks a random bit $\mathsf{b}$ in each execution independently. This process takes two rounds.
  2. In round 3, every party $\mathsf{P}_i$ does the following: For each $k \in [n]$, for each $j \in [n]$ and each OT execution $t$ with party $\mathsf{P}_j$, do:
     (a) If $i > j$, compute $\mathsf{ct} = \mathsf{enc}(\mathsf{ek}, \{s_{0,t}, s_{1,t}\})$ and $\sigma_{\mathsf{ct}} = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{ct})$. Output $(\mathsf{ct}, \sigma_{\mathsf{ct}})$.
     (b) If $i < j$, compute $\mathsf{ct} = \mathsf{enc}(\mathsf{ek}, \{b, s_{b,t}\})$ and $\sigma_{\mathsf{ct}} = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{ct})$. Output $(\mathsf{ct}, \sigma_{\mathsf{ct}})$.
- **Input Commitment Phase:**

1. **Round 1:**
   Each party $\mathsf{P}_i$ broadcasts $(\mathsf{b}_i, \mathsf{c}_i)$ where $\mathsf{b}_i, \mathsf{c}_i$ were computed in the token exchange phase.
2. **Round 2:**
   Each party $\mathsf{P}_i$ does the following:
   - For each $j \in [n]$, using the verifier algorithm ($\mathsf{RZKAOK.Verify}$), engage in an execution of a RZKAOK with the token $\mathbf{T}_{mpc}^{j \to i}$ (who acts as the prover) for the statement that $(\mathsf{st}_j) \in L_1$ where $\mathsf{st}_j = (\mathsf{b}_j, \mathsf{c}_j)$. As part of the RZKAOK, if the next message of the verifier is $\mathsf{msg}$, query the token with input ("$\mathsf{RZKAOK}$", $\mathsf{msg}$) in that round.
   - Abort if the above argument doesn't verify.
   - For each $j \in [n]$, compute and broadcast $\sigma_{\mathsf{c}_j} = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{c}_j)$ and $\sigma_{\mathsf{b}_j} = \mathsf{Sign}_{\mathsf{sk}}(\mathsf{b}_j)$.

– **Computation Phase:**
Each party $\mathsf{P}_i$ does the following :

1. Run an execution of the MPC protocol $\pi$ amongst itself and the $(n-1)$ "MPC" tokens it received. That is, protocol $\pi$ is executed amongst the $n$ parties $\mathbf{T}_{mpc}^{1 \to i}, \ldots, \mathbf{T}_{mpc}^{(i-1) \to i}, \mathsf{P}_i, \mathbf{T}_{mpc}^{(i+1) \to i}, \ldots, \mathbf{T}_{mpc}^{n \to i}$ for the functionality $f$ where the $k^{th}$ party uses input $\mathsf{inp}_i$, randomness $\mathsf{r}_i$ and correlated randomness as the decrypted values in the set of authenticated ciphertexts $\mathsf{ct}$ broadcast by party $\mathsf{P}_k$ in the OT phase.[11] Initiate the protocol by sending "$\mathsf{MPC}$" to each token.
2. Here, $\mathsf{P}_i$ acts as the channel and sends the messages broadcast by any party (aka token) to all the other parties (aka tokens). Along with each message, to each token $\mathbf{T}_{mpc}^{j \to i}$, $\mathsf{P}_i$ also sends the following:
   - The set of $\mathsf{ct}, \sigma_{\mathsf{ct}}$ broadcast by party $\mathsf{P}_j$ in the OT phase. This is the encryption of the correlated randomness for the token $\mathbf{T}_{mpc}^{j \to i}$ in the protocol $\pi$.
   - For each $k \in [n]$, $(\mathsf{b}_k, \mathsf{c}_k, \sigma_{\mathsf{b}_k}, \sigma_{\mathsf{c}_k})$ which are the authenticated input commitments.
3. Whenever a token $\mathbf{T}_{mpc}^{j \to i}$ sends a message $\mathsf{msg}_j$ in round $t$, additionally it also acts as a prover in an execution of a RWIAOK argument with every other token $\mathbf{T}_{mpc}^{k \to i}$ as the verifier for the statement $\mathsf{st}_{j,t} = (\mathsf{b}_k, \mathsf{c}_j, \mathsf{Trans}, \mathsf{msg}_j, t) \in L_2$ using witness $w_{j,t} = (\mathsf{x}_j, \mathsf{r}_{\mathsf{c}_j}, \mathsf{cor.rand}, \perp)$. Here, $\mathsf{Trans}$ denotes the transcript of the protocol upto round $(t-1)$ and $\mathsf{cor.rand}$ is the decryption of all the $\mathsf{ct}$ it receives. Once again $\mathsf{P}_i$ acts as the channel.
4. Finally, compute and output $\mathsf{out} = \pi.\mathsf{Out}(\mathsf{x}_i, \mathsf{Trans})$ where $\mathsf{Trans}$ denotes the transcript of the protocol.

**Remark:** In the above description, we were assuming non-interactive commitments (which require injective one way functions) to ease the exposition. In order

[11] To ease the exposition, we assume that $\mathsf{x}_k$ and $\mathsf{r}_k$ are hardwired inside each token. Instead, we can have each party broadcast encrypted signed versions of them which are sent to the respective token along with the other messages.

**Note:** For better understanding of the rest of the protocol, this figure actually describes the code of token $\mathbf{T}_{mpc}^{j\to i}$ created by party $\mathsf{P}_j$ and sent to $\mathsf{P}_i$. The code of $\mathbf{T}_{mpc}^{i\to j}$ is symmetrical.

**Constants**: $(\mathsf{x}_j, \mathsf{c}_j, \mathsf{r}_{\mathsf{x}_j}, \mathsf{a}_j, \mathsf{b}_j, \mathsf{k}_j^{rzk}, \mathsf{k}_j^{rwi}, \mathsf{PRF}, (\mathsf{sk}, \mathsf{vk}), \mathsf{ek})$

1. If input = ("RZKAOK", msg): using a random tape defined by $\mathsf{PRF}(\mathsf{k}_j^{rzk}, 0^{n+1})$ and the prover algorithm (RZKAOK.Prove), engage in an execution of a RZKAOK with the querying party playing the role of the verifier for the statement $(\mathsf{st}_j) \in L_1$ where $\mathsf{st}_j = (\mathsf{b}_j, \mathsf{c}_j)$ using witness $(\mathsf{a}_j, \mathsf{x}_j, \mathsf{r}_{\mathsf{c}_j})$ where msg is the next message of the verifier in the protocol. Output the next message of the prover.
2. If input = ("MPC", $\{\mathsf{ct}, \sigma_{\mathsf{ct}}\}, \{\mathsf{b}_k, \mathsf{c}_k, \sigma_{\mathsf{b}_k}, \sigma_{\mathsf{c}_k}\}$), do the following:
   (a) If the signatures verify, engage in an execution of the MPC protocol $\pi$ with $(n-1)$ other parties for the functionality $f$ using input $\mathsf{inp}_j$, randomness $\mathsf{r}_j$ and correlated randomness as the set of decryptions of $\{\mathsf{ct}\}$. Here, the querying party acts as the channel.
   (b) In round $t$, if party $\mathsf{P}_k$ sends a message msg, also engage in an execution of a RWIAOK argument acting as the verifier with party $\mathsf{P}_k$ as the prover for the statement $\mathsf{st}_{k,t} = (\mathsf{b}_j, \mathsf{c}_k, \mathsf{Trans}, \mathsf{msg}, t) \in L_2$ where Trans denotes the transcript of the protocol upto round $(t-1)$.
   (c) In round $t$, after sending a message msg, for every other party $\mathsf{P}_k$, engage in an execution of a RWIAOK argument using the prover algorithm with party $\mathsf{P}_k$ acting as the verifier for the statement $\mathsf{st}_{j,t} = (\mathsf{b}_k, \mathsf{c}_j, \mathsf{Trans}, \mathsf{msg}, t) \in L_2$ using witness $w_{j,t} = (\mathsf{x}_j, \mathsf{r}_{\mathsf{c}_j}, \mathsf{cor.rand}, \perp)$ where Trans denotes the transcript of the protocol upto round $(t-1)$ and cor.rand is the decryption of the set of ct using the secret key ek.

Fig. 5: Code of token $\mathbf{T}_{mpc}^{j\to i}$

to rely on just one way functions, we switch our commitment protocol to a two-round one where the receiver sends the first message.

**Reusability:** If we want to allow our tokens to be reused an unbounded number of times for performing multiple MPC protocols between the same set of parties, we can tweak the protocol as follows: instead of hardwiring $\mathsf{P}_i$'s input $\mathsf{x}_i = (\mathsf{inp}_i, \mathsf{r}_i)$ inside the tokens $\mathbf{T}_{mpc}^{i\to j}$ sent by $\mathsf{P}_i$, we can just hardwire an encryption key $\mathsf{ek}_i$ for a symmetric encryption scheme. Now, in this setting, the tokens are exchanged apriori in an initial token exchange phase. Then, in the first round, when $\mathsf{P}_i$ sends $\mathsf{c}_i = \mathsf{Commit}(\mathsf{x}_i)$ and message $\mathsf{b}_i$, it also sends $\mathsf{ct}_i = \mathsf{enc}_{\mathsf{ek}_i}(\mathsf{x}_i)$ and $\sigma_{\mathsf{ct}_i} = \mathsf{Sign}(\mathsf{ct}_i)$. That is, it sends an encryption of its input and a signature on this encryption. Every party $\mathsf{P}_j$ is now expected to also query the token $\mathbf{T}_{mpc}^{i\to j}$ with $(\mathsf{ct}_i, \sigma_{\mathsf{ct}_i})$ along with $\mathsf{c}_i, \mathsf{b}_i$ in every query. The token verifies the signature, decrypts the message to learn the input $\mathsf{x}_i$ and then proceeds to use it for the rest of the computation.

### 6.2 Security

We formally prove security in the full version of the paper.

## References


AAG+14. Shashank Agrawal, Prabhanjan Ananth, Vipul Goyal, Manoj Prabhakaran, and Alon Rosen. Lower bounds in the hardware token model. In *TCC*, 2014.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, 1996.

BGGL01. Boaz Barak, Oded Goldreich, Shafi Goldwasser, and Yehuda Lindell. Resettably-sound zero-knowledge and its applications. *FOCS*, 2001.

BJOV18. Saikrishna Badrinarayanan, Abhishek Jain, Rafail Ostrovsky, and Ivan Visconti. Non-interactive secure computation from one-way functions. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 118–138. Springer, 2018.

BP13. Nir Bitansky and Omer Paneth. On the impossibility of approximate obfuscation and applications to resettable cryptography. In *STOC*, 2013.

BP15. Nir Bitansky and Omer Paneth. On non-black-box simulation and the impossibility of approximate obfuscation. *SIAM J. Comput.*, 2015.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

CDPW07. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, 2007.

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

CGS08. Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for UC secure computation using tamper-proof hardware. In *EUROCRYPT*, 2008.

CJS14. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. *CCS*, 2014.

CKS+14. Seung Geol Choi, Jonathan Katz, Dominique Schröder, Arkady Yerukhimovich, and Hong-Sheng Zhou. (efficient) universally composable oblivious transfer using a minimal number of stateless tokens. In *TCC*, 2014.

COP+14. Kai-Min Chung, Rafail Ostrovsky, Rafael Pass, Muthuramakrishnan Venkitasubramaniam, and Ivan Visconti. 4-round resettably-sound zero knowledge. In *TCC*, 2014.

COPV13. Kai-Min Chung, Rafail Ostrovsky, Rafael Pass, and Ivan Visconti. Simultaneous resettability from one-way functions. In *FOCS*, 2013.

CPS13. Kai-Min Chung, Rafael Pass, and Karn Seth. Non-black-box simulation from one-way functions and applications to resettable security. In *STOC*, 2013.

CPS16. Kai-Min Chung, Rafael Pass, and Karn Seth. Non-black-box simulation from one-way functions and applications to resettable security. *SIAM J. Comput.*, 2016.

DKM11. Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. Unconditional and composable security using a single stateful tamper-proof hardware token. In *TCC*, 2011.

DKM12. Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. Statistically secure linear-rate dimension extension for oblivious affine function evaluation. In *ICITS*, 2012.

DKMN15a. Nico Döttling, Daniel Kraschewski, Jörn Müller-Quade, and Tobias Nilges. From stateful hardware to resettable hardware using symmetric assumptions. In *ProvSec*, 2015.

DKMN15b. Nico Döttling, Daniel Kraschewski, Jörn Müller-Quade, and Tobias Nilges. General statistically secure computation with bounded-resettable hardware tokens. In *TCC*, 2015.

DMMN13. Nico Döttling, Thilo Mie, Jörn Müller-Quade, and Tobias Nilges. Implementing resettable uc-functionalities with untrusted tamper-proof hardware-tokens. In *TCC*, 2013.

GGM86. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 1986.

GIS+10. Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, 2010.

GKR08. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, 2008.

HPV16. Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In *TCC 2016-B*, 2016.

HPV17. Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Constant round adaptively secure protocols in the tamper-proof hardware model. In *PKC*, 2017.

IKO+11. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, 2011.

IPS08. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, 2008.

Kat07. Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, 2007.

Kil88. Joe Kilian. Founding cryptography on oblivious transfer. In *STOC*, 1988.

Kol10. Vladimir Kolesnikov. Truly efficient string oblivious transfer using resettable tamper-proof tokens. In *TCC*, 2010.

MMN16. Jeremias Mechler, Jörn Müller-Quade, and Tobias Nilges. Universally composable (non-interactive) two-party computation from untrusted reusable hardware tokens. *IACR Cryptology ePrint Archive*, 2016:615, 2016.

MS08. Tal Moran and Gil Segev. David and goliath commitments: UC computation for asymmetric parties using tamper-proof hardware. In *EUROCRYPT*, 2008.

Nao91. Moni Naor. Bit commitment using pseudorandomness. *J. Cryptology*, 1991.

Nil15. Tobias Nilges. *The Cryptographic Strength of Tamper-Proof Hardware*. PhD thesis, Karlsruhe Institute of Technology, 2015.

Rom90. John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 387–394. ACM, 1990.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.