Resettably Secure Computation

Vipul Goyal Amit Sahai

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

Abstract

The notion of resettable zero-knowledge (rZK) was introduced by Canetti, Goldreich, Goldwasser and Micali (FOCS'01) as a strengthening of the classical notion of zero-knowledge. A rZK protocol remains zero-knowledge even if the verifier can reset the prover back to its initial state anytime during the protocol execution and force it to use the same random tape again and again. Following this work, various extensions of this notion were considered for the zero-knowledge and witness indistinguishability functionalities.

In this paper, we initiate the study of resettability for more general functionalities. We first consider the setting of resettable two-party computation where a party (called the user) can reset the other party (called the smartcard) anytime during the protocol execution. After being reset, the smartcard comes back to its original state and thus the user has the opportunity to start interacting with it again (knowing that the smartcard will use the same set of random coins). In this setting, we show that it is possible to secure realize *all PPT computable functionalities* under the most natural (simulation based) definition. Thus our results show that in cryptographic protocols, the reliance on randomness and the ability to keep state can be made significantly weaker. Our simulator for the aforementioned resettable two-party computation protocol (inherently) makes use of non-black box techniques. Second, we provide a construction of *simultaneous* resettable multi-party computation with an honest majority (where the adversary not only controls a minority of parties but is also allowed to reset any number of parties at any point). Interestingly, all our results are in the plain model.

1 Introduction

The notion of resettable zero-knowledge (rZK) was introduced by Canetti et al [CGGM00] with a motivation towards obtaining zero-knowledge protocols for highly adversarial environments. In rZK, the verifier is given the additional power that anytime during the protocol execution, it can "reset" the prover back to its initial state thus restarting the prover with the same configuration and coin tosses. This notion is motivated by several natural questions. Firstly, it address the question: "is zero-knowledge possible when the prover uses the *same random coins* in more than one execution?" and (surprisingly) gives a positive answer to it. Secondly, it shows that zero-knowledge protocols can be securely implemented by devices which can neither reliably keep state nor toss coins online. An example of such a device might be a resettable stateless "smartcard" with secure hardware (which can be reset, e.g., by switching off its power) [CGGM00]. Thus, rZK can be viewed as "zero-knowledge protocols for stateless devices". Canetti et al [CGGM00] provide rZK proof system (for NP) with non-constant number of rounds and resettable witness indistinguishable (rWI) proof system with a constant number of rounds.

Canetti et al [CGGM00] observed that since the prover can be reset, an adversarial verifier can actually achieve the effect of interacting with the prover *concurrently* in several sessions, thus in particular implying concurrent zero knowledge [DNS98]. In more detail, the verifier can start a session with the prover and while the interaction is still in progress, can reset the prover to start another interaction. It could reset the prover again in the middle to that interaction and *come back to the previous interaction* by just running the protocol identically up to the point where it reset the prover before. Thus, the verifier

gets the flexibility of choosing its messages in one interaction based on the messages of the prover in some other interaction. In other words, the verifier can essentially interact with the prover in various sessions and can interleave these sessions as it pleases. Thus, a rZK protocol is a concurrent ZK protocol as well [CGGM00]. Following this work, research on improving the round complexity of concurrent ZK also led to rZK with improved round complexity [KP01, PRS02].

Subsequent to the introduction of this notion, various extensions were considered. Barak, Goldreich, Goldwasser and Lindell [BGGL01] studied *resettably-sound* ZK (rsZK) where it is the verifier that can be reset by the prover. The main challenge is to design a ZK protocol which retains its soundness even when verifier uses the same random coins in multiple executions. Relying the non black-box techniques of Barak [Bar01], Barak et al were able to construct constant round rsZK arguments. These rsZK arguments were also used to construct resettable zero-knowledge arguments of knowledge [BGGL01]. An open question raised by [BGGL01] is: do there exist ZK protocols where both the prover and the verifier are resettable by each other? While this still remains an open problem, partial progress was made in [DL07]. The notion of resettability was also studied extensively in the bare public key model introduced by [CGGM00] with a goal of obtaining more efficient protocols (see [YZ07] and the references therein).

Going Beyond ZK – Our Contributions. Common to all the prior work on the notion of resettability is that they consider only the zero-knowledge (or closely related) functionality. This raises the following natural question:

"Do there exist other classes of functionalities for which resettably secure protocols can be obtained?"

In other words, do there exist protocols for other tasks such that the "security" is retained even if one of parties participating in the protocol can be reset by the other one? We initiate the study of general resettable computation and answer the above question in the affirmative.

–**Resettable Two-Party Computation.** In fact, we prove a general completeness theorem for resettable computation showing that it is possible to construct a protocol to securely realize *any PPT computable functionality*. Our results are for the setting where one party (called the user) can reset the other party (called the smartcard) during the protocol execution. We first formalize this notion of resettable two-party computation and give a natural simulation based security definition for it. We then construct a general two-party computation protocol under standard (polynomial time) cryptographic assumption. We in fact give a "resettability compiler" which can compile any semi-honest secure (in the standalone setting) protocol into one that is resettably secure. The simulator for our resettable two-party computation protocol makes use of non-black box techniques. We note that non-black box simulation is inherent since Barak et al [BGGL01] show that it is impossible to obtain rsZK arguments (for languages outside $\mathcal{BPP}$) using black-box simulation and rsZK arguments for **NP** is only a special case of a two-party functionality.

–**Resettable Multi-Party Computation.** Given the above results, two natural questions that arise then are: (a) Do there exists general secure resettable *multi-party* computation protocols (where one or more of the parties are resettable)?, and, (b) Can the construction be extended to handle cases where both parties can potentially reset each other? Towards that end, we first observe that the our construction for resettable two-party computation can be extended using standard techniques to show the existence of resettable multi-party computation (where only one of the parties can be reset) with dishonest majority. Next, we offer a construction of *simultaneous* resettable multi-party computation (where all the parties are resettable) assuming a majority of the parties behave honestly. That is, the adversary not only controls a minority of parties but is also allowed to reset any number of honest parties at any point in the protocol execution. At the core of our construction is a new construction of families of multi-party 1-round (and thus automatically resettable) zero-knowledge arguments of knowledge which are simulation sound [Sah99].

Our results show that in cryptographic protocols, the reliance on randomness and the ability to keep state can be made significantly weaker. We in fact show a simple transformation from any resettably

secure protocol (as per our definitions) to a fully stateless one. By this we mean that the party which was resettable in the original protocol need not maintain any state at all in the transformed protocol.

Concurrency vs Resettability. As discussed earlier, if a party can be reset, the other party can actually achieve the effect of interacting with it *concurrently* in several sessions. This fact is the reason for the folklore that a resettably secure protocol should also be concurrently secure (i.e., concurrently self-composable). However far reaching impossibility results have been proven showing that a large class of functionalities cannot be securely realized [Lin03, Lin04] (even in the fixed roles setting) in the plain model. This stands in sharp contrast to our general positive results for resettable two-party computation.

In resettable two-party computation, an adversarial user already has the power to reset the smartcard and interact with it as many times as it likes. This fact that the number of interactions cannot be controlled is precisely what makes resettable two-party computation possible. In the formulation of our ideal model for defining security, we give the adversarial user the power to interact with the smartcard as many times it likes. Given that an adversarial user is allowed to reset the smartcard in the real model (thus creating new problems), emulating the view of such an adversary in the ideal world is only possible if several interactions with the smartcard are allowed. In other words, we are only allowing the user to do in the ideal world what he is already allowed to do in reality. By giving our ideal adversary such extra power, we are able to construct protocols satisfying our definition in the plain model. In our construction, the number of times the simulator sends the reset request in the ideal world is polynomially related to the number of times the real world adversary sends the reset request. An open problem raised by the current work is to design a construction having a *precise simulation* [MP06] with respect to the number of outputs.

Combining our results with the impossibility results of Lindell [Lin03, Lin04], we get a first separation between the notions of resetability and concurrency. That is, a resettably secure protocol (as per our definitions) is not always a concurrently secure protocol (even for fixed roles) and vice versa (this direction is implicit in [CGGM00]). In fact there exists a large class of functionalities for which concurrently self-composable protocols do not exist but resettably secure protocols do (as we show that they exist for every two-party functionality).

Stateless vs Stateful Devices. Our results have interesting implications about the power of stateless devices. Consider a stateless device (holding an input, possibly unable to toss coins online) trying to run a protocol for secure computation of some functionality with a user. Our results show that *stateless devices can run secure computation protocols for every task.* Of course, stateless devices can only be used when one does not want to limit the number of protocol executions that the device carries out (with a particular input).

What if one does want to limit the number of interactions that the device carries out? We remark that it is also possible to obtain the following "best of both worlds" protocol in such a case. *In case* the adversary is not able to reset the device during the protocol interaction, the protocol provides a traditional security guarantee (i.e., security as per the ideal/real world simulation paradigm, see Canetti [Can00]). However if it turns out that the adversary *was successfully* able to reset the device, the *maximum* the adversary can do is to achieve the effect of running the protocol several times with the device (possibly choosing different input each time).

While "protocols for stateless devices" are naturally useful for weak and inexpensive devices (e.g., smartcard), other applications of such protocols could include making a powerful server (providing services to many clients) stateless to prevent denial of service attacks.

Universally Composable Multi-Party Computation using Tamper Proof Hardware. To show one example of the power of our results, we consider the recent work on obtaining universally composable multi-party computation using tamper proof hardware tokens [Kat]. As noted before, broad impossibility results have been proven showing that a large class of functionalities cannot be UC securely realized in

the plain model [CF01, CKL06]. These severe impossibility results motivated the study of other models involving some sort of *trusted* setup assumptions (assuming a trusted third party), where general positive results can be obtained. To avoid these trust assumptions (while still maintaining feasibility of protocols), Katz recently proposed using a *physical setup*. In his model, the physical setup phase includes the parties exchanging tamper proof hardware tokens implementing some functionality.

The security of the construction in [Kat] relies on the ability of the tamper-resistant hardware to maintain state (even when, for example, the power supply is cut off). In particular, the parties need to execute a four-round coin flipping protocol with the tamper-resistant hardware. Using our techniques, one can immediately relax this requirement and make the token completely stateless. In particular, we can apply our compiler to the coin flipping protocol in [Kat] and obtain a new construction where the token, when fed with an input x, only outputs $f(x)$ for a fixed f and halts. A construction having such a property was first obtained recently by Chandran et al [CGS08] by relying on techniques that are very specific to the setting of UC secure computation with tamper proof hardware. However our construction has an added advantage that a possibly adversarial token does not learn any information about the input of the honest party using it (and hence the security is retained even when the adversarial creator of the token "recaptures" it at a later point of time). This is leads to the first construction of UC secure computation with stateless and recapturable tokens. On the downside, as opposed to [CGS08], the security of this construction would rely on the adversary "knowing" the code of the tokens which it distributes to the honest parties (see [CGS08] for more details).

Open Problems. The main question left open by the current work is: "Do there exist two-party and multi-party computation protocols in the *dishonest majority* setting where more than one party is resettable?". Eventually, one would like to construct simultaneous resettable multi-party computation where the adversary can control any number of parties and can reset any number of honest parties at any point (or show that such protocols cannot exist). The first step to make any progress towards answering the above questions is proving or disproving the existence of a simultaneous resettable zero-knowledge argument (first mentioned as an open problem in the work of Barak et al [BGGL01]).

2 The Resettable Ideal Model

2.1 Resettable Two-Party Computation

Informally speaking, our model for resettable two-party computation is as follows. We consider a smartcard $\mathbb{S}$ holding several inputs $x_1^1, \ldots, x_1^{num}$ and random tapes $\omega_1^1, \ldots, \omega_1^{num}$. We denote by X_1 the full input and randomness vector (i.e., the concatenation of all the inputs and random tapes) held by $\mathbb{S}$. The ith *incarnation* of the smartcard $\mathbb{S}$ is a deterministic strategy defined by the pair (x_1^i, ω_1^i) as in [CGGM00]. **We stress that while each incarnation has its own random tape, as in [CGGM00], when a particular incarnation is reset, it starts over with the same random tape.** Thus, we model different smartcards as the different incarnations. We consider a user $\mathbb{U}$ holding an input x_2 interested in interacting with the ith incarnation of the smartcard $\mathbb{S}$. The user activates the ith incarnation of the smartcard and runs a protocol with it to securely compute a function $f(.,.)$. We do not explicitly define a way of activating the ith incarnation; it could either be through physical means or by sending an initial message to $\mathbb{S}$ specifying the incarnation $\mathbb{U}$ would like to interact with. At any point during the protocol execution, the user $\mathbb{U}$ can *reset* the smartcard $\mathbb{S}$ to its initial state, thus, having a chance to start interaction again with any incarnation with a fresh input. At the end of the protocol, both the parties get the output $f(x_1, x_2)$, $x_1 = x_1^i$ where i and x_2 are the incarnation and the inputs which were most recently selected by the user $\mathbb{U}$. We remark that we only consider *one side resettability*, that is, the smartcard $\mathbb{S}$ is not allowed to reset the user $\mathbb{U}$.

To formalize the above requirement and define security, we extend the standard paradigm for defining secure computation. We define an ideal model of computation and a real model of computation, and

require that any adversary in the real model can be *emulated* (in the specific sense described below) by an adversary in the ideal model. In a given execution of the protocol we assume that all inputs have length κ, the security parameter. We consider a static adversary which chooses whom to corrupt before execution of the protocol. In our model, both the parties get the same output (the case where parties should get different outputs can be easily handled using standard techniques). Finally, we consider *computational* security only and therefore restrict our attention to adversaries running in probabilistic polynomial time.

IDEAL MODEL. In the ideal model there is a trusted party which computes the desired functionality based on the inputs handed to it by the players. Then an execution in the ideal model proceeds as follows:

Select incarnation The user $\mathbb{U}$ sends an incarnation index i to the trusted party which then passes it on to the smartcard $\mathbb{S}$.

Inputs The smartcard $\mathbb{S}$ has input x_1 while the user $\mathbb{U}$ has input x_2.

Send inputs to trusted party Both $\mathbb{S}$ and $\mathbb{U}$ send their inputs to the trusted party. An honest party always sends its real inputs to the trusted party. A corrupted party, on the other hand, may decide to send modified value to the trusted party.

Trusted party computes the result The trusted party sets the result to be $f(x_1, x_2)$. It generates and uses uniform random coin if required for the computation of f.

Trusted party sends results to adversary The trusted party sends the result $f(x_1, x_2)$ to either $\mathbb{S}$ or $\mathbb{U}$ depending upon whoever is the adversary.

Trusted party sends results to honest players The adversary, depending on its view up to this point, does the following. It either sends the *abort* signal in which case the trusted party sends $\perp$ to the honest party. Or it could signal the trusted party to *continue* in which case the trusted party sends the result $f(x_1, x_2)$ to the honest party.

Reset ideal world at any point In case the user $\mathbb{U}$ is the adversary, during the execution of any of the above steps, it can send the signal *reset* to the trusted party. In that case, the trusted party sends *reset* to the smartcard $\mathbb{S}$ and the ideal world comes back to the *select incarnation* stage.

Outputs An honest party always outputs the response it received from the trusted party. The adversary outputs an arbitrary function of its entire view throughout the execution of the protocol.

For a given adversary $\mathcal{A}$, the *execution of f in the ideal model* on X_1, x_2 is defined as the output of the honest parties along with the output of the adversary resulting from the process above. It is denoted by $\text{IDEAL}_{f,\mathcal{A}}(X_1, x_2)$.

REAL MODEL. An honest party follows all instructions of the prescribed protocol, while a adversarial party may behave arbitrarily. If the user $\mathbb{U}$ is the adversarial party, it can *reset* the smartcard $\mathbb{S}$ at any point during the protocol execution. After getting reset, $\mathbb{S}$ comes back to its original state which it was in when starting the protocol execution thus allowing $\mathbb{U}$ to choose a fresh input and start interaction again with any incarnation of the smartcard $\mathbb{S}$. At the conclusion of the protocol, an honest party computes its output as prescribed by the protocol. Without loss of generality, we assume the adversary outputs exactly its entire view of the execution of the protocol.

For a given adversary $\mathcal{B}$ and protocol Σ for resettably computing f, the *execution of Σ in the real model* on X_1, x_2 (denoted $\text{REAL}_{\Sigma,\mathcal{B}}(X_1, x_2)$) is defined as the output of the honest parties along with the output of the adversary resulting from the above process.

Having defined these models, we now define what is meant by a resettable two-party computation protocol. By *probabilistic polynomial time* (PPT), we mean a probabilistic Turing machine with non-uniform advice whose running time is bounded by a polynomial in the security parameter κ. By *expected probabilistic polynomial time* (EPPT), we mean a Turing machine whose *expected* running time is bounded by some polynomial, for *all* inputs.

Definition 1 *Let f and Σ be as above. Protocol Σ is a* secure protocol for computing f *if for every* PPT *adversary $\mathcal{A}$ corrupting either of the two players in the real model, there exists an* EPPT *adversary $\mathcal{S}$ corrupting that player in the ideal model, such that:*

$$\{\text{IDEAL}_{f,\mathcal{S}}(X_1, x_2)\}_{(X_1,x_2)\in(\{0,1\}^*)^2} \stackrel{c}{\equiv} \{\text{REAL}_{\Sigma,\mathcal{A}}(X_1, x_2)\}_{(X_1,x_2)\in(\{0,1\}^*)^2}.$$

Our real model translates to the so called *multiple incarnation non-interleaving* setting in the terminology of [CGGM00]. This setting was shown to be equivalent to the *multiple incarnation interleaving* setting for the case of zero-knowledge and their proof can be extended to the general case as well. In other words, a protocol Σ which is secure when the user $\mathbb{U}$ is allowed only to interact with one incarnation at a time remain secure even if $\mathbb{U}$ is allowed to *concurrently* interact with any number of incarnation simultaneously. For simplicity of exposition, we only consider the setting when the inputs of smartcard $\mathbb{S}$ are all fixed in advance (while $\mathbb{S}$ is acting as honest party in the protocol). However we remark that our protocols also work for the more general case when the inputs of $\mathbb{S}$ are adaptively chosen possibly based on the outputs in the previous protocol executions. More details regarding these issues will be provided in the full version.

2.2 Simultaneous Resettable Multi-Party Computation

For lack of space, we provide the model for this case in Appendix B. The main changes from the two-party case is that we consider an adversary who controls a minority of the parties and can reset any number of honest parties at any point during the protocol.

2.3 Extensions

In this section, we informally describe two extensions which can be applied to our constructions proven secure as per our definitions. More formal details will be provided in the full version.

Going from Resettable to Stateless. Any protocol which is resettably secure can be transformed to a stateless protocol using relatively standard techniques. In other words, the parties which were allowed to be resettable in the original protocol need not maintain any state at all in the transformed protocol. By a stateless device we mean that the device only supports a "request-reply" interaction (i.e., the device just outputs $f(x)$ when fed with x for some fixed f). We describe the case of two party first assuming both the parties are resettable (the case where one party is resettable is only simpler). Let we have parties P_1 and P_2 participating in the original resettably secure protocol Σ. Now we define a protocol Σ' having parties P_1' and P_2'. Each of these parties will have a secret key of a CCA-2 secure encryption scheme and a secret MAC key. The party P_1' computes the first message to be sent in the protocol Σ' by running P_1 internally. However it then sends to P_2' not only the computed message but also an encryption of the *current state* of P_1 and a MAC on it. Party P_2' similarly computes the reply by feeding the received message to P_2 and sends to P_1' not only the computed reply but also (a) the received encrypted state of P_1 and the MAC, and, (b) an encryption of the current state of P_2 and a MAC on it using its own keys. Thus for the next round, P_1' can decrypt, verify and *load* the received state into P_1, feed it the incoming reply and then compute the next outgoing message. P_2 can similarly continue with the protocol. The case of multi-party protocols can also be handled by first transforming the given protocol into one where only *one party* sends a message in any given round and then applying ideas similar to the one for the two party case to this resulting protocol.

Getting the Best of Both Worlds. One might ask the question: is it possible to have a single protocol such that *in case* the adversary is not able to reset the device, the protocol provides a traditional security guarantee (i.e., security as per the ideal/real world simulation paradigm, see Canetti [Can00]). However if it turns out that the adversary *is successfully* able to reset the device, the protocol still provides security

as per the resettable ideal model definition presented above. We remark that it is easy to transform both our constructions into ones which provide such a best of both worlds guarantee (however we do not know if our transformation works for all constructions). We build a counter into the device which gets incremented with every protocol execution (whether successfully completed or not). The randomness used by the device for a protocol execution comes from the application of a PRF to the current counter value. This guarantees that in case the device *is* able to keep such a counter successfully, the randomness used in each execution is fresh and independent of others. Thus, it is easy to show that one can use a significantly simpler simulator (which can only handle standalone executions) to prove security of our constructions in such a setting.

3 Building Blocks

Our protocols make use of the following building blocks: a commitment scheme COM based on one way permutations, computational zero-knowledge proofs and proofs of knowledge, zaps [DN00], resettable sound zero-knowledge arguments [BGGL01] and the PRS concurrent zero-knowledge preamble [PRS02]. More details about these building blocks are provided in Appendix A for lack of space.

4 Resettable Two-Party Computation

4.1 The Construction

We now describe how to transform any given two party protocol Π (which is only semi-honest secure) into a resettably secure protocol Σ. Prior to the beginning of the protocol, we assume that the smartcard $\mathbb{S}$ and the user $\mathbb{U}$ have agreed upon the incarnation of $\mathbb{S}$ for the protocol (see Section 2.1). As noted in Section 2.1, each incarnation of the smartcard $\mathbb{S}$ has its own independent random tape. We assume that the private inputs to $\mathbb{S}$ and $\mathbb{U}$ in the protocol Π are x_1 and x_2 respectively. The smartcard $\mathbb{S}$ denotes the party which can be reset by the other party $\mathbb{U}$ in the protocol Σ. We view the random tape of the smartcard as a tuple (G, R_{rs}). Here G denotes the description of a function $G : \{0,1\}^{\leq \text{poly}(\kappa)} \rightarrow \{0,1\}^{\text{poly}(\kappa)}$ taken from an ensemble of pseudorandom functions and R_{rs} denotes a random string which $\mathbb{S}$ will use while acting as a verifier of a *resettable sound zero-knowledge* argument (see Section A.4 for more details). Let R denote the uniform distribution. The protocol proceeds as follows.

PRS Preamble Phase

1. $\mathbb{U} \rightarrow \mathbb{S}$: Generate $r_2 \stackrel{\$}{\leftarrow} R$ and let $\beta = (x_2, r_2)$. Here r_2 is the randomness to be used (after coin flipping with $\mathbb{S}$) by the user $\mathbb{U}$ at various stages of the protocol Σ (including to carry out the protocol Π) as explained later on. We assume that r_2 is of sufficient size to allow $\mathbb{U}$ to execute all such stages. Generate random shares $\{\beta^0_{i,\ell}\}^k_{i,\ell=1}, \{\beta^1_{i,\ell}\}^k_{i,\ell=1}$ such that $\beta^0_{i,\ell} \oplus \beta^1_{i,\ell} = \beta$ for every i, ℓ. Using the commitment scheme COM, commit to all these shares. Denote these commitments by $\{B^0_{i,\ell}\}^k_{i,\ell=1}, \{B^1_{i,\ell}\}^k_{i,\ell=1}$.

   Let msg be the concatenation of all these commitment strings, i.e., $msg = B^0_{1,1}||\ldots||B^0_{k,k}$ $||B^1_{1,1}||\ldots||B^1_{k,k}$. We call msg to be the *determining message* of this session (since it commits the user $\mathbb{U}$ to its input and randomness). The random tape used by the smartcard $\mathbb{S}$ to carry out rest of the protocol Σ (except when acting as a verifier of a resettable sound ZK argument) will be determined by the application of the pseudorandom function G to the determining message msg. Again, we assume that $G(msg)$ is of sufficient size to allow the execution of all the steps.

2. $\mathbb{U} \leftrightarrow \mathbb{S}$: The user $\mathbb{U}$ and the smartcard $\mathbb{S}$ will now use a resettable-sound zero-knowledge argument system (rsP, rsV) (relying a non-black box simulator [BGGL01]). $\mathbb{U}$ emulates the prover rsP and proves the following statement to the resettable verifier rsV (emulated by $\mathbb{S}$): the above PRS commit

phase is a *valid commit phase* (see Appendix A.5). In other words, there exist values $\hat{\beta}$, $\{\hat{\beta}^0_{i,\ell}\}^k_{i,\ell=1}$, $\{\hat{\beta}^1_{i,\ell}\}^k_{i,\ell=1}$ such that (a) $\hat{\beta}^0_{i,\ell} \oplus \hat{\beta}^1_{i,\ell} = \hat{\beta}$ for every i, ℓ, and, (b) Commitments $\{B^0_{i,\ell}\}^k_{i,\ell=1}, \{B^1_{i,\ell}\}^k_{i,\ell=1}$ can be decommitted to $\{\hat{\beta}^0_{i,\ell}\}^k_{i,\ell=1}, \{\hat{\beta}^1_{i,\ell}\}^k_{i,\ell=1}$.

The user $\mathbb{U}$ uses a fresh (uncommitted) random tape for emulation of the prover rsP. The random tape used by $\mathbb{S}$ to emulate the resettable verifier rsV comes from R_{rs}. Going forward, this will be the case with all the resettable sound zero-knowledge arguments in our protocol Σ.

3. For $\ell = 1, \ldots k$:

   (a) $\mathbb{S} \rightarrow \mathbb{U}$: Send challenge bits $b_{1,\ell}, \ldots, b_{k,\ell} \stackrel{\$}{\leftarrow} \{0,1\}^k$.

   (b) $\mathbb{U} \rightarrow \mathbb{S}$: Decommit to $B^{b_{1,\ell}}_{1,\ell}, \ldots, B^{b_{k,\ell}}_{k,\ell}$.

   The random tape required by $\mathbb{S}$ to generate the above challenge bits comes from $G(msg)$.

4. $\mathbb{S} \rightarrow \mathbb{U}$: Since the underlying protocol Π is secure only against semi-honest adversaries, the random coins used by each party are required to be unbiased. Hence $\mathbb{S}$ generates $r'_2 \stackrel{\$}{\leftarrow} R$ (using the random tape from $G(msg)$) and sends it to $\mathbb{U}$. Define $r''_2 = r_2 \oplus r'_2$. Now r''_2 is the randomness which will be used by $\mathbb{U}$ for carrying out the protocol Π (among other things).

Smartcard Input Commitment Phase

1. $\mathbb{S} \rightarrow \mathbb{U}$: Generate a string $r_1 \stackrel{\$}{\leftarrow} R$ and let $\alpha = (x_1, r_1)$. Commit to α using the commitment scheme COM and denote the commitment string by A. The random tape required to generate the string r_1 and to compute the commitment A comes from $G(msg)$.

2. $\mathbb{S} \leftrightarrow \mathbb{U}$: Now $\mathbb{S}$ has to give a proof of knowledge of the opening of the commitment A to $\mathbb{U}$. In other words, $\mathbb{S}$ has to prove that it knows a value $\hat{\alpha} = (\hat{x_1}, \hat{r_1})$ such that the commitment A can be decommitted to $\hat{\alpha}$. This proof is given as follows. $\mathbb{S}$ and $\mathbb{U}$ use an ordinary computational zero-knowledge proof of knowledge system (P_{pok}, V_{pok}) (see Section A.2) where $\mathbb{S}$ and $\mathbb{U}$ emulates the prover P_{pok} (proving the above statement) and the verifier V_{pok} respectively. However the random tape used by $\mathbb{U}$ to emulate the verifier V_{pok} is required to come from r''_2. To achieve this, the interaction proceeds as follows. Let t'_{pok} be the number of rounds in (P_{pok}, V_{pok}) where a round is defined to have a message from P_{pok} to V_{pok} followed by a reply from V_{pok} to P_{pok}. For $j = 1, \ldots t'_{pok}$:

   - $\mathbb{S} \rightarrow \mathbb{U}$: $\mathbb{S}$ sends the next prover message computed as per the system (P_{pok}, V_{pok}). The random tape used by $\mathbb{S}$ to emulate P_{pok} comes from $G(msg)$.
   - $\mathbb{U} \rightarrow \mathbb{S}$: $\mathbb{U}$ sends the next verifier message computed as per the system (P_{pok}, V_{pok}) *using randomness* r''_2.
   - $\mathbb{U} \leftrightarrow \mathbb{S}$: $\mathbb{U}$ and $\mathbb{S}$ now execute a resettable-sound zero-knowledge argument where $\mathbb{U}$ emulates the prover rsP and proves the following statement to rsV emulated by $\mathbb{S}$: the PRS commit phase has a major decommitment $\hat{\beta} = (\hat{x_2}, \hat{r_2})$ such that the sent verifier message is consistent with the randomness $\hat{r_2} \oplus r'_2$ (where r'_2 was as sent by $\mathbb{S}$ in the PRS preamble phase).

   The above system can be seen as a *resettable zero-knowledge argument of knowledge* system [BGGL01]. However when used in our context as above, the simulator of this system will be straightline.

3. $\mathbb{U} \rightarrow \mathbb{S}$: $\mathbb{U}$ generates $r'_1 \stackrel{\$}{\leftarrow} R$ using the random tape r''_2 and sends it to $\mathbb{S}$. Define $r''_1 = r_1 \oplus r'_1$. Now r''_1 is the randomness which will be used by $\mathbb{S}$ for carrying out the protocol Π.

4. $\mathbb{U} \leftrightarrow \mathbb{S}$: $\mathbb{U}$ and $\mathbb{S}$ now execute a resettable-sound zero-knowledge argument. $\mathbb{U}$ emulates the prover rsP and proves the following statement to rsV emulated by $\mathbb{S}$: the PRS commit phase has a major decommitment (see Section A.5) $\hat{\beta} = (\hat{x_2}, \hat{r_2})$ such that message r'_1 is consistent with the randomness $\hat{r_2} \oplus r'_2$.

Secure Computation Phase

Let the underlying protocol Π have t rounds[1] where one round is defined to have a message from $\mathbb{S}$ to $\mathbb{U}$ followed by a reply from $\mathbb{U}$ to $\mathbb{S}$. Let transcript T_1^j (resp. T_2^j) be defined to contain all the messages exchanged between $\mathbb{S}$ and $\mathbb{U}$ before the point party $\mathbb{S}$ (resp. $\mathbb{U}$) is supposed to send a message in round j. Now, each message sent by either party in the protocol Π is compiled into a *message block* in Σ. For $j = 1, \ldots t$:

1. $\mathbb{S} \rightarrow \mathbb{U}$: $\mathbb{S}$ sends the next message m_1^j $(= \Pi(T_1^j, x_1, r''_1))$ as per the protocol Π. Now $\mathbb{S}$ has to prove to $\mathbb{U}$ that the sent message m_1^j was honestly generated using input x_1 and randomness r''_1. In other words, $\mathbb{S}$ has to prove the following statement: there exist a value $\hat{\alpha} = (\hat{x_1}, \hat{r_1})$ such that: (a) the message m_1^j is consistent with the input $\hat{x_1}$ and the randomness $\hat{r_1} \oplus r'_1$ (i.e., $m_1^j = \Pi(T_1^j, \hat{x}, \hat{r_1} \oplus r'_1)$), and, (b) commitment A can be decommitted to $\hat{\alpha}$.

   This proof is given as follows. $\mathbb{S}$ and $\mathbb{U}$ use an ordinary computational zero-knowledge proof system (P, V) (see Section A.2) where $\mathbb{S}$ and $\mathbb{U}$ emulates the prover P (proving the above statement) and the verifier V respectively. However the random tape used by $\mathbb{U}$ to emulate the verifier V is required to come from r''_2. To achieve this, the interaction proceeds as follows. Let t' be the number of rounds in (P, V) where a round is defined to have a message from P to V followed by a reply from V to P. For $j' = 1, \ldots t'$:

   - $\mathbb{S} \rightarrow \mathbb{U}$: $\mathbb{S}$ sends the next prover message computed as per the system (P, V). The random tape used by $\mathbb{S}$ to emulate P comes from $G(msg)$.
   - $\mathbb{U} \rightarrow \mathbb{S}$: $\mathbb{U}$ sends the next verifier message computed as per the system (P, V) *using randomness* r''_2.
   - $\mathbb{U} \leftrightarrow \mathbb{S}$: $\mathbb{U}$ and $\mathbb{S}$ now execute a resettable-sound zero-knowledge argument where $\mathbb{U}$ emulates the prover rsP and proves the following statement to rsV emulated by $\mathbb{S}$: the PRS commit phase has a major decommitment $\hat{\beta} = (\hat{x_2}, \hat{r_2})$ such that the sent verifier message is consistent with the randomness $\hat{r_2} \oplus r'_2$.

   To summarize, the random tape used by $\mathbb{U}$ to emulate the verifier V is required to be committed in advance. However $\mathbb{S}$ is free to use any random tape while emulating the prover P (although in the interest of its own security, $\mathbb{S}$ is instructed to use randomness from $G(msg)$.

2. $\mathbb{U}$ sends the next message m_2^j $(= \Pi(T_2^j, x_2, r''_2))$ as per the protocol Π. $\mathbb{U}$ and $\mathbb{S}$ now execute a resettable-sound zero-knowledge argument where $\mathbb{U}$ emulates the prover rsP and proves to $\mathbb{S}$ that m_2^j was honestly generated using input x_2 and randomness r''_2. More precisely, $\mathbb{U}$ proves the following statement: the PRS commit phase has a major decommitment $\hat{\beta} = (\hat{x_2}, \hat{r_2})$ such that m_2^j is consistent with the input $\hat{x_2}$ and the randomness $\hat{r_2} \oplus r'_2$.

This completes the description of the protocol Σ. The usage of randomness in the above protocol can be summarized as follows. After sending the very first message (i.e., the determining message msg), the only fresh randomness that can be used by the user $\mathbb{U}$ is while emulating the prover rsP of resettable sound zero-knowledge arguments. The smartcard $\mathbb{S}$ is essentially "free" to use any randomness it wants

[1]This assumption is only made for simplicity of exposition. It is easy to extend our construction to handle protocols whose round complexity is not upper bounded by a fixed polynomial.

(except while computing messages of the underlying protocol Π). An honest $\mathbb{S}$ always sets its random tape to $G(msg)$ to carry out the protocol Σ .

At the end of above protocol Σ, both the parties will hold the desired output. We stress that we require only standard (standalone) semi-honest security from the underlying protocol Π. Thus, when we set the underlying protocol Π to be the constant round two-party computation protocol of Yao [Yao86], the resulting protocol Σ has k $(= \omega(\log \kappa))$ rounds. To obtain a constant round protocol, the first step would be the construction of a concurrent zero-knowledge argument system in a constant number of rounds. We also remark that the above resettable two-party computation also implies (under standard assumptions) resettable multi-party computation (where only one of the parties can be reset) with dishonest majority. The construction for resettable multi-party computation can be obtained using standard techniques from the two-party one (i.e., the $n-1$ "non-resettable" parties will use a regular multi-party computation protocol [GMW87] among them to emulate a single party holding $n-1$ inputs).

In appendix C, we prove that the protocol Σ is a resettable two-party computation protocol by proving the following two theorems.

Theorem 1 (Security Against a Malicious $\mathbb{U}^*$) *The compiled protocol Σ is secure against a malicious $\mathbb{U}^*$ as per the definition in Section 2.1.*

Theorem 2 (Security Against a Malicious $\mathbb{S}^*$) *The compiled protocol Σ is secure against a malicious $\mathbb{S}^*$ in the sense of the definition in Section 2.1.*

5 Simultaneous Resettable Multi-Party Computation with Honest Majority

5.1 The Construction

We now describe how to transform any given protocol Π (which is only semi-honest secure) into a simultaneous resettably secure protocol Σ with honest majority. We assume n parties $P_1, \ldots, P_n$ where a majority of the parties behave honestly. All the parties are "resettable", or in other words, the adversarial parties can reset any number of honest parties at any time during the protocol execution. We assume that before the protocol starts, the parties have agreed upon which incarnation will be used by which party. The private inputs of parties $P_1, \ldots, P_n$ are denoted by $x_1, \ldots, x_n$ respectively. Let R denote the uniform distribution. The protocol Σ proceeds as follows.

Input Commitment Phase

Each party P_i does the following computations. Any randomness required for these computations comes from the random tape of (appropriate incarnation of) P_i which is potentially reusable in other sessions.

- Generate a function $G_i : \{0,1\}^{\leq \text{poly}(\kappa)} \rightarrow \{0,1\}^{\text{poly}(\kappa)}$ randomly from an ensemble of pseudorandom functions and let $\alpha_i = (x_i, G_i)$. Compute a commitment to α_i using the commitment scheme COM and denote it by A_i.
- Generate the first verifier message $Z_i \leftarrow f_{zaps}(\kappa)$ of a zap system (see Section A.3).
- Generate a pair (PK_i, SK_i) of public and secret keys using the key generation algorithm of a semantically secure public key encryption system having perfect completeness.
- Generate n strings $a_i^1, \ldots, a_i^n$ from the domain of a one way function F. For all j, compute $b_i^j = F(a_i^j)$.

Additionally, we assume that a party P_i has a random tape $R_{i,zkv}$ which it uses for the verification of messages of a 1 round ZKAOK system as we explain later on. P_i now broadcasts the values $A_i, Z_i, PK_i, b_i^1, \ldots, b_i^n$. Let the string broadcast (i.e., $A_i||Z_i||PK_i||\ b_i^1||\ldots||b_i^n$) be denoted by msg_i. The string msg_i is called the *determining message* of party P_i for this session. Note that since a party P_i may have to reuse its random tape, the determining message msg_i may be identical across various protocol executions.

The random tape used by P_i to carry out rest of the protocol Σ (except for the verification of the messages of the 1 round ZKAOK system) will be determined by the application of the pseudorandom function G_i to the (concatenation of) determining messages of all other parties. That is, denote $R_i = G_i(msg_1||\ldots||msg_{i-1}||msg_{i+1}||\ldots||msg_n)$. Now R_i serves as the random tape of P_i for the rest of the protocol. We assume that R_i is of sufficient size to allow the execution of all the steps.

Construction of a 1-round zero-knowledge argument of knowledge. We now describe the construction of a family of 1-round zero-knowledge argument of knowledge (ZKAOK) systems. The (i,j)th argument system is used by party P_i to prove statements to party P_j. Additionally, the argument systems (i,j) and (k,ℓ), where $i \neq k$, are simulation sound w.r.t. each other. To prove a statement $x \in L$ to party P_j, a party P_i holding the witness w (for the given witness relation) proceeds as follows:

- P_i breaks the witness w into n shares $w_1, \ldots, w_n$ using the Shamir threshold secret sharing scheme [Sha79] such that a majority of the shares are sufficient to reconstruct the witness w. For all k, P_i encrypts w_k under the public key PK_k. Let the ciphertext be denoted by C_k. P_i now broadcasts all the n ciphertexts so generated.
- P_i finally generates and sends the prover message of the zap system (acting on the verifier message Z_j) proving that one of the following statements is true:
  1. The ciphertexts $C_1, \ldots, C_n$ represent the encryption of shares of a valid witness. More precisely, there exists strings $\hat{w_1}, \ldots, \hat{w_n}$ such that: (a) for all k, C_k is a valid encryption of $\hat{w_k}$, and, (b) $\hat{w_1}, \ldots, \hat{w_n}$ are valid shares of a single string $\hat{w}$ as per the Shamir secret sharing scheme, and, (c) $\hat{w}$ is a valid witness for the witness relation (i.e., $x \in L$).
  2. The ciphertexts $C_1, \ldots, C_n$ represent the encryption of shares of a majority of preimages of the strings $b_1^i, \ldots, b_n^i$ under the one way function F. More precisely, there exists strings $\hat{s_1}, \ldots, \hat{s_n}$ such that: (a) for all k, C_k is a valid encryption of $\hat{s_k}$, and, (b) $\hat{s_1}, \ldots, \hat{s_n}$ are valid shares of a single string $\hat{s}$ as per the Shamir secret sharing scheme, and, (c) $\hat{s} = (\hat{a_1}^i, \ldots, \hat{a_n}^i)$ and there exists a set S_{maj} of indices such that $|S_{maj}| > n/2$ and for all $\ell \in S_{maj}$, $b_\ell^i = F(\hat{a_\ell}^i)$.

Thus, we have a *trapdoor condition* which allows a party P_i to give a simulated argument using the preimages of $b_1^i, \ldots, b_n^i$. Note that the "trapdoor" for each party is "independent". That is, informally speaking, even given the preimages of strings $b_1^i, \ldots, b_n^i$, a party P_j with $j \neq i$ will be unable to give a simulated argument.

Coin Flipping Phase

Since the underlying protocol Π is secure only against semi-honest adversaries, the random coins used by each party in Π are required to be unbiased. Hence the parties run a 2 round coin flipping phase to generate a long unbiased public random string as given below. As noted before, the random tape that a party P_i uses to execute this stage (i.e., generate of random strings, commitment and messages of the 1-round ZKAOK system) comes from R_i. However the random tape (if needed) used by P_i to verify the messages of the ZKAOK system comes from $R_{i,zkv}$.

- In the first round, each party P_i generates $R'_i \stackrel{\$}{\leftarrow} R$ and broadcasts a commitment B_i to R'_i using the commitment scheme COM. For all $j \neq i$, party P_i additionally broadcasts a ZKAOK (using the (i,j)th 1-round ZKAOK system) proving that the commitment B_i was correctly computed using randomness R_i. More precisely, P_i proves that there exists a string $\hat{\alpha}_i = (\hat{x}_i, \hat{G}_i)$ such that: (a) the commitment A_i can be decommitted to $\hat{\alpha}_i$, and, (b) the commitment B_i to a random string (and the random string itself) was computed using randomness $\hat{G}_i(msg_1||\ldots||msg_{i-1}||msg_{i+1}||\ldots||msg_n)$. Note that this ZKAOK is given for a specific witness relation such that the witness allows extraction of such a $\hat{\alpha}_i$.

- In the second round, each party P_i broadcasts the committed string R'_i (without providing any decommitment information). For all $j \neq i$, party P_i additionally broadcasts a ZKAOK (using the (i,j)th 1-round ZKAOK system) proving that the commitment B_i can be decommitted to the string R'_i. Denote $r'_1||\ldots||r'_n = R'_1 \oplus \cdots \oplus R'_n$. At this point, the strings $r'_1, \ldots, r'_n$ are all guaranteed to be random.

Each P_i further privately generates a random r_i. Define $r''_i = r_i \oplus r'_i$. Now r''_i is the randomness that will be used by party P_i to carry out the underlying protocol Π in the next stage.

Secure Computation Phase

Let the underlying protocol Π have t rounds[2] where any number of parties can send a message in any given round. Let transcript T^j be defined to contain all the messages broadcast *before* the round j. Now, for $j = 1, \ldots t$:

1. P_i sends the next message m_i^j $(= \Pi(T^j, x_i, r''_i))$ as per the protocol Π. Note that m_i^j could potentially be $\perp$.

2. For all $k \neq i$, party P_i additionally broadcasts a ZKAOK (using the (i,k)th 1-round ZKAOK system) proving that the sent message m_i^j was honestly generated using input x_i and randomness r''_i. In other words, P_i proves the following statement: there exist a value $\hat{\alpha}_i = (\hat{x}_i, \hat{G}_i)$ such that: (a) the message m_i^j is consistent with the input $\hat{x}_i$ and the randomness $\hat{r}_i \oplus r'_i$ (i.e., $m_i^j = \Pi(T^j, \hat{x}_i, \hat{r}_i \oplus r'_i)$) where $\hat{r}_i$ is generated from $\hat{G}_i$, and, (b) commitment A_i can be decommitted to $\hat{\alpha}_i$. As before, the random tape used by P_i for generation and verification of the messages of ZKAOK system comes from R_i and $R_{i,zkv}$ respectively.

This completes the description of the protocol Σ. The usage of randomness in the above protocol can be summarized as follows. After sending the very first message (i.e., the determining message msg_i), the only fresh and uncommitted random tape that *can potentially* be used by a malicious party P_i is for the generation and verification of the 1-round ZKAOK messages (although the honest parties are instructed to use the committed random tape for the *generation* of 1-round ZKAOK messages).

Applying the above transformation to the constant round protocol of Beaver et al [BMR90], we obtain a constant round protocol Σ (secure against a minority of malicious parties). Our protocol is based on computational assumptions; the existence of NIZK (or equivalently, two round zaps [DN00]) to be precise. We note that it is easy to rule out information theoretically secure protocols in our setting very similar to how Barak et al [BGGL01] ruled out resettably-sound zero-knowledge *proofs*. The basic idea is that since an honest party has only a bounded size secret information (i.e., the input and the random tape), an unbounded dishonest party can interact with it several times (by resetting it each time) so as to "almost" learn its input/output behavior (and hence an honest party input "consistent" with that behavior). More details will be provided in the full version.

[2] As before, this assumption is only made for simplicity of exposition.

Let $\mathcal{M}$ be the list of malicious parties. Denote the list of honest parties $\mathcal{H} = \{P_1, \ldots, P_n\} - \mathcal{M}$. We prove the following theorem in appendix D,

Theorem 3 (Security Against a Minority of Malicious Parties) *The compiled protocol Σ is secure as per definition 2.2 against the coalition of malicious parties represented by $\mathcal{M}$ as long as $|\mathcal{M}| < n/2$.*

References

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS*, pages 106–115, 2001.

[BGGL01] Boaz Barak, Oded Goldreich, Shafi Goldwasser, and Yehuda Lindell. Resettably-sound zero-knowledge and its applications. In *FOCS*, pages 116–125, 2001.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513. ACM, 1990.

[Can00] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 13(1):143–202, 2000.

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

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

[CGS08] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for uc secure computation using tamper-proof hardware. EUROCRYPT, 2008.

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

[DL07] Yi Deng and Dongdai Lin. Instance-dependent verifiable random functions and their application to simultaneous resettability. In Naor [Nao07], pages 148–168.

[DN00] Cynthia Dwork and Moni Naor. Zaps and their applications. In *FOCS*, pages 283–293, 2000.

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

[GK96] Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for np. *J. Cryptology*, 9(3):167–190, 1996.

[GL02] Shafi Goldwasser and Yehuda Lindell. Secure computation without agreement. In Dahlia Malkhi, editor, *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2002.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC '87: Proceedings of the 19th annual ACM conference on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

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

[HR06] Shai Halevi and Tal Rabin, editors. *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*. Springer, 2006.

[Kat] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *Eurocrypt 2007*.

[KP01] Joe Kilian and Erez Petrank. Concurrent and resettable zero-knowledge in poly-loalgorithm rounds. In *STOC*, pages 560–569, 2001.

[Lin03] Yehuda Lindell. Bounded-concurrent secure two-party computation without setup assumptions. In *STOC*, pages 683–692. ACM, 2003.

[Lin04] Yehuda Lindell. Lower bounds for concurrent self composition. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 203–222. Springer, 2004.

[MOSV06] Daniele Micciancio, Shien Jin Ong, Amit Sahai, and Salil P. Vadhan. Concurrent zero knowledge without complexity assumptions. In Halevi and Rabin [HR06], pages 1–20.

[MP06] Silvio Micali and Rafael Pass. Local zero knowledge. In Jon M. Kleinberg, editor, *STOC*, pages 306–315. ACM, 2006.

[Nao91] Moni Naor. Bit commitment using pseudorandomness. *J. Cryptology*, 4(2):151–158, 1991.

[Nao07] Moni Naor, editor. *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*. Springer, 2007.

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

[RK99] Ransom Richardson and Joe Kilian. On the concurrent composition of zero-knowledge proofs. In *EUROCRYPT*, pages 415–431, 1999.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *FOCS*, pages 543–553, 1999.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.

[YZ07] Moti Yung and Yunlei Zhao. Generic and practical resettable zero-knowledge in the bare public-key model. In Naor [Nao07], pages 129–147.

A Building Blocks

A.1 Non-interactive Perfectly Binding Commitment Scheme with a Unique Decommitment

In our protocol, we shall use a non-interactive perfectly binding commitment scheme with the properties that every commitment has a unique decommitment and the verification of the decommitment is deterministic. An example of such a scheme is the scheme that commits to the bit b by

$com(b;(r,x)) = r||\pi(x)||(x \cdot r) \oplus b$ where π is a one-way permutation on the domain $\{0,1\}^k$, $x \cdot y$ denotes the inner-product of x and y over $GF(2)$, and $x, r \leftarrow U_k$. We denote this commitment scheme by COM.

A.2 Computational Zero-knowledge Proofs and Proofs of Knowledge for all of *NP*

We shall use a computational zero-knowledge proof (ZK proof) for every language in **NP** with negligible soundness error and perfect completeness. Such proof systems can be constructed out of any one way function. One way to construct them is to create statistically binding commitments based on a OWF [HILL99, Nao91]. These commitments can then be used in the 3-colorability protocol of [GMW91] to give us a ZK proof for any language in **NP**. We can then repeat the protocol κ^2 times to achieve negligible soundness error. The protocol obtained will also have perfect completeness. Note that it is also possible to obtain constant round protocols using techniques based on parallel repetitions [GK96] (assuming collision resistant hash functions). We denote the final protocol with negligible soundness error as (P, V). We will also need a (computational) ZK *proof of knowledge* protocol for all of **NP** which can be constructed using similar techniques. We denote the ZK proof of knowledge protocol by (P_{pok}, V_{pok}).

A.3 Two Round Zaps

Zaps are two round public coin witness indistinguishable proofs introduced by Dwork and Naor [DN00]. Zaps further have the special property that the first message (sent by the prover) can be reused for multiple proofs. We denote an algorithm for sampling the first message (of the verifier) by $f_{zaps}(\kappa)$.

A.4 Resettable Sound Zero-Knowledge Arguments

Resettable sound zero-knowledge (rsZK) arguments were introduced by Barak et al [BGGL01]. As in resettable zero-knowledge [CGGM00], rsZK arguments deal with the zero-knowledge functionality but consider the setting when the *verifier* is resettable by the prover. Barak et al [BGGL01] gave a construction of rsZK arguments relying on the non-black techniques introduced by Barak [Bar01]. They also ruled out rsZK arguments having a black-box simulator (except for languages in $\mathcal{BPP}$) thus showing that usage of non-black box techniques is inherent. In our constructions, we rely crucially on the fact that rsZK arguments (as defined by [BGGL01]) have the property that soundness holds even if the verifier can use the same random string in multiple zero-knowledge argument executions even for different statements. We denote such an rsZK argument system by (rsP, rsV).

A.5 Preamble from PRS [PRS02]

In this subsection, we describe the preamble from [PRS02] and give its useful properties for our context. We note that [RK99, KP01] also have similar preambles which could be used for our purpose (although with an increase in the number of rounds).

The preamble of the PRS protocol is as follows. Let κ be the security parameter of the system and k be any super-logarithmic function in κ. Let α be the bit string we wish to commit to and γ be length of α. We break α up into two random shares k^2 times. Let these shares be denoted by $\{\alpha^0_{i,\ell}\}^k_{i,\ell=1}$ and $\{\alpha^1_{i,\ell}\}^k_{i,\ell=1}$ with $\alpha^0_{i,\ell} \oplus \alpha^1_{i,\ell} = r$ for every i, ℓ. The verifier will commit to these bits using a statistically binding commitment scheme COM, with fresh randomness each time.[3] The verifier then sends these k^2 commitments to the prover. This is then followed by k iterations where in the ℓth iteration, the prover sends a random k-bit string $b_\ell = b_{1,\ell}, \ldots, b_{k,\ell}$, and the verifier decommits to the commitments $\mathrm{COM}(\alpha^{b_{1,\ell}}_{1,\ell}), \ldots, \mathrm{COM}(\alpha^{b_{k,\ell}}_{k,\ell})$.

The goal of this protocol is to enable the simulator to be able to rewind and find the value α with high probability by following a fixed strategy. Since the verifier commitments are set after the first round,

[3] In the original PRS preamble, statistically hiding commitment schemes are used. However we use statistically binding (and computationally hiding) ones.

once we rewind the verifier, the simulator will have the opportunity to have the verifier open both the α^0 commitment and the α^1 commitment. In the concurrent setting, rewinding a protocol can be difficult since one may rewind past the start of some other protocol in the system as observed by [DNS98]. The remarkable property of this protocol is that there is a fixed rewinding strategy the simulator can use to get the value of α, for every concurrent cheating verifier strategy $\mathbb{V}^*$, with high probability.

We will follow [MOSV06] in formalizing the properties of the PRS preamble we need. Without loss of generality, assume that there are Q concurrent sessions. Recall that k is the number of rounds of the PRS preamble.

We call the simulator for the PRS preamble CEC-Sim. CEC stands for concurrently-extractable commitments. CEC-Sim will have oracle access to $\mathbb{V}^*$ and will get the following inputs.

- Commitments schemes $\mathcal{COM} = COM_1, COM_2, \ldots, COM_Q$, where COM_s is the commitment scheme used for session s.
- Parameters γ, k, n and Q, all given in unary.

We also need the following definitions adapted from [MOSV06]:

Definition 2 (Major Decommitment) *A* major decommitment *is a reveal after the PRS preamble in which $\mathbb{V}^*$ reveals the opening of commitments $\{COM(\sigma^0_{i,\ell})\}^k_{i,\ell=1}$ and $\{COM(\sigma^1_{i,\ell})\}^k_{i,\ell=1}$. The prover $\mathbb{P}$ only accepts the major decommitment if: (a) all these openings are valid openings to the commitments in the transcript, and, (b) there exists σ such that for all i, ℓ, $\sigma^0_{i,\ell} \oplus \sigma^1_{i,\ell} = \sigma$.*

Definition 3 (Valid Commit Phase) *For a transcript T of the commit phase interaction between $\mathbb{P}$ and $\mathbb{V}^*$, let $T[s]$ denote the messages in session s. $T[s]$ is a* valid commit phase transcript *if there exists a major decommitment D such that $P(T[s], D)$= accept.*

Definition 4 (Compatibility) *Message M=$(\sigma, \sigma^0_{i,j}, \sigma^1_{i,j})$ is* compatible *with $T[s]$ if*

1. $\sigma = \sigma^0_{i,j} \oplus \sigma^1_{i,j}$
2. *There exist commitments $COM_s(\sigma^0_{i,j})[s]$ and $COM_s(\sigma^1_{i,j})[s]$ that are part of the transcript of the first message of $T[s]$.*

Observe that if a message M=$(\sigma, \sigma^0_{i,j}, \sigma^1_{i,j})$ is compatible with the transcript $T[s]$, the cheating verifier can major-decommit to a message different from σ only with probability at most b_{com}. Thus we call σ the *extracted message.*

Definition 5 *A Simulator $CEC - Sim^{\mathbb{V}^*}$ has the* concurrent extraction *property if for every interaction T it has with $\mathbb{V}^*$, it also provides (on a separate output tape) an array of messages $(M_1, M_2, \ldots, M_Q)$ with the following property:*

For every session $s \in \{1, 2, \ldots, Q\}$, if $T[s]$ is a valid commit phase transcript, then M_s is compatible with $T[s]$.

A simulator that has the concurrently extractable property is also called a *concurrently-extractable simulator.*

Using the simulation and rewinding techniques in [PRS02], we can obtain a concurrently-extractable simulator for the PRS preamble. Let $\langle \mathbb{P}, \mathbb{V}^* \rangle$ denote the output of $\mathbb{V}^*$ after concurrently interacting with $\mathbb{P}$.

Lemma 1 *(implicit in [PRS02], adapted from [MOSV06]). There exists a PPT concurrently-extractable simulator CEC-Sim with a fixed strategy SIMULATE such that for $\mathcal{COM}$ and all concurrent adversaries $\mathbb{V}^*$, for settings of parameters σ=poly(n), $k = \tilde{O}(\log n)$, and Q =poly(n), we have the ensembles*

$$\left\{ \textit{CEC-Sim}^{\mathbb{V}^*}(\mathcal{COM}, 1^\sigma, 1^k, 1^n, 1^Q) \right\}_{n \in \mathbb{N}} \text{ and } \left\{ \langle \mathbb{P}, \mathbb{V}^* \rangle (\mathcal{COM}, 1^\sigma, 1^k, 1^n, 1^Q) \right\}_{n \in \mathbb{N}}$$

have statistical difference ϵ, where ϵ is negligible.

B Simultaneous Resettable Multi-Party Computation Model

We consider a system of n parties $P_1, \ldots, P_n$ who interact with each other trying to compute a joint function of their inputs. Each party could have various incarnations where the ith incarnation of a party P_j is defined by the input and random tape tuple (x_j^i, ω_j^i). We denote by X_i the full input and random tape vector of P_i and let $\vec{X} = (X_1, \ldots, X_n)$. Before the protocol begins, we assume that all parties have agreed upon which incarnation will be used by which party. We consider an adversary $\mathcal{A}$ controlling a minority of parties. At any point during the protocol execution, $\mathcal{A}$ can reset any number of honest parties.

As in the two party case, we formalize the above requirement by defining an ideal and a real model. We consider static adversaries (corrupting a minority of the parties) and computational security. We do not consider the issue of output delivery and only guarantee security with abort. This is because the adversary is anyway allowed to reset the honest parties (which in particular means that even after the protocol execution is complete, he can reset the honest parties back to their initial state). For the simplicity of exposition, we assume that all the n parties in the protocol share a *broadcast channel* among them. Since we do not consider the issue of output delivery, such a broadcast channel can actually be implemented by secure point to point channels (which in turn can be implemented, e.g., by shared keys or a PKI) in case of honest majority. See [GL02] for more discussions on this issue.

IDEAL MODEL. In the ideal model there is a trusted party which computes the desired functionality based on the inputs handed to it by the players. Let $\mathcal{M} \subset \{P_1, \ldots, P_n\}$ denote the set of players malicious by the adversary. Then an execution in the ideal model proceeds as follows:

Select incarnation The adversary $\mathcal{A}$ sends an index vector to the trusted party specifying the incarnation for each of the honest parties. The trusted party passes on this information to the honest parties.

Inputs Each party P_i has input x_i. We represent the vector of inputs by $\vec{x}$.

Send inputs to trusted party Honest parties always send their inputs to the trusted party. Corrupted parties, on the other hand, may decide to send modified values to the trusted party.

Trusted party computes the result The trusted party sets the result to be $f(\vec{x})$.

Trusted party sends results to adversary The trusted party sends the result $f(\vec{x})$ to party P_i for all $P_i \in \mathcal{M}$.

Trusted party sends results to honest players The adversary, depending on its view up to this point, prepares a (possibly empty) list of honest parties which should get the output. The trusted party sends the result to the parties in that list and sends $\perp$ to others.

Reset ideal world at any point The adversary $\mathcal{A}$, during the execution of any of the above steps, can send the signal *reset* to the trusted party. In that case, the trust party sends *reset* to all the honest parties and the ideal world comes back to the *select incarnation* stage.

Outputs An honest party P_i always outputs the response it received from the trusted party. Malicious parties output $\perp$, by convention. The adversary outputs an arbitrary function of its entire view (which includes the view of all malicious parties) throughout the execution of the protocol.

For a given adversary $\mathcal{A}$, the *execution of f in the ideal model* on $\vec{X}$ is defined as the output of the honest parties along with the output of the adversary resulting from the process above. It is denoted by $\text{IDEAL}_{f,\mathcal{A}}(\vec{X})$.

REAL MODEL. Honest parties follow all instructions of the prescribed protocol, while malicious parties are coordinated by a single adversary and may behave arbitrarily. The adversary can *reset* any number

of honest parties at any point during the protocol execution. After getting reset, an honest party comes back to its original state. At the conclusion of the protocol, honest parties compute their output as prescribed by the protocol, while the malicious parties output $\perp$. Without loss of generality, we assume the adversary outputs exactly its entire view of the execution of the protocol.

For a given adversary $\mathcal{B}$ and protocol Σ for resettably computing f, the *execution of* Σ *in the real model* on $\vec{X}$ (denoted $\text{REAL}_{\Sigma,\mathcal{B}}(\vec{X})$ is defined as the output of the honest parties along with the output of the adversary resulting from the above process.

Having defined these models, we now define what is meant by a secure protocol.

Definition 6 *Let f and Σ be as above. Protocol Σ is a* t-secure protocol for computing (f, g) *if for every* PPT *adversary $\mathcal{A}$ corrupting at most t players in the real model, there exists an* EPPT *adversary $\mathcal{S}$ corrupting those t players in the ideal model, such that:*

$$\left\{\text{IDEAL}_{f\mathcal{S}}(\vec{X})\right\}_{\vec{X}\in(\{0,1\}^*)^n} \stackrel{c}{\equiv} \left\{\text{REAL}_{\Sigma,\mathcal{A}}(\vec{X})\right\}_{\vec{X}\in(\{0,1\}^*)^n}.$$

C Security Analysis of the Resettable Two-Party Computation Protocol

Theorem 1 (Security Against a Malicious $\mathbb{U}^*$) The compiled protocol Σ is secure against a malicious $\mathbb{U}^*$ as per the definition in Section 2.1.

PROOF. Given S_Π, the simulator for the underlying protocol Π, we show how to construct a simulator Sim_1 for the protocol Σ. Sim_1 will interact with the adversary $\mathbb{U}^*$ (on behalf of $\mathbb{S}$) and simulate its view. Sim_1 will output a simulated transcript from a distribution which is computationally indistinguishable from the distribution of the transcript of a real interaction. The simulator Sim_1 is described below.

Description of the Simulator Sim_1. The simulator Sim_1 consists of two parts: a wrapper S_{wrap} and a core simulator S_{core}. S_{wrap} acts as a "wrapper" around the malicious $\mathbb{U}^*$ and handles all its communication as well as reset requests. S_{wrap} in turn communicates with the core simulator S_{core} but is not allowed to reset S_{core}. Very informally, the function of S_{wrap} is to convert a reset attack into a concurrent attack. We first describe the operation of S_{wrap} in detail.

The Wrapper S_{wrap}. The wrapper S_{wrap} runs the malicious $\mathbb{U}^*$ and interacts with it. S_{wrap} in turn interacts with S_{core} in several sessions concurrently and, very roughly, forwards it all the messages received from $\mathbb{U}^*$ except the messages of the resettable sound zero-knowledge (rsZK) argument (and except certain "duplicate messages" as explained later). First we define the *user tuple* for a session as the set of the *determining message* (see section 4.1) for that session and the incarnation index of the smartcard $\mathbb{S}$ which $\mathbb{U}^*$ requested to be activated for the session. Now the number of concurrent sessions between S_{wrap} and S_{core} is equal to the total number of *unique user tuples* that S_{wrap} ever receives from $\mathbb{U}^*$. In more detail, these interactions take place as follows.

- The first message msg that S_{wrap} receives from $\mathbb{U}^*$ (after $\mathbb{U}^*$ requested a particular incarnation e of $\mathbb{S}$ to be activated) is the *determining message* of the session. Now there are two possibilities:

  1. S_{wrap} has received the user tuple (msg, e) for the first time from $\mathbb{U}^*$ for this particular incarnation. In this case, S_{wrap} opens a new session with S_{core} specifying the incarnation e (however S_{wrap} doesn't yet send any messages in that session). Each concurrent session of S_{wrap} with S_{core} is indexed by its user tuple (msg, e).

2. S_{wrap} received the same user tuple (msg, e) in a session earlier from $\mathbb{U}^*$ (before S_{wrap} was reset by $\mathbb{U}^*$). This means that a session with index (msg, e) is already in progress between S_{wrap} and S_{core}. Hence S_{wrap} does nothing in this case.

   Going forward, for session (msg, e), we categorize every message (which is not a message of the rsZK argument system) received from $\mathbb{U}^*$ as either a *duplicate message* or an *original message*. A duplicate message means that the corresponding message for session (msg, e) was *already forwarded* to S_{core} earlier. This in turn means that the same message was received earlier from $\mathbb{U}^*$ and was *successfully authenticated* (i.e., an obvious cheating attempt was not detected and its associated rsZK argument, if any, was completed successfully).

- The wrapper S_{wrap} and $\mathbb{U}^*$ now execute a resettable sound ZK argument as usual where $\mathbb{U}^*$ proves to S_{wrap} that msg represents a valid PRS commit phase. For all e, S_{wrap} has a fixed random string R^e_{rs} which it uses when acting as a verifier for all the rsZK arguments for sessions where incarnation e was activated. After successful completion of this rsZK argument, if msg is an original message (i.e., was not already forwarded to S_{core}) S_{wrap} forwards it to S_{core}. Else, msg is discarded as a duplicate message.

- During the PRS preamble challenge response phase, S_{wrap} does the following for $\ell = 1, \ldots k$.

  1. If the previous message received from $\mathbb{U}^*$ was a duplicate message, S_{wrap} already had the challenge bits $b_{1,\ell}, \ldots, b_{k,\ell} \stackrel{\$}{\leftarrow} \{0,1\}^k$ stored. Else S_{wrap} received them from S_{core} on forwarding it the previous message. Thus, S_{wrap} forwards these challenge bits to $\mathbb{U}^*$.
  2. S_{wrap} receives the response from $\mathbb{U}^*$ and checks if all the decommitments are valid. If so, it forwards the response to S_{core} if it is an original message (and does nothing otherwise).

- For rest of the protocol (i.e., during the smartcard input commitment and secure computation phases), S_{wrap} behaves as follows.

  1. If the previous message received from $\mathbb{U}^*$ was a duplicate message, S_{wrap} already had the next message to be forwarded to it. Else S_{wrap} received it from S_{core} on forwarding it the previous message. Thus, S_{wrap} forwards the next message to $\mathbb{U}^*$.
  2. S_{wrap} receives the response from $\mathbb{U}^*$ and then, as usual, executes a resettable sound ZK argument with $\mathbb{U}^*$ to ensure the correctness of the received message. After successful completion of the rsZK argument, if the received message is an original message, S_{wrap} forwards it to S_{core} (and does nothing otherwise).

During the execution of the above protocol, whenever S_{wrap} receives a reset request from $\mathbb{U}^*$, it treats it as a "session on hold" request. That is, S_{wrap} just stops the execution of the current session with S_{core} and switches to the execution of another session (depending on the user tuple received from $\mathbb{U}^*$ after the reset request).

If at any point, S_{wrap} receives a message from $\mathbb{U}^*$ such that it is successfully authenticated and a corresponding message *different from this message* was already forwarded to S_{core} earlier, S_{wrap} aborts. S_{wrap} also aborts anytime an obvious cheating attempt is detected from $\mathbb{U}^*$.

At the end of the interaction between S_{wrap} and $\mathbb{U}^*$, if $\mathbb{U}^*$ aborts (without completing the last session), S_{wrap} sends the signal *abort* to S_{core}. In this case, the honest smartcard $\mathbb{S}$ would get $\perp$ as its output in the ideal world. Else, S_{wrap} sends the signal *finish* to S_{core} along with the user tuple (msg_{final}, e_{final}) of the last finished session.

The Core Simulator S_{core}. Simulator S_{core} accepts interaction in various concurrent sessions from S_{wrap}. The strategy of S_{core} is as follows.

- In the PRS preamble phase, S_{core} uses the PRS simulator [PRS02] to extract the values committed to in each of sessions as follows. The wrapper S_{wrap} interacts concurrently with S_{core} in several concurrent sessions. S_{core} runs the concurrently extractable simulator CEC-Sim (see Section A.5) and recovers a value $\beta[i]$ compatible (see definition 4) with the PRS commit phase transcript of session i. CEC-Sim works by creating various *look ahead threads* (see [PRS02]) in addition to the main thread of interaction with S_{wrap}. If CEC-Sim fails (and outputs $\perp$), the core simulator S_{core} simply aborts and outputs $\perp$. The strategy of S_{core} in the rest of the phases (for the main as well the look ahead threads) is described later.

  Now since the PRS commit phase of each of the session is guaranteed to be a valid commit phase (by the soundness of the rsZK arguments accepted by S_{wrap}), we have that $\beta[i](= (x_2[i], r_2[i]))$ is the major decommitment of the PRS commit phase of session i. That is, S_{core} has extracted the input $x_2[i]$ and all the randomness $r_2[i]$ to be used by S_{wrap} in the execution of the rest of the protocol in the ith session.

- In each (i.e., main as well as look ahead) of the threads, for session i, S_{core} now uses the simulator S_Π of the protocol Π (secure only against semi-honest adversaries) to generate a protocol transcript. The simulator S_Π is run on the extracted input $x_2[i]$ (for this particular thread). S_Π starts executing and, at some point, makes a call to the trusted party (in the ideal world) with some input $x'_2[i]$. S_{core} forwards this call to the actual ideal world trusted party (recall that S_{core} is interacting on behalf of the honest $\mathbb{S}$ in the protocol Σ) specifying the appropriate incarnation of $\mathbb{S}$ and then passes on the response to S_Π. However in the ideal world, instead of signaling the trusted party to send the output to the honest $\mathbb{S}$ as well, S_{core} sends the *reset* signal to the trusted party (hence allowing itself to make multiple queries for completing various sessions and various threads). Finally, S_Π halts and outputs a transcript $m_1^1[i], m_2^1[i], \ldots, m_1^t[i], m_2^t[i]$ (of the execution of protocol Π for session i of the current thread) and an associated randomness $r_\Pi[i]$.

  The first task of S_{core} now is to "force" the wrapper S_{wrap} (and hence $\mathbb{U}^*$) to use randomness $r_\Pi[i]$ during execution of the underlying protocol Π later on. To that effect, S_{core} computes a random string $r'_2[i]$ such that $r''_2[i]$ $(=r_2[i] \oplus r'_2[i])$ satisfies these requirements (i.e., using $r''_2[i]$ in rest of Σ would amount to using $r_\Pi[i]$ in Π). S_{core} sends $r'_2[i]$ to S_{wrap} to finish the PRS preamble phase.

- In each of the threads, for session i, S_{core} generates a random $\alpha[i] = (x_1[i], r_1[i])$ and completes the smartcard input commitment phase honestly using it.

- In each of the threads, the core simulator S_{core} runs the secure computation phase as follows. For session i, the goal of S_{core} now is to force the transcript $m_1^1[i], m_2^1[i], \ldots, m_1^t[i], m_2^t[i]$ onto the wrapper S_{wrap} (and hence onto $\mathbb{U}^*$). This is done as follows. In round $j, 1 \leq j \leq t$, S_{core} sends the message $m_1^j[i]$ to S_{core} (instead of sending a message as per the input and randomness committed to in the smartcard input commitment phase earlier). Now S_{core} is required to give a zero-knowledge proof that $m_1^j[i]$ was computed correctly. However S_{core} has already extracted the randomness $r''_2[i]$ which S_{wrap} will use while acting as a verifier of this proof. Hence, S_{core} can generate a simulated transcript for the proof *offline* and then force it onto S_{wrap} using techniques similar to those being used to force the transcript of protocol Π. After completion of the zero-knowledge proof, S_{wrap} forwards its reply to S_{core}. Clearly, the reply of S_{wrap} has to be the message $m_2^j[i]$ with all but negligible probability (by the soundness of the rsZK argument which S_{wrap} accepted from $\mathbb{U}^*$).

- The core simulator S_{core}, having finished all the sessions with S_{wrap}, now looks at the final message received from S_{wrap}. If S_{wrap} sent the *abort* signal, S_{core} sends *abort* to the trusted party in the ideal world, thus causing the output of $\mathbb{S}$ to be $\perp$. Otherwise, S_{core} looks at the received session index $i = msg_{final}$ and retrieves the query $x'_2[i]$ made by the simulator S_Π while generating the protocol transcript for the session i during the main thread. S_{core} now requests the smartcard incarnation e_{final} to be activated in the ideal world and sends input $x'_2[i]$ to the trusted party.

After getting the output, it instructs the trusted party to send the output also to $\mathbb{S}$ thus completing the ideal world execution.

This completes the description of the simulator Sim_1.

Indistinguishability of the Real and Simulated Views. We will consider a series of hybrid experiments and show that the successive hybrids are indistinguishable from each other. Our initial experiment will be the actual protocol as executed by $\mathbb{U}^*$ and $\mathbb{S}$. Our final experiment will be the simulated protocol as described above. Below, we only give a sketch of our hybrid experiments.

Experiment H_0. This experiment corresponds to when the simulator Sim_1 has the required input and randomness vector X_1 (i.e., x_1^e and (G^e, R_{rs}^e) for incarnation e) and runs the protocol honestly with the malicious user $\mathbb{U}^*$.

Experiment H_1. In this experiment, the simulator Sim_1 is broken into a core simulator S_{core} (having input x_1^e and G^e for all e) and a wrapper S_{wrap} (having input R_{rs}^e for all e). The wrapper S_{wrap} behaves as described above (see the description of the simulator Sim_1). However the core simulator S_{core} handles the concurrent sessions honestly and computes all its replie.s using x_1^e and G^e as per the protocol specifications. The only difference between H_0 and H_1 is that now the interaction with $\mathbb{U}^*$ is aborted if at any point, S_{wrap} receives a message from $\mathbb{U}^*$ such that it is successfully authenticated and a corresponding message *different from this message* was already forwarded to S_{core} earlier. By the soundness of the rsZK arguments (see Section A.4), it can be shown that the probability of this event happening is negligible. Thus, we have that the view of $\mathbb{U}^*$ in experiment H_1 is indistinguishable from the view in H_0.

Experiment H_2. This experiment is defined exactly as the previous one with the exception that now S_{core} discards the input G^e for all e and instead uses freshly generated random tape in all the concurrent sessions. Consider incarnation e. Assuming there were ℓ concurrent sessions with (distinct) determining messages $msg_1, \ldots, msg_\ell$, we are essentially replacing the string $G^e(msg_1), \ldots, G^e(msg_\ell)$ with a uniformly chosen string. Both the strings are computationally indistinguishable by the property of the pseudorandom function G^e. Thus, we have that the view of $\mathbb{U}^*$ in experiment H_2 is indistinguishable from the view in H_1.

Experiment H_3. This experiment is defined exactly as the previous one with the exception that now S_{core} uses the concurrently extracting simulator CEC-Sim to extract the values committed to by S_{wrap} in the PRS commit phase in each of sessions as explained before. Thus for each of the threads, for session i, S_{core} has the committed value $\beta[i] = (x_2[i], r_2[i])$. After finishing the PRS commit phase, for each of the sessions, S_{core} runs the rest of the protocol as in Experiment H_2 (in the main as well as the look ahead threads). Note that if CEC-Sim fails (and outputs $\perp$), S_{core} simply aborts. Lemma 1 states that the view of $\mathbb{U}^*$ in this experiment (with simulated PRS commit phase) remains indistinguishable from the one in experiment H_2.

Experiment H_4. This experiment is defined exactly as the previous one with the exception that now the simulator S_{core} starts simulating the ZKPOK being given in the smartcard input commitment phase (in all the sessions of all the threads). S_{core} uses the simulator S_{pok} of the system $(P_{pok}, V_{pok}$ and generates a simulated transcript *offline* (since it has already extracted the randomness of the verifier S_{wrap}). S_{core} then forces this simulated transcript onto S_{wrap}. By the zero-knowledge property of the proof system $(P_{pok}, V_{pok}$ (and security of the rsZK argument system), it follows that the view of $\mathbb{U}^*$ in experiment H_4 is indistinguishable from the view in H_3.

Experiment H_5. This experiment is defined exactly as the previous one with the exception that now in each of the threads, for session i, the simulator S_{core} generates a random $\alpha[i] = (x_1[i], r_1[i])$ and commits to it in the smartcard input commitment phase (as opposed to committing the real input and random tape which it is using to complete the protocol). Since this commitment is not being used anywhere else in the protocol, by the computationally hiding property of the commitment scheme COM, the view of $\mathbb{U}^*$ in experiment H_5 is indistinguishable from the view in H_4.

Experiment H_6. This experiment is defined exactly as the previous one with the exception that now the simulator S_{core} starts giving a real ZKPOK of the committed $\alpha[i]$ in the smartcard input commitment phase (in all the sessions of all the threads) as opposed to a simulated one. Again by the zero-knowledge property of the proof system $(P_{pok}, V_{pok}$, it follows that the view of $\mathbb{U}^*$ in experiment H_6 is indistinguishable from the view in H_5. Note that at this point, the smartcard input commitment phase is essentially "disconnected" and independent of the rest of the protocol. However the simulator S_{core} is still using the real input during the secure computation phase.

Experiment H_7. In this experiment, in each of the threads, for session i, S_{core} now uses the simulator S_Π of the protocol Π to generate a protocol transcript. All the calls to the trusted party in the ideal world are made (as required by S_Π) and then a *reset* signal is sent by S_{core}. Furthermore, after finishing all the interactions, S_{core} also either sends the *abort* signal to the trusted party or completes the ideal world interaction with the right (final) input as appropriate. Now S_Π halts and outputs a transcript $m_1^1[i], m_2^1[i], \ldots, m_1^t[i], m_2^t[i]$ and an associated randomness $r_\Pi[i]$. S_{core} now computes a random string $r_2'[i]$ and defines $r_2''[i]$ $(=r_2[i] \oplus r_2'[i])$. $r_2'[i]$ is computed such that using $r_2''[i]$ in rest of Σ would amount to using $r_\Pi[i]$ in Π. Now this experiment is defined exactly as the previous one with the exception that now the simulator S_{core} sends the string $r_2'[i]$ to S_{wrap} to finish the PRS preamble phase. Clearly, the view of $\mathbb{U}^*$ in experiment H_7 is identical to the view in H_6.

Experiment H_8. This experiment is defined exactly as the previous one with the exception that now the simulator S_{core} stops using the actual input during the secure computation phase and instead forces the generated $m_1^1[i], m_2^1[i], \ldots, m_1^t[i], m_2^t[i]$ transcript onto S_{wrap}. As described below (see the description of the simulator), this is done by sending the next message as usual and forcing a simulated transcript of the associated zero-knowledge proof onto S_{wrap}. The security of the underlying protocol Π and that of the proof system (P, V) implies that the forced transcript is indistinguishable from the real transcript. Thus it follows that the view of $\mathbb{U}^*$ in experiment H_8 is indistinguishable from the view in H_7.

Note that the simulator in the experiment H_8 is our actual simulator Sim_1. Thus, the output of the simulator Sim_1 is computationally indistinguishable from the distribution of the transcript of a real interaction. This completes our proof.

■

Theorem 2 (Security Against a Malicious $\mathbb{S}^*$) The compiled protocol Σ is secure against a malicious $\mathbb{S}^*$ in the sense of the definition in Section 2.1.

PROOF. Given S_Π, the simulator for the underlying protocol Π, we show how to construct a simulator Sim_2 for the protocol Σ. Sim_2 will interact with a given incarnation of the adversary $\mathbb{S}^*$ (on behalf of $\mathbb{U}$) in a standard standalone setting and simulate its view. Sim_2 will output a simulated transcript from a distribution which is computationally indistinguishable from the distribution of the transcript of a real interaction. The simulator Sim_2 is described below.

Description of the Simulator Sim_2. The strategy of the simulator Sim_2 is as follows.

- Throughout the protocol Σ, whenever Sim_2 is required to participate as a prover rsP in a resettable sound zero-knowledge argument to prove a given statement, Sim_2 uses the simulator rsS of the rsZK argument system (rsP, rsV) to give a simulated proof to the malicious smartcard $\mathbb{S}^*$. The simulator rsS is a non-black box straightline simulator (see Section A.4) which makes use of the code of $\mathbb{S}^*$ and produces a view which, to $\mathbb{S}^*$, is computationally indistinguishable from the view when $\mathbb{U}$ was giving an honestly generated proof. If, for any of the proofs, the simulator rsS fails and outputs $\perp$, Sim_2 aborts the whole simulation and outputs $\perp$ as well.

- In the PRS preamble phase, Sim_2 generates a string $\beta = (x_2, r_2)$ randomly. The simulator Sim_2 then runs the PRS preamble phase honestly using β (except for the fact that Sim_2 now gives a simulated rsZK argument as noted earlier).

- In the smartcard input commitment phase, Sim_2 now uses the knowledge extractor E_{pok} (see Section A.2) of the ZKPOK system (P_{pok}, V_{pok}) to extract the string $\alpha = (x_1, r_1)$ committed to by the smartcard $\mathbb{S}^*$. The extractor E_{pok} works simply by rewinding $\mathbb{S}^*$ (while Sim_2 continues to give simulated rsZK arguments in all the resulting threads). If the extractor E_{pok} fails and outputs $\perp$, Sim_2 aborts the whole simulation and outputs $\perp$ as well.

- The simulator Sim_2 now uses the simulator S_Π of the protocol Π (secure only against semi-honest adversaries) to generate a protocol transcript. The simulator S_Π is run on the extracted input x_1 of the smartcard $\mathbb{S}^*$. S_Π starts executing and, at some point, makes a call to the trusted party (in the ideal world) with some input x_1'. Sim_2 forwards this call to the actual ideal world trusted party (recall that Sim_2 is interacting on behalf of the honest $\mathbb{U}$ in the protocol Σ) and passes on the response to S_Π. However in the ideal world, Sim_2 does not yet signal the trusted party to send the output to the honest $\mathbb{U}$. Finally, S_Π halts and outputs a transcript $m_1^1, m_2^1, \ldots, m_1^t, m_2^t$ (of the execution of protocol Π) and an associated randomness r_Π.

  The first task of Sim_2 now is to "force" $\mathbb{S}^*$ to use randomness r_Π during execution of the underlying protocol Π later on. To that effect, Sim_2 computes a random string r_1' such that r_1'' $(=r_1 \oplus r_1')$ satisfies these requirements (i.e., using r_1'' would amount to using r_Π in Π). Sim_2 sends r_1' to $\mathbb{S}^*$ (and gives a simulated rsZK argument) to finish the smartcard input commitment phase.

- The simulator Sim_2 runs the secure computation phase and forces the generated protocol transcript onto $\mathbb{S}$. In round j, clearly the message sent by $\mathbb{S}^*$ has to be m_1^j with all but negligible probability. This is because of the associated zero-knowledge proof where $\mathbb{S}^*$ acts as the prover P and proves to Sim_2 (acting as the verifier V) the consistency of the sent message with the committed input x_1 and random tape r_1''. Sim_2 uses a fresh random tape while emulating the verifier V (and simulates the rsZK argument associated with every verifier message). In reply to the smartcard message m_1^j, Sim_2 sends the message m_2^j from the generated protocol transcript (as opposed to sending a message using the input and random tape committed to in the PRS preamble phase) and simulates the associated rsZK argument.

- If the above simulation is completed successfully and neither party aborts, Sim_2 instructs the trusted party in the ideal world to send the output to the honest user $\mathbb{U}$. Else, Sim_2 sends *abort* to the trusted party.

It can be shown that the above simulated transcript comes from a distribution which is computationally indistinguishable from the distribution of the transcript of a real interaction. This is done using a sequence of hybrids similar to the ones in the proof of theorem 1. More details will be provided in the full version.

■

D Security Analysis of the Simultaneous Resettable Multi-Party Computation Protocol

Let $\mathcal{M}$ be the list of malicious parties. Denote the list of honest parties $\mathcal{H} = \{P_1, \ldots, P_n\} - \mathcal{M}$.

Theorem 3 (Security Against a Minority of Malicious Parties) The compiled protocol Σ is secure as per definition 2.2 against the coalition of malicious parties represented by $\mathcal{M}$ as long as $|\mathcal{M}| < n/2$.

PROOF. Given S_Π, the simulator for the underlying protocol Π, we show how to construct a simulator Sim for the protocol Σ. Sim will interact with the adversary $\mathcal{A}$ controlling parties represented by the coalition $\mathcal{M}$ (on behalf of the honest parties $\mathcal{H}$) and simulate its view. Sim will output a simulated transcript from a distribution which is computationally indistinguishable from the distribution of the transcript of a real interaction. The simulator Sim is described below.

Description of the Simulator Sim. As in the proof of theorem 1, the simulator Sim consists of two parts: a wrapper S_{wrap} and a core simulator S_{core}. S_{wrap} acts as a "wrapper" around the adversary $\mathcal{A}$ and handles all its communication as well as reset requests. S_{wrap} in turn communicates with the core simulator S_{core} but is not allowed to reset S_{core}. Very informally, the function of S_{wrap} is to convert a reset attack into a concurrent attack. We first describe the operation of S_{wrap} in detail.

The Wrapper S_{wrap}. The wrapper S_{wrap} runs the adversary $\mathcal{A}$ and interacts with it. S_{wrap} in turn interacts with S_{core} in several sessions concurrently and, very roughly, forwards it all the *authenticated* messages received from $\mathcal{A}$ (except certain "duplicate messages" as explained later). We first define an *adversary tuple* for a session as the set consisting of the determining messages of the all the malicious parties and the incarnation indices of all the honest parties which $\mathcal{A}$ requested to be activated for that session. Now the number of concurrent sessions between S_{wrap} and S_{core} is equal to the number of *unique adversary tuples* that S_{wrap} ever receives from $\mathcal{A}$. For the purpose of this proof, we assume w.l.g. that in a give round, first all the honest parties broadcast their messages and only then the adversarial parties broadcast their messages (i.e., we allow the adversary to be rushing).

In more detail, S_{wrap} works as follows. In the very beginning, S_{wrap} receives from S_{core} the determining message for all incarnation of all honest parties. Now assume that the selection of adversary $\mathcal{A}$ of the incarnation of honest parties for a session be denoted by E (where E is the set consisting of incarnation indices).

- The wrapper S_{wrap} sends to $\mathcal{A}$ the honest party determining message set S_h consisting of appropriate determining messages as per the incarnation index set E. Further, it receives the malicious party determining message set S_m. Now there are two possibilities:

  1. S_{wrap} has received the adversary tuple (S_m, E) for the first time from $\mathcal{A}$. In this case, S_{wrap} opens a new session with S_{core} specifying the incarnation index set E. Each concurrent session of S_{wrap} with S_{core} is indexed by its adversary tuple (S_m, E). In this session, S_{wrap} receives the honest party determining message set S_h and sends the malicious party determining message set S_m to S_{core}. s
  2. S_{wrap} received the same tuple (S_m, E) in a session earlier from $\mathcal{A}$ (before S_{wrap} was reset by $\mathcal{A}$, more on reset attacks later). This means that a session with index (S_m, E) is already in progress between S_{wrap} and S_{core}. Hence S_{wrap} does nothing in this case.

     Going forward, for session (S_m, E), we categorize every message (which is not a message of the 1-round ZKAOK system) received from $\mathcal{A}$ as either a *duplicate message* or an *original message*. A duplicate message means that the corresponding message for session (S_m, E) was *already forwarded* to S_{core} earlier. This in turn means that the same message was received

earlier from $\mathcal{A}$ and was *successfully authenticated* (i.e., an obvious cheating attempt was not detected and its associated 1-round ZKAOK was completed successfully).

- For rest of the protocol (i.e., during the coin flipping and secure computation phases), S_{wrap} behaves as follows.

  1. If the previous message received from $\mathcal{A}$ (consisting of the messages of all the malicious parties in that round) was a duplicate message, S_{wrap} already had the next message (i.e., messages of the honest parties for the next round) to be forwarded to it. Else S_{wrap} received it from S_{core} on forwarding it the previous message. Thus, S_{wrap} forwards the next message to $\mathcal{A}$.
  2. S_{wrap} receives the response from $\mathcal{A}$ and verifies the 1-round ZKAOK messages contained in it. The randomness used by S_{wrap} to verify the ZKAOK messages meant for the eth incarnation of party P_i is $R^e_{i,zkv}$. After successful verification of all the ZKAOK messages thus proving correctness of the received message, if the received message is an original message, S_{wrap} forwards it to S_{core}. Furthermore, S_{wrap} *also* forwards the ZKAOK messages (which it verified) associated with the received message. However if the received message is a duplicate message, S_{wrap} neither forwards the received message nor the associated ZKAOK (even if the associated ZKAOK is different from the one forwarded earlier).

During the execution of the above protocol, w.l.g., we assume that S_{wrap} only resets all the honest parties together. This is because if S_{wrap} resets only a (strict) subset of the honest parties, the honest parties will detect that they are "out of sync" with each other (in the next round) and hence will all abort. This means that the effect of $\mathcal{A}$ resetting a strict subset of honest parties is the same as that when $\mathcal{A}$ aborts the protocol. Now whenever S_{wrap} receives a reset request (to reset all honest parties) from $\mathcal{A}$, it treats it as a "session on hold" request. That is, S_{wrap} just stops the execution of the current session with S_{core} and switches to the execution of another session (depending on the adversary tuple received from $\mathcal{A}$ after the reset request).

If at any point, S_{wrap} receives a message from $\mathcal{A}$ such that it is successfully authenticated and a corresponding message *different from this message* was already forwarded to S_{core} earlier, S_{wrap} aborts. S_{wrap} also aborts anytime an obvious cheating attempt is detected from $\mathcal{A}$.

At the end of the interaction between S_{wrap} and $\mathcal{A}$, if $\mathcal{A}$ aborts (without completing the final session), S_{wrap} sends the signal *abort* to S_{core}. In this case, the honest parties would get $\perp$ as their output in the ideal world. Else, S_{wrap} sends the signal *finish* to S_{core} along with the adversary tuple $(S_{m,final}, E_{final})$ of the last finished session.

The Core Simulator S_{core}. Simulator S_{core} accepts interaction in various concurrent sessions from S_{wrap}. The strategy of S_{core} is as follows.

- For every incarnation of every honest party, S_{core} sets the input to be a random string (of appropriate size) and computes and stores the determining message. Now for each the sessions, S_{core} completes the input commitment phase with S_{wrap} otherwise honestly using these determining messages. Let the determining message of party P_i in the input commitment phase be $A_i, Z_i, PK_i, b^1_i, \ldots, b^n_i$.

- The first round of the coin flipping phase for each session s is executed as follows. S_{core} sends commitments to *random strings* for each honest party (as opposed to computing the string using any committed random tape) to S_{wrap}. Furthermore, for every honest party P_i and every malicious party P_j, S_{core} computes a *simulated* 1-round ZKAOK of the correctness of the commitment and sends it to S_{wrap}. Such a simulated proof is computed using the preimages of a majority of strings from $b^i_1, \ldots, b^i_n$ as the witness. Since S_{core} decided a majority of determining messages for the session, it must have the required preimages. After receiving the malicious party messages for this

round in reply from S_{wrap}, for every session s and every malicious party P_j, S_{core} *extracts* the input $\alpha_j[s](= (x_j[s], G_j[s]))$ committed by P_j as part of the determining message. Note that S_{core} will be able to extract $\alpha_j[s]$ since: (a) It has a majority of the secret keys corresponding to the public keys contained in the determining messages, and, (b) It can be shown that the ZKAOK of malicious party P_j has to be computed by using a witness of the actual statement as the witness for computing the prover message of the zap system. This is because since if a majority of the preimages of the strings $b_1^j, \ldots, b_n^j$ is used as a witness, it implies an efficient inverting algorithm for the OWF F. At the end of the first round, S_{core} has extracted $\alpha_j[s]$ and hence $R_j[s]$ for each malicious party P_j for session s.

- In each session s, S_{core} now uses the simulator S_Π of the protocol Π (secure only against semi-honest adversaries) to generate a protocol transcript. The simulator S_Π is run on the extracted input set consisting of $x_j[s]$ for all malicious P_j's. S_Π starts executing and, at some point, makes a call to the trusted party (in the ideal world) with some input set consisting of $x'_j[s]$ for all malicious P_j's. S_{core} forwards this call to the actual ideal world trusted party (recall that S_{core} is interacting on behalf of the honest parties in the protocol Σ) specifying the appropriate incarnation honest parties and then passes on the response to S_Π. However in the ideal world, instead of signaling the trusted party to send the output to the honest parties as well, S_{core} sends the *reset* signal to the trusted party (hence allowing itself to make multiple queries for completing various sessions). Finally, S_Π halts and outputs a transcript $(m_1^1[s], \ldots, m_n^1[s]), \ldots, (m_1^t[s], \ldots, m_n^t[s])$ (of the execution of protocol Π for session s) and an associated randomness set consisting of $r''_j[s]$ for each malicious P_j.

  The first task of S_{core} now is to "force" each malicious P_j to use randomness $r''_j[s]$ during execution of the underlying protocol Π later on. To that effect, S_{core} computes a string $R'_i[s]$ for each honest P_i such that for each malicious P_j, $r''_j[s] = r_j[s] \oplus r'_j[s]$ where $r'_1[s]|| \ldots ||r'_n[s] = R'_1[s] \oplus \cdots \oplus R'_n[s]$ and $r_j[s]$ comes from the extracted $R_j[s]$ as per the protocol. To complete the coin flipping phase, S_{core} sends to S_{wrap} the computed strings $R'_i[s]$ for each honest P_i. S_{core} also sends a simulated ZKAOK to every honest party P_i and every malicious party P_j to prove the correctness of each $R'_i[s]$. S_{wrap} must reply with the right committed string $R'_j[s]$ for each malicious P_j (by the soundness of the ZKAOK).

- The core simulator S_{core} runs the secure computation phase as follows. For session i, the goal of S_{core} now is to force the transcript $(m_1^1[s], \ldots, m_n^1[s]), \ldots, (m_1^t[s], \ldots, m_n^t[s])$ onto the wrapper S_{wrap} (and hence onto $\mathcal{A}$). This is done as follows. In round $k, 1 \leq k \leq t$, S_{core} sends the message set consisting of $m_i^k[s]$ for each honest P_i to S_{core} (instead of sending a message as per the input and randomness committed to in the input commitment phase earlier). In addition, S_{core} si.mulates all the 1-round ZKAOK required to prove correctness of the sent messages. Now S_{wrap} forwards its reply to S_{core}. Clearly, the reply of S_{wrap} has to be the message set consisting of $m_j^k[s]$ with all but negligible probability (by the soundness of the ZKAOK system).

- The core simulator S_{core}, having finished all the sessions with S_{wrap}, now looks at the final message received from S_{wrap}. If S_{wrap} sent the *abort* signal, S_{core} sends *abort* to the trusted party in the ideal world, thus causing the output of honest to be $\perp$. Otherwise, S_{core} looks at the received session index $s = (S_{m,final}, E_{final})$ and retrieves the query consisting of $x'_j[s]$ for all $P_j \in \mathcal{M}$ made by the simulator S_Π while generating the protocol transcript for the session s. S_{core} now requests the honest party incarnations as per E_{final} to be activated in the ideal world and sends input set $x'_j[s]$ to the trusted party. After getting the output, it instructs the trusted party to send the output also to honest parties thus completing the ideal world execution.

This completes the description of the simulator Sim. Using a standard hybrid argument, it can be shown that the above simulated transcript comes from a distribution which is computationally indistinguishable from the distribution of the transcript of a real interaction. More details will be provided in the full version.

■