Revocation in Publicly Verifiable Outsourced Computation

James Alderman*, Christian Janson, Carlos Cid**, and Jason Crampton

Information Security Group, Royal Holloway, University of London
Egham, Surrey, TW20 0EX, United Kingdom
{James.Alderman.2011, Christian.Janson.2012}@live.rhul.ac.uk
{Carlos.Cid, Jason.Crampton}@rhul.ac.uk

Abstract. The combination of software-as-a-service and the increasing use of mobile devices gives rise to a considerable difference in computational power between servers and clients. Thus, there is a desire for clients to outsource the evaluation of complex functions to an external server. Servers providing such a service may be rewarded per computation, and as such have an incentive to cheat by returning garbage rather than devoting resources and time to compute a valid result.
In this work, we introduce the notion of Revocable Publicly Verifiable Computation (RPVC), where a cheating server is revoked and may not perform future computations (thus incurring a financial penalty). We introduce a Key Distribution Center (KDC) to efficiently handle the generation and distribution of the keys required to support RPVC. The KDC is an authority over entities in the system and enables revocation. We also introduce a notion of blind verification such that results are verifiable (and hence servers can be rewarded or punished) without learning the value. We present a rigorous definitional framework, define a number of new security models and present a construction of such a scheme built upon Key-Policy Attribute-based Encryption.

Keywords— Publicly Verifiable Outsourced Computation, Key Distribution Center, Key-policy Attribute-based Encryption, Revocation

1 Introduction

It is increasingly common for mobile devices to be used as general computing devices. There is also a trend towards cloud computing and enormous volumes of

* The first author acknowledges support from BAE Systems Advanced Technology Centre under a CASE Award.

** This research was partially sponsored by US Army Research laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defence, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

data ("big data") which means that computations may require considerable computing resources. In short, there is a growing discrepancy between the computing resources of end-user devices and the resources required to perform complex computations on large datasets. This discrepancy, coupled with the increasing use of software-as-a-service, means there is a requirement for a client device to be able to delegate a computation to a server.

Consider, for example, a company that operates a "bring your own device" policy, enabling employees to use personal smartphones and tablets for work. Due to resource limitations, it may not be possible for these devices to perform complex computations locally. Instead, a computation is outsourced over some network to a more powerful server (possibly outside the company, offering software-as-a-service, and hence untrusted) and the result of the computation is returned to the client device. Another example arises in the context of battlefield communications where each member of a squadron of soldiers is deployed with a reasonably light-weight computing device. The soldiers gather data from their surroundings and send it to regional servers for analysis before receiving tactical commands based on results. Those servers may not be fully trusted e.g. if the soldiers are part of a coalition network. Thus a soldier must have an assurance that the command has been computed correctly. A final example could consider sensor networks where lightweight sensors transmit readings to a more powerful base station to compute statistics that can be verified by an experimenter.

In simple terms, given a function F to be computed by a server S, the client sends input x to S, who should return $F(x)$ to the client. However, there may be an incentive for the server (or an imposter) to cheat and return an invalid result $y \neq F(x)$ to the client. The server may wish to convince a client of an incorrect result, or (particularly if servers are rewarded per computation performed) the server may be too busy or may not wish to devote resources to perform the computation. Thus, the client wishes to have some assurance that the result y returned by the server is, in fact, $F(x)$. This problem, known as *Verifiable Outsourced Computation* (VC), has attracted a lot of attention in the community recently. In practical scenarios, it may well be desirable that cheating servers are prevented from performing future computations, as they are deemed completely untrustworthy. Thus, future clients need not waste resources delegating to a 'bad' server, and servers are disincentivised from cheating in the first place as they will incur a significant (financial) penalty from not receiving future work. Many current schemes have an expensive pre-processing stage run by the client. However, it is likely that many different clients will be interested in outsourcing computations, and that functions of interest to each client will substantially overlap, as in the "bring your own device" scenario above. It is also conceivable that the number of servers offering to perform such computations will be relatively low (limited to a reasonably small number of trusted companies with plentiful resources). Thus, it is easy to envisage a situation in which many computationally limited clients wish to outsource computations of the same (potentially large) set of functions to a set of untrusted servers. Current VC schemes do not support this kind of scenario particularly well.

Our main contribution, then, is to introduce the new notion of *Revocable Publicly Verifiable Computation* (RPVC). We also propose the introduction of a Key Distribution Center (KDC) to perform the computationally intensive parts of VC and manage keys for all clients, and we simplify the way in which the computation of multiple functions is managed. We enable the revocation of misbehaving servers (those detected as cheating) such that they cannot perform further computations until recertified by the KDC, as well as "blind verification", a form of output privacy, such that the verifier learns whether the result is valid but not the value of the output. Thus the verifier may reward or punish servers appropriately without learning function outputs. We give a rigorous definitional framework for RPVC, that we believe more accurately reflects real environments. This new framework both removes redundancy and facilitates additional functionality, leading to several new security notions.

In the next section, we briefly review related work. In Section 3, we define our framework and the relevant security models. In Section 4, we provide a concrete instantiation of our framework using Attribute-based Encryption as well as full security proofs. Additional background details can be found in the Appendix.

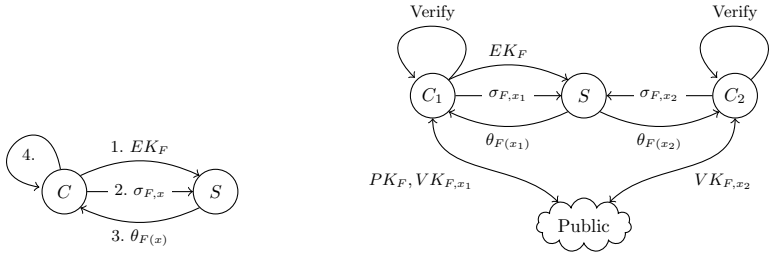
Notation. In the remainder of this paper we use the following notation. If A is a probabilistic algorithm we write $y \leftarrow A(\cdot)$ for the action of running A on given inputs and assigning the result to an output y. We denote the empty string by ϵ and use PPT to denote probabilistic polynomial-time. We say that $\mathrm{negl}(\cdot)$ is a negligible function on its input and κ denotes the security parameter. We denote by $\mathcal{F}$ the family of Boolean functions closed under complement – that is, if F belongs to $\mathcal{F}$ then $\overline{F}$, where $\overline{F}(x) = F(x) \oplus 1$, also belongs to $\mathcal{F}$. We denote the domain of F by $\mathsf{Dom}(F)$ and the range by $\mathsf{Ran}(F)$. By $\mathcal{M}$ we denote a message space and the notation $\mathcal{A}^{\mathcal{O}}$ is used to denote the adversary $\mathcal{A}$ being provided with oracle access. Finally, $[n]$ denotes the set $\{1, \ldots, n\}$.

2 Verifiable Computation Schemes and Related Work

The concept of non-interactive verifiable computation was introduced by Gennaro et al. [5] and may be seen as a protocol between two polynomial-time parties: a *client*, C, and a *server*, S. A successful run of the protocol results in the provably correct computation of $F(x)$ by the server for an input x supplied by the client. More specifically, a VC scheme comprises the following steps [5]:

1. KeyGen (*Run once*): C computes evaluation information EK_F that is given to S to enable it to compute F;
2. $\mathsf{ProbGen}$ (*Run multiple times*): C sends the encoded input $\sigma_{F,x}$ to S;
3. $\mathsf{Compute}$ (*Run multiple times*): S computes $F(x)$ using EK_F and $\sigma_{F,x}$ and returns an encoding of the output $\theta_{F(x)}$ to C;
4. Verify (*Run multiple times*): C checks whether $\theta_{F(x)}$ encodes $F(x)$.

The operation of a VC scheme is illustrated in Figure 1a. KeyGen may be computationally expensive but the remaining operations should be efficient for the client. The cost of setup is amortized over multiple computations of F.



(a) A VC system (b) A PVC system

Fig. 1: The operation of verifiable computation schemes

In prior work, Gennaro et al. [5] gave a construction using Garbled Circuits [11], which provides a "one-time" Verifiable Outsourced Computation allowing a client to outsource the evaluation of a function on a single input. However, the construction is insecure if the circuit is reused on a different input and thus this cost cannot be amortized. Moreover, the cost of generating a new garbled circuit is approximately equal to the cost of evaluating the function itself. The authors therefore suggested using fully homomorphic encryption [6] to re-randomise the circuit to allow multiple executions. In independent and concurrent work, Carter et al. [3] introduce a third party to generate garbled circuits for such schemes but require this entity to be online throughout and model the system as a secure multi-party computation between the client, server and third-party. Some works [7, 4] consider the multi-client case where functions are computed over joint input from multiple clients. Parno et al. [10] introduced *Publicly Verifiable Computation* (PVC), where a single client C_1 computes EK_F, as well as publishing information PK_F that enables other clients to encode inputs (so only one client has to run the expensive pre-processing stage). A client submits an input x and may publish $VK_{F,x}$ to allow other clients to verify the output. The operation of a PVC scheme is illustrated in Figure 1b. It uses the same four algorithms as VC but KeyGen and ProbGen now output public values that other clients may use to encode inputs and verify outputs. Parno et al. gave an instantiation of PVC using Key-Policy Attribute-based Encryption (KP-ABE) for a class of Boolean functions. Further details are available in Appendix A.

3 Revocable Publicly Verifiable Computation

We now describe our new notion of PVC, which we call *Revocable Publicly Verifiable Computation* (RPVC). We assume there is a Key Distribution Center (KDC) and many clients which make use of multiple untrusted or semi-trusted servers to perform complex computations. Multiple servers may be certified, by the KDC, to compute the same function F. As we briefly explained in the introduction, there appear to be good reasons for adopting an architecture of this nature and several scenarios in which such an architecture would be appropriate. The increasing popularity of relatively lightweight mobile computing devices in the workplace means that complex computations may best be performed by

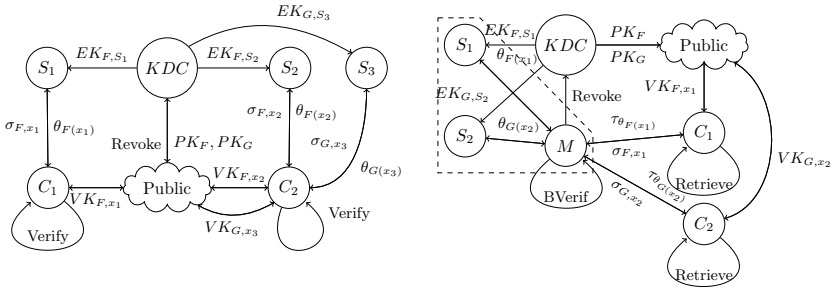
more powerful servers run by the organization or in the cloud and we would wish to have some guarantee that those servers are certified to perform certain functions. It is essential that we can verify the results of the computation. If cloud services are competing on price to provide "computation-as-a-service" then it is important that a server cannot obtain an unfair advantage by simply not bothering to compute $F(x)$ and returning garbage instead. It is also important that a server who is not certified cannot return a result without being detected.

Key Distribution Center. Existing frameworks assume that a client or clients run the expensive phases of a VC scheme and that a single server performs the outsourced computation. We believe that this is undesirable for a number of reasons, irrespective of whether the client is sufficiently powerful. First, in a real-world system, we may wish to outsource the setup phase to a trusted third party. In this setting, the third party would operate rather similarly to a certificate authority, providing a trust service to facilitate other operations of an organization (in this case outsourced computation, rather than authentication). Second, we may wish to enforce an access control policy limiting the functions each client can outsource; an internal trusted entity would operate both as a facilitator of outsourced computation and as a policy enforcement point. (We will examine the integration of RPVC and access control in future work.)

We consider the KDC to be a separate entity to illustrate separation of duty between the clients that request computations, and the KDC that is authoritative on the system and users. The KDC could be authoritative over many sets of clients (e.g. at an organizational level as opposed to a work group level), and we minimise its workload to key generation and revocation only. It may be tempting to suggest that the KDC, as a trusted entity, performs all computations itself. However we believe that this is not a practical solution in many real world scenarios, e.g. the KDC could be an authority within the organization responsible for user authorization that wishes to enable workers to securely use cloud-based software-as-a-service. As an entity within organization boundaries, performing all computations would negate the benefits gained from outsourcing computations to externally available servers.

System Architecture. In this paper we consider two system architectures, which we call the Standard Model and the Manager Model. The *standard model* is a natural extension of the PVC architecture with the addition of a KDC (as shown in Fig. 2a). The entities comprise a set of clients, a set of servers and a KDC. The KDC initializes the system and generates keys to enable verifiable computation. Clients submit computation requests to a particular server and publish some verification information. Any party can verify the correctness of a server's output. If the output is incorrect, the client may report the server to the KDC for revocation which will prevent the server from performing any further computations. The *manager model*, in contrast, employs an additional Manager entity who "owns" a pool of computation servers (as shown in Fig. 2b). Clients submit jobs to the manager, who will select a server from the pool based on

Algorithm	Run by			
	VC	**PVC**	**RPVC Standard**	**RPVC Manager**
KeyGen	C_1	C_1	KDC	KDC
ProbGen	C_1	$C_1, C_2, \ldots$	$C_1, C_2, \ldots$	$C_1, C_2, \ldots$
Compute	S	S	$S_1, S_2, \ldots$	$S_1, S_2, \ldots$
Verify	C_1	$C_1, C_2, \ldots$	$C_1, C_2, \ldots$	–
Blind Verify	–	–	–	M
Retrieve	–	–	–	$C_1, C_2, \ldots$



(a) Standard Model (b) Manager model

Fig. 2: The operation of RPVC

workload scheduling, available resources or by a bidding process if servers are to be rewarded per computation. A plausible scenario is that servers enlist with a manager to "sell" the use of spare resources, whilst clients subscribe to utilise these through the manager. Results are returned to the manager who should be able to verify the server's work. The manager forwards correct results to the client whilst a misbehaving server may be reported to the KDC for revocation, and the job assigned to another server. In some situations we may not desire external entities to access the result, yet there remain legitimate reasons for the manager to perform verification. Thus we introduce "blind verification", as hinted by Parno et al. [10], such that the manager (or other entity) may verify the validity of the computation without learning the output, while the client holds an extra key that enables the output to be retrieved.

3.1 Formal Definition

Definition 1. A *Revocable Publicly Verifiable Outsourced Computation Scheme (RPVC)* comprises the following algorithms:

- $(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$: Run by the KDC to establish public parameters PP and a master secret key MK.
- $PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$: Run by the KDC to generate a public delegation key, PK_F, for a function F.
- $SK_S \leftarrow \mathsf{Register}(S, MK, PP)$: Run by the KDC to generate a personalised signing key SK_S for a computation server S.

- $EK_{F,S} \leftarrow \mathsf{Certify}(S, F, MK, PP)$: Run by the KDC to generate a certificate in the form of an evaluation key $EK_{F,S}$ for a function F and server S.
- $(\sigma_{F,x}, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{ProbGen}(x, PK_F, PP)$: ProbGen is run by a client to delegate the computation of $F(x)$ to a server. The output value $RK_{F,x}$ is used to enable output retrieval after the blind verification step.
- $\theta_{F(x)} \leftarrow \mathsf{Compute}(\sigma_{F,x}, EK_{F,S}, SK_S, PP)$: Run by a server S holding an evaluation key $EK_{F,S}$, SK_S and an encoded input $\sigma_{F,x}$ of x, to output an encoding, $\theta_{F(x)}$, of $F(x)$, including an identifier of S.
- $(\tilde{y}, \tau_{\theta_{F(x)}}) \leftarrow \mathsf{Verify}(\theta_{F(x)}, VK_{F,x}, RK_{F,x}, PP)$:
  Verification comprises:
  - $(RT_{F,x}, \tau_{\theta_{F(x)}}) \leftarrow \mathsf{BVerif}(\theta_{F(x)}, VK_{F,x}, PP)$: Run by any verifying party (standard model), or by the manager (manager model), in possession of $VK_{F,x}$ and an encoded output, $\theta_{F(x)}$. This outputs a token $\tau_{\theta_{F(x)}} = (\mathsf{accept}, S)$ if the output is valid, or $\tau_{\theta_{F(x)}} = (\mathsf{reject}, S)$ if S misbehaved. It also outputs a retrieval token $RT_{F,x}$ which is an encoding of the actual output value.
  - $\tilde{y} \leftarrow \mathsf{Retrieve}(\tau_{\theta_{F(x)}}, RT_{F,x}, VK_{F,x}, RK_{F,x}, PP)$: Run by a verifier holding $RK_{F,x}$ to retrieve the actual result $\tilde{y}$ which is either $F(x)$ or $\perp$.[1]
- $\{EK_{F,S'}\}$ or $\perp \leftarrow \mathsf{Revoke}(\tau_{\theta_{F(x)}}, MK, PP)$: Run by the KDC if a misbehaving server is reported i.e. that Verify returned $\tau_{\theta_{F(x)}} = (\mathsf{reject}, S)$ (if $\tau_{\theta_{F(x)}} = (\mathsf{accept}, S)$ then this algorithm outputs $\perp$). It revokes all evaluation keys $EK_{\cdot,S}$ of the server S thereby preventing S from performing any further evaluations. Updated evaluation keys $EK_{\cdot,S'}$ are issued to all servers.[2]

Although not stated, the KDC may update the public parameters PP during any algorithm. A RPVC scheme is *correct* if the verification algorithm almost certainly outputs accept when run on a valid verification key and an encoded output, where the encoded output is honestly produced by a computation server given a validly generated encoded input and evaluation key. That is, if all algorithms are run honestly then the result should almost certainly be accepted.

3.2 Security Models

We now formalize several notions of security as a series of cryptographic games. The adversary against a particular function F is modelled as a PPT algorithm $\mathcal{A}$ run by a challenger with input parameters chosen to represent the knowledge of a real attacker as well the security parameter κ and a parameter $q_t > 1$ denoting the number of queries the adversary makes to the Revoke oracle before the challenge is generated. The adversary algorithm may maintain state and be multi-stage and we overload the notation by calling each of these adversary algorithms $\mathcal{A}$. The notation $\mathcal{A}^{\mathcal{O}}$ denotes the adversary $\mathcal{A}$ being provided with oracle access to the following functions: $\mathsf{FnInit}(\cdot, MK, PP)$, $\mathsf{Register}(\cdot, MK, PP)$,

[1] Note that if a server is not given $RK_{F,x}$ then it too cannot learn the output.

[2] In some instantiations, it may not be necessary to issue entirely new evaluation keys to each entity. In Sect. 4, we only need to issue a partially updated key for example.

$\mathsf{Certify}(\cdot,\cdot,\cdot,MK,PP)$ and $\mathsf{Revoke}(\cdot,\cdot,\cdot,MK,PP)$.[3] For each game, we define the *advantage* and *security* of $\mathcal{A}$ as:

Definition 2. *The* advantage *of a PPT adversary $\mathcal{A}$ making a polynomial number of queries q (including q_t* Revoke *queries) is defined as follows, where $\mathbf{X} \in \{sSS\text{-}PubVerif, sSS\text{-}Revocation, sSS\text{-}VindictiveM\}$:*

- $Adv_{\mathcal{A}}^{\mathbf{X}}(\mathcal{RPVC}, F, 1^\kappa, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathbf{X}}[\mathcal{RPVC}, F, q_t, 1^\kappa] = 1]$
- $Adv_{\mathcal{A}}^{VindictiveS}(\mathcal{RPVC}, F, 1^\kappa, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{VindictiveS}[\mathcal{RPVC}, F, 1^\kappa] = 1]$
- $Adv_{\mathcal{A}}^{BVerif}(\mathcal{RPVC}, F, 1^\kappa, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{BVerif}[\mathcal{RPVC}, F, 1^\kappa] = 1] - \max_{y \in \mathsf{Ran}(F)} (\Pr_{x \in \mathsf{Dom}(F)}[F(x) = y])$.

A RPVC is secure against Game $\mathbf{X}$, VindictiveS or BVerif *for a function F, if for all PPT adversaries $\mathcal{A}$, $Adv_{\mathcal{A}}^{\mathbf{X},VindictiveS,BVerif}(\mathcal{RPVC}, F, 1^\kappa, q) \leq \mathrm{negl}(\kappa)$.*

Public Verifiability In Game 1 we extend the Public Verifiability game of Parno et al. [10] to formalize that multiple colluding servers should be unable to convince *any* verifying party of an incorrect output (i.e. that Verify returns accept on an encoded output not representing the true output of the computation). We define a selective, semi-static notion[4] such that the adversary must select its challenge input before seeing the public parameters and must declare a list of entities that must be revoked at the challenge time before receiving oracle access.

The adversary first selects an input value to be outsourced. The challenger initializes a list of currently revoked entities Q_{Rev} and a time parameter t before running Setup and FnInit to create a public delegation key for the function F (lines 2 to 5). The adversary is given the generated public parameters and must output a list $\overline{R}$ of servers to be revoked when the challenge is created. It is then given oracle access to the above functions which simulate all values known to a real server as well as those learnt through corrupting entities. The challenger responds to Certify and Revoke queries as detailed in Oracle Queries 1 and 2 respectively. It must ensure that Q_{Rev} is kept up-to-date by adding or removing the queried entity, and in the case of revocation must increment the time parameter. It also ensures that issued keys will not lead to a trivial win.

Once the adversary has finished this query phase (and in particular, due to the parameterisation of the adversary, after exactly q_t Revoke queries), the challenger must check that the queries made by the adversary has indeed left the list of revoked entities to be at least that selected beforehand by the adversary. If there is a server that the adversary included on $\overline{R}$ but is not currently revoked, then the adversary loses the game. Otherwise, the challenger generates the challenge by running ProbGen on $x^\star$. The adversary is given the resulting encoded input and oracle access again, and wins the game if it creates an encoded output that verifies correctly yet does not encode the correct value $F(x^\star)$.

[3] We do not need to provide a Verify oracle since this is a publicly verifiable scheme and $\mathcal{A}$ is given verification keys (thus we also avoid the rejection problem).

[4] This is due to the selective IND-sHRSS game that we base the construction upon. Since this is used in a black-box manner however, a stronger primitive may allow this game to be improved accordingly.

Game 1 $\mathbf{Exp}_{\mathcal{A}}^{sSS\text{-}PubVerif}[\mathcal{RPVC}, F, q_t, 1^\kappa]$:

1: $x^\star \leftarrow \mathcal{A}(1^\kappa)$;
2: $Q_{\mathrm{Rev}} = \epsilon$;
3: $t = 1$;
4: $(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$;
5: $PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$;
6: $\overline{R} \leftarrow \mathcal{A}(PK_F, PP)$;
7: $\mathcal{A}^{\mathcal{O}}(PK_F, PP)$;
8: **if** $(\overline{R} \not\subseteq Q_{\mathrm{Rev}})$ **return** 0;
9: $(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}) \leftarrow \mathsf{ProbGen}(x^\star, PK_F, PP)$;
10: $\theta^\star \leftarrow \mathcal{A}^{\mathcal{O}}(\{\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}\}, EK_{F,\mathcal{A}}, SK_{\mathcal{A}}, PK_F, PP)$;
11: **if** $((((\tilde{y}, \tau_{\theta^\star}) \leftarrow \mathsf{Verify}(\theta^\star, VK_{F,x^\star}, RK_{F,x^\star}, PP))$
and $((\tilde{y}, \tau_{\theta^\star}) \neq (\perp, (\mathsf{reject}, \cdot)))$ **and** $(\tilde{y} \neq F(x^\star))))$
12: **return** 1;
13: **else return** 0;

Oracle Query 1 $\mathcal{O}^{\mathsf{Certify}}(S, F', MK, PP)$:

1: **if** $((F' = F$ **and** $S \notin \overline{R})$ **or** $(t = q_t$ **and** $\overline{R} \not\subseteq Q_{\mathrm{Rev}} \setminus S))$ **return** $\perp$;
2: $Q_{\mathrm{Rev}} = Q_{\mathrm{Rev}} \setminus S$;
3: **return** $\mathsf{Certify}(S, F', MK, PP)$;

Oracle Query 2 $\mathcal{O}^{\mathsf{Revoke}}(\tau_{\theta_{F'(x)}}, MK, PP)$:

1: $t = t + 1$;
2: **if** $(\tau_{\theta_{F'(x)}} = (\mathsf{accept}, \cdot))$ **return** $\perp$;
3: **if** $(t = q_t$ **and** $\overline{R} \not\subseteq Q_{\mathrm{Rev}} \cup S)$ **return** $\perp$;
4: $Q_{\mathrm{Rev}} = Q_{\mathrm{Rev}} \cup S$;
5: **return** $\mathsf{Revoke}(\tau_{\theta_{F'(x)}}, MK, PP)$;

Revocation The notion of revocation requires that any subsequent computations by a server detected as misbehaving should be rejected (even if the result is correct). Thus a misbehaving server may be completely removed from the system and will be punished by not receiving rewards for future work.

The selective, semi-static notion of Revocation given in Game 2 proceeds exactly as the sSS-PubVerif game except that the adversary wins if it outputs *any* result (even a correct encoding of $F(x^\star)$) that is accepted as a valid response from any server that was revoked at the time of the challenge. This game also uses the Certify and Revoke oracles specified in Oracle Queries 1 and 2 respectively.

Vindictive Server This notion is motivated by the manager model where the client does not a priori know the identities of servers selected from the pool. Since an invalid result can lead to revocation, this reveals a new threat model (particularly if servers are rewarded per computation). A malicious server may return incorrect results but attribute them to an alternate server ID such that an (honest) server is revoked, thus reducing the size of the server pool and increasing the future reward for the malicious server. In Game 3, the challenger maintains a list of registered entities Q_{Reg}. The game proceeds similarly to the previous notions, except that, on lines 6 and 7, the adversary selects a target server ID, $\tilde{S}$,

Game 2 $\mathbf{Exp}_{\mathcal{A}}^{sSS\text{-}Revocation}[\mathcal{RPVC}, F, q_t, 1^\kappa]$:

$x^\star \leftarrow \mathcal{A}(1^\kappa)$;
$Q_{\mathrm{Rev}} = \epsilon$;
$t = 1$;
$(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$;
$PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$;
$\overline{R} \leftarrow \mathcal{A}(PK_F, PP)$;
$\mathcal{A}^{\mathcal{O}}(PK_F, PP)$;
if $(\overline{R} \not\subseteq Q_{\mathrm{Rev}})$ **return** 0;
$(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}) \leftarrow \mathsf{ProbGen}(x^\star, PK_F, PP)$;
$\theta^\star \leftarrow \mathcal{A}^{\mathcal{O}}(\sigma_{x^\star}, VK_{F,x^\star}, RK_{F,x^\star}, PK_F, PP)$;
if $(((\tilde{y}, (\mathsf{accept}, S)) \leftarrow \mathsf{Verify}(\theta^\star, VK_{F,x^\star}, RK_{F,x^\star}, PP))$ **and** $(S \in \overline{R})$ **then**
  return 1
else
  return 0

Game 3 $\mathbf{Exp}_{\mathcal{A}}^{VindictiveS}[\mathcal{RPVC}, F, 1^\kappa]$:

$Q_{\mathrm{Reg}} = \epsilon$;
$(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$;
$PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$;
$x^\star \leftarrow \mathcal{A}^{\mathcal{O}}(PK_F, PP)$;
$(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}) \leftarrow \mathsf{ProbGen}(x^\star, PK_F, PP)$;
$\tilde{S} \leftarrow \mathcal{A}^{\mathcal{O},\mathsf{Register2}}(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}, PK_F, PP)$ subject to (1);
$\theta^\star \leftarrow \mathcal{A}^{\mathcal{O},\mathsf{Compute},\mathsf{Register2}}(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}, PK_F, PP)$ subject to (2);
if $((\tilde{y}, \tau_{\theta^\star}) \leftarrow \mathsf{Verify}(\theta^\star, VK_{F,x^\star}, RK_{F,x^\star}, PP))$ **and** $((\tilde{y}, \tau_{\theta^\star}) = (\perp, (\mathsf{reject}, \tilde{S})))$ **and** $(\perp \nleftarrow \mathsf{Revoke}(\tau_{\theta^\star}, MK, PP)))$ **then**
  return 1
else
  return 0

he wishes to be revoked and generates an encoded output that will cause this. He is given oracle access subject to the following constraints to avoid trivial wins:
(1) No query of the form $\mathcal{O}^{\mathsf{Register}}(\tilde{S}, MK, PP)$ was made;
(2) As above and no query $\mathcal{O}^{\mathsf{Compute}}(\sigma_{F,x_i^\star}, EK_{F,\tilde{S}}, SK_{\tilde{S}}, PP)$ was made.
In addition, he is provided with an oracle, Register2, which performs the Register algorithm but *does not* return the resulting key SK_S (it may however update the public parameters to reflect the additional registered entity). The adversary may query *any* identity to Register2 (including $\tilde{S}$). We also modify the standard Register oracle such that if an identity has been previously queried to the Register2 oracle, it generates the same parameters (and vice versa). The adversary wins if the KDC believes $\tilde{S}$ returned $\tilde{y}$ and revokes $\tilde{S}$.

Vindictive Manager This is a natural extension of the Public Verifiability notion to the manager model where a vindictive manager may attempt to provide a client with an incorrect answer. We remark that instantiations may vary depending on the level of trust given to the manager: a completely trusted manager may simply return the result to a client, whilst an untrusted manager may

Game 4 $\mathbf{Exp}_{\mathcal{A}}^{sSS\text{-}VindictiveM}[\mathcal{RPVC}, F, q_t, 1^\kappa]$:

$x^\star \leftarrow \mathcal{A}(1^\kappa)$;
$Q_{\mathrm{Rev}} = \epsilon$;
$t = 1$
$(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$;
$PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$;
$\overline{R} \leftarrow \mathcal{A}(PK_F, PP)$
$\mathcal{A}^{\mathcal{O}}(PK_F, PP)$;
if $((\overline{R} \not\subseteq Q_{\mathrm{Rev}})$ **or** $(\overline{R} = \mathcal{U}_{\mathrm{ID}}))$ **return** 0;
$S \xleftarrow{\$} \mathcal{U}_{\mathrm{ID}} \setminus \overline{R}$;
$SK_S \leftarrow \mathsf{Register}(S, MK, PP)$;
$EK_{F,S} \leftarrow \mathsf{Certify}(S, F, MK, PP)$;
$(\sigma_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}) \leftarrow \mathsf{ProbGen}(x^\star, PK_F, PP)$;
$\theta_{F(x^\star)} \leftarrow \mathsf{Compute}(\sigma_{F,x^\star}, EK_{F,S}, SK_S, PP)$;
$(RT_{F,x^\star}, \tau_{\theta_{F(x^\star)}}) \leftarrow \mathcal{A}^{\mathcal{O}}(\sigma_{F,x^\star}, \theta_{F(x^\star)}, VK_{F,x^\star}, PK_F, PP)$;
if $(\tilde{y} \leftarrow \mathsf{Retrieve}(\tau_{\theta_{F(x^\star)}}, RT_{F,x^\star}, VK_{F,x^\star}, RK_{F,x^\star}, PP))$
and $(\tilde{y} \neq F(x^\star))$ **and** $(\tilde{y} \neq \perp)$ **then**
  return 1
else
  return 0

have to provide the full output from the server. Here we consider a semi-trusted manager where the clients would still like a final, efficient check.

The security model is presented in Game 4. First, the adversary selects its challenge input $x^\star$, and the challenger initializes a list of revoked entities Q_{Rev} and a time paramter t. It also sets up the system and gives the public parameters to the adversary, who must select a list $\overline{R}$ of servers to be revoked at the challenge time. We require that $\overline{R}$ is not the full set of all servers in the system, as one non-revoked identity is required to generate the challenge. The adversary then gets oracle access (using the Certify and Revoke oracles specified in Oracle Queries 1 and 2 respectively). If, after finishing this query phase (and in particular after q_t Revoke queries), the list of revoked entities does not include $\overline{R}$ then the adversary loses the game. Otherwise, a server S is chosen at random from the set of all server identities $\mathcal{U}_{\mathrm{ID}}$ excluding $\overline{R}$ (as these must be revoked at the challenge time). This server is used to generate the challenge. If not already done, the challenger registers and certifies S for F, and runs ProbGen on the challenge input, before finally running Compute to generate an encoded output $\theta_{F(x^\star)}$. The adversary is then given the encoded input, verification key and $\theta_{F(x^\star)}$, as well as oracle access, and must output a retrieval token $RT_{F,x^\star}$ and an acceptance token $\tau_{\theta_{F(x^\star)}}$. The challenger runs Retrieve on $RT_{F,x}$ to get an output value $\tilde{y}$, and the adversary wins if the challenger accepts this output and $\tilde{y} \neq F(x^\star)$.

Blind Verification With this notion we aim to show that a verifier that does not hold the retrieval token $RT_{F,x}$ chosen in ProbGen cannot learn the value of $F(x)$ given the encoded output. This property was hinted at by Parno et al. [10] but was not formalized. The game begins as usual with the challenger initializing the system. The challenger then selects an input at random from the domain of F, and a random server S. It registers and certifies S, runs ProbGen for the

Game 5 $\mathbf{Exp}_{\mathcal{A}}^{BVerif}[\mathcal{RPVC}, F, 1^\kappa]$:

$(PP, MK) \leftarrow \mathsf{Setup}(1^\kappa)$;
$PK_F \leftarrow \mathsf{FnInit}(F, MK, PP)$;
$x \xleftarrow{\$} \mathsf{Dom}(F)$;
$S \xleftarrow{\$} \mathcal{U}_{\mathrm{ID}}$;
$SK_S \leftarrow \mathsf{Register}(S, MK, PP)$;
$EK_{F,S} \leftarrow \mathsf{Certify}(S, F, MK, PP)$;
$(\sigma_{F,x}, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{ProbGen}(x, PK_F, PP)$;
$\theta_{F(x)} \leftarrow \mathsf{Compute}(\sigma_{F,x}, EK_{F,S}, SK_S, PP)$;
$\hat{y} \leftarrow \mathcal{A}^{\mathcal{O}}(\theta_{F(x)}, VK_{F,x}, PK_F, PP)$;
if $(\hat{y} = F(x))$ **then**
  return 1
else
  return 0

chosen input and runs Compute to generate an output $\theta_{F(x)}$. This is given to the adversary along with the verification key and oracle access, and the adversary wins if it can guess the value of $F(x)$ without seeing the retrieval key. Clearly, the adversary can trivially make a guess for $F(x)$ based on a priori knowledge of the distribution of F over all possible inputs. Unless F is balanced (i.e. outputs 1 exactly half the time), the adversary could gain an advantage. Thus, we define security by subtracting the most likely guess for $F(x)$.

4 Construction

We now provide an instantiation of a RPVC scheme. Our construction is based on that used by Parno et al. [10] (summarised in App. A) which uses Key-Policy Attribute-based Encryption (KP-ABE) in a black-box manner to outsource the computation of a Boolean function.[5] Notice that to achieve the outsourced evaluation of functions with n bit outputs, it is possible to evaluate n different functions, each of which applies a mask to output the single bit in position i.

Recall that if $\perp$ is returned by the server then the verifier is unable to determine whether $F(x) = 0$ *or* whether the server misbehaved. To avoid this issue, we follow Parno *et al.* and restrict the family of functions $\mathcal{F}$ to be the set of Boolean functions closed under complement. That is, if $F \in \mathcal{F}$ then $\overline{F}(x) = F(x) \oplus 1$ also belongs to $\mathcal{F}$. Then, the client encrypts two random messages m_0 and m_1. The server is required to return the decryption of those ciphertexts. Thus, a well-formed response $\theta_{F(x)}$, comprising recovered plaintexts (d_b, d_{1-b}), satisfies the following, where $RK_{F,x} = b$:

$$(d_b, d_{1-b}) = \begin{cases} (m_b, \perp), & \text{if } F(x) = 1; \\ (\perp, m_{1-b}), & \text{if } F(x) = 0. \end{cases} \tag{1}$$

[5] Following Parno et al. we restrict our attention to Boolean functions, and in particular the complexity class NC^1 which includes all circuits of depth $\mathcal{O}(\log n)$. Thus functions we can outsource can be built from common operations such as AND, OR, NOT, equality and comparison operators, arithmetic operators and regular expressions.

4.1 Technical Details

We require an *indirectly revocable KP-ABE scheme* comprising the algorithms ABE.Setup, ABE.KeyGen, ABE.KeyUpdate, ABE.Encrypt and ABE.Decrypt. We also use a signature scheme with algorithms Sig.KeyGen, Sig.Sign and Sig.Verify, and a one-way function g. Let $\mathcal{U}$ be the universe of attributes acceptable by the ABE scheme, and let $\mathcal{U} = \mathcal{U}_{\text{attr}} \cup \mathcal{U}_{\text{ID}} \cup \mathcal{U}_{\text{time}} \cup \mathcal{U}_{\mathcal{F}}$ where: attributes in $\mathcal{U}_{\text{attr}}$ form characteristic tuples for input data, as detailed in Appendix A; $\mathcal{U}_{\text{ID}}$ comprises attributes representing entity identifiers; $\mathcal{U}_{\text{time}}$ comprises attributes representing time periods issued by the time source $\mathbb{T}$; and finally $\mathcal{U}_{\mathcal{F}}$ comprises attributes that represent functions in $\mathcal{F}$. Define a bijective mapping between functions $F \in \mathcal{F}$ and attributes $f \in \mathcal{U}_{\mathcal{F}}$. Then the policy $F \wedge f$ denotes adding a conjunctive clause requiring the presence of the label f to the expression of the function F, and $(x \cup f)$ denotes adding the function attribute to the attribute set representing the input data x. This will prevent servers using alternate evaluation keys for a given input and hence we are able to certify servers to compute multiple functions.

Parno et al. [10] considered two models of publicly verifiable computation. In single function PVC, the function to be computed is embedded in the public parameters, whilst in multi-function PVC delegation keys for multiple functions can be generated and a single encoded input can be used to input the same data to multiple functions. To achieve this latter notion, Parno et al. required the somewhat complex primitive of KP-ABE with Outsourcing [9]. In this work, we take a different approach. We believe that in practical environments it is unrealistic to expect a server to compute just a single function, and we also believe that it is a reasonable cost expectation to prepare an encoded input per computation, and that the input data to different functions may well differ. Thus, whereas Parno et al. use complex primitives to allow an encoded input to be used for computations of different functions on the same data, we use the simple trick of adding a conjunctive clause to the functions requiring the presence of the appropriate function label in the input data – that is, the function F is encoded in a decryption key for the policy $F \wedge f$ where f is the attribute representation of F in $\mathcal{U}_{\mathcal{F}}$; the complement function $\overline{F}$ is encoded as a key for $\overline{F} \wedge f$; and we encode the input data x to the function F as $x \cup f$. Thus, the client must perform the ProbGen stage per computation as the function label in the input data will differ, but servers can be certified for multiple functions and may not use a key for one function to compute on data intended for another (since the function label required by the conjunctive clause in the key will not be present in the input data). As a result, and unlike the single function notion of Parno et al., we are able to provide the adversary with oracle access in our security games.

The scheme of Parno et al. required a one-key IND-CPA notion of security for the underlying KP-ABE scheme. This is a more relaxed notion than considered in the vast majority of the ABE literature (where the adversary is given a KeyGen oracle and the scheme must prevent collusion between holders of different decryption keys). Parno et al. could use this property due to their restricted system model where the client is certified for only a single function per set of public parameters (so the client must set up a new ABE environment per func-

tion). In our setting, we must be able to certify servers for multiple functions and hence the KDC must be able to issue multiple keys and we require the more standard, multi-key notion of security usually considered for ABE schemes.

4.2 Instantiation

Informally the scheme operates as follows.

1. RPVC.Setup establishes public parameters and a master secret key by calling the ABE.Setup algorithm twice. This algorithm also initializes a time source[6] $\mathbb{T}$, a list of revoked servers, and a two-dimensional array of registered servers L_{Reg} – the array is indexed in the first dimension by server identities and the first dimension will store signature verification keys while the second will store a list of functions that server is authorized to compute.
2. RPVC.FnInit simply outputs the public parameters.
3. RPVC.Register creates a public-private key pair by calling the signature KeyGen algorithm. This is run by the KDC (or the manager in the manager model) and updates L_{Reg} to store the verification key for S.
4. RPVC.Certify creates the key $EK_{F,S}$ that will be used by a server S to compute F by calling the ABE.KeyGen and ABE.KeyUpdate algorithms twice – once with a "policy" for F and once with the complement $\overline{F}$. It also updates L_{Reg} to include F. Note that since we have a form of multi-function PVC, we must prevent a server certified to perform two different functions, F and G (that differ on their output) from using the key for G to retrieve the plaintext and claiming it as a result for F. To prevent this, we add an additional attribute to the input set in ProbGen encoding the function the input should applied to, and add a conjunctive clause for such an attribute to the key policies. Thus an input set intended for F (including the F attribute) will only satisfy a key issued for F (comprising the F conjunctive clause), and a key for G will not be satisfied as G is not in the input set.
5. RPVC.ProbGen creates a problem instance $\sigma_{F,x} = (c_b, c_{1-b})$ by encrypting two randomly chosen messages under an attribute set corresponding to x, and a verification key $VK_{F,x}$ by applying a one-way function g to the messages. The ciphertexts and verification tokens are ordered randomly according to $RK_{F,x} = b$ for a random bit b, such that the positioning of an element does not imply whether it relates to F or to $\overline{F}$.
6. RPVC.Compute is run by a server S. Given an input $\sigma_{F,x} = (c_b, c_{1-b})$ it returns $(m_0, \perp)$ if $F(x) = 1$ or $(\perp, m_1)$ if $F(x) = 0$ (ordered according to $RK_{F,x}$ chosen in RPVC.ProbGen) and a signature on the output.
7. RPVC.Verify either accepts the output $\theta_{F(x)} = (d_b, d_{1-b})$ or rejects it. This algorithm verifies the signature on the output and confirms the output is correct by applying g and comparing with $VK_{F,x}$. In RPVC.BVerif the verifier can compare pairwise between the components of $\theta_{F(x)}$ and $VK_{F,x}$ to determine correctness but as they are unaware of the value of $RK_{F,x}$, they

[6] $\mathbb{T}$ could be a counter that is maintained in the public parameters or a networked clock.

do not know the order of these elements and hence whether the correct output corresponds to F or $\overline{F}$ being satisfied i.e. if $F(x) = 1$ or 0 respectively. The verifier outputs an accept or reject token as well as the output value $RT_{F,x} \in \{d_b, d_{1-b}, \perp\}$ where $RK_{F,x} = b$. Parno et al. [10] gave a one line remark that permuting the key pairs and ciphertexts given out in ProbGen could give output privacy. We believe that doing so would require four decryptions in the Compute stage to ensure the correct keys have been used (since an incorrect key,associated with different public parameters, but for a satisfying attribute set will return an incorrect, random plaintext which is indistinguishable from a valid, random message). Since our construction fixes the order of the key pairs, we do not have this issue and only require two decryptions. In RPVC.Retrieve a verifier that has knowledge of $RK_{F,x}$ can check whether the output from BVerif matches m_0 or m_1.

8. RPVC.Revoke is run by the KDC and redistributes fresh keys to all non-revoked servers. This algorithm first refreshes the time source $\mathbb{T}$ (e.g. increments $\mathbb{T}$ if it is a counter). It then updates L_{Reg} and L_{Rev}, and updates $EK_{F,S}$ using the results of two calls to the ABE.KeyUpdate algorithm.

More formally, our scheme is defined by Algorithms 1–9.

Alg. 1 $(PP, MK) \leftarrow \mathsf{RPVC.Setup}(1^\kappa)$

1: Let $\mathcal{U} = \mathcal{U}_{\text{attr}} \cup \mathcal{U}_{\text{ID}} \cup \mathcal{U}_{\text{time}} \cup \mathcal{U}_{\mathcal{F}}$
2: $(MPK^0_{\text{ABE}}, MSK^0_{\text{ABE}}) \leftarrow \mathsf{ABE.Setup}(1^\kappa, \mathcal{U})$
3: $(MPK^1_{\text{ABE}}, MPK^1_{\text{ABE}}) \leftarrow \mathsf{ABE.Setup}(1^\kappa, \mathcal{U})$
4: **for** $S \in \mathcal{U}_{\text{ID}}$ **do**
5: $\quad L_{\text{Reg}}[S][0] = \epsilon$
6: $\quad L_{\text{Reg}}[S][1] = \{\epsilon\}$
7: $L_{\text{Rev}} = \epsilon$
8: Initialise $\mathbb{T}$
9: $PP = (MPK^0_{\text{ABE}}, MPK^1_{\text{ABE}}, L_{\text{Reg}}, \mathbb{T})$
10: $MK = (MSK^0_{\text{ABE}}, MSK^1_{\text{ABE}}, L_{\text{Rev}})$

Alg. 2 $PK_F \leftarrow \mathsf{RPVC.FnInit}(F, MK, PP)$

1: Set $PK_F = PP$

Alg. 3 $SK_S \leftarrow \mathsf{RPVC.Register}(S, MK, PP)$

1: $(SK_{\text{Sig}}, VK_{\text{Sig}}) \leftarrow \mathsf{Sig.KeyGen}(1^\kappa)$
2: $SK_S = SK_{\text{Sig}}$
3: $L_{\text{Reg}}[S][0] = VK_{\text{Sig}}$

Alg. 4 $EK_{F,S} \leftarrow \mathsf{RPVC.Certify}(S, F, MK, PP)$

1: $L_{\text{Reg}}[S][1] = L_{\text{Reg}}[S][1] \cup F$
2: $L_{\text{Rev}} = L_{\text{Rev}} \setminus S$
3: $t \leftarrow \mathbb{T}$
4: $SK^0_{\text{ABE}} \leftarrow \mathsf{ABE.KeyGen}(S, F \wedge f, MSK^0_{\text{ABE}}, MPK^0_{\text{ABE}})$
5: $SK^1_{\text{ABE}} \leftarrow \mathsf{ABE.KeyGen}(S, \overline{F} \wedge f, MSK^1_{\text{ABE}}, MPK^1_{\text{ABE}})$
6: $UK^0_{L_{\text{Rev}},t} \leftarrow \mathsf{ABE.KeyUpdate}(L_{\text{Rev}}, t, MSK^0_{\text{ABE}}, MPK^0_{\text{ABE}})$
7: $UK^1_{L_{\text{Rev}},t} \leftarrow \mathsf{ABE.KeyUpdate}(L_{\text{Rev}}, t, MSK^1_{\text{ABE}}, MPK^1_{\text{ABE}})$
8: $EK_{F,S} = (SK^0_{\text{ABE}}, SK^1_{\text{ABE}}, UK^0_{L_{\text{Rev}},t}, UK^1_{L_{\text{Rev}},t})$

Alg. 5 $(\sigma_{F,x}, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{RPVC.ProbGen}(x, PK_F, PP)$

$t \leftarrow \mathbb{T}$
$(m_0, m_1) \xleftarrow{\$} \mathcal{M} \times \mathcal{M}$
$b \xleftarrow{\$} \{0,1\}$
$c_b \leftarrow \mathsf{ABE.Encrypt}(m_b, (x \cup f), t, MPK^0_{\mathrm{ABE}})$
$c_{1-b} \leftarrow \mathsf{ABE.Encrypt}(m_{1-b}, (x \cup f), t, MPK^1_{\mathrm{ABE}})$
Output: $\sigma_{F,x} = (c_b, c_{1-b})$, $VK_{F,x} = (g(m_b), g(m_{1-b}), L_{\mathrm{Reg}})$ and $RK_{F,x} = b$

Alg. 6 $\theta_{F(x)} \leftarrow \mathsf{RPVC.Compute}(\sigma_{F,x}, EK_{F,S}, SK_S, PP)$

Input: $EK_{F,S} = (SK^0_{\mathrm{ABE}}, SK^1_{\mathrm{ABE}}, UK^0_{L_{\mathrm{Rev}},t}, UK^1_{L_{\mathrm{Rev}},t})$ and $\sigma_{F,x} = (c_b, c_{1-b})$
Parse $\sigma_{F,x}$ as (c, c')
$d_b \leftarrow \mathsf{ABE.Decrypt}(c, SK^0_{\mathrm{ABE}}, MPK^0_{\mathrm{ABE}}, UK^0_{L_{\mathrm{Rev}},t})$
$d_{1-b} \leftarrow \mathsf{ABE.Decrypt}(c', SK^1_{\mathrm{ABE}}, MPK^1_{\mathrm{ABE}}, UK^1_{L_{\mathrm{Rev}},t})$
$\gamma \leftarrow \mathsf{Sig.Sign}((d_b, d_{1-b}, S), SK_S)$
Output: $\theta_{F(x)} = (d_b, d_{1-b}, S, \gamma)$

Alg. 7 $(RT_{F,x}, \tau_{\theta_{F(x)}}) \leftarrow \mathsf{RPVC.BVerif}(\theta_{F(x)}, VK_{F,x}, PP)$

Input: $VK_{F,x} = (g(m_b), g(m_{1-b}), L_{\mathrm{Reg}})$ and $\theta_{F(x)} = (d_b, d_{1-b}, S, \gamma)$
if $F \in L_{\mathrm{Reg}}[S][1]$ **then**
  if $\mathsf{accept} \leftarrow \mathsf{Sig.Verify}((d_b, d_{1-b}, S), \gamma, L_{\mathrm{Reg}}[S][0])$ **then**
    if $g(m_b) = g(d_b)$ **then** Output $(RT_{F,x} = d_b, \tau_{\theta_{F(x)}} = (\mathsf{accept}, S))$
    else if $g(m_{1-b}) = g(d_{1-b})$ **then** Output $(RT_{F,x} = d_{1-b}, \tau_{\theta_{F(x)}} = (\mathsf{accept}, S))$
    else
      Output $(RT_{F,x} = \perp, \tau_{\theta_{F(x)}} = (\mathsf{reject}, S))$
Output $(RT_{F,x} = \perp, \tau_{\theta_{F(x)}} = (\mathsf{reject}, \perp))$

Alg. 8 $\tilde{y} \leftarrow \mathsf{RPVC.Retrieve}(\tau_{\theta_{F(x)}}, RT_{F,x}, VK_{F,x}, RK_{F,x}, PP)$

Input: $VK_{F,x} = (g(m_b), g(m_{1-b}), L_{\mathrm{Reg}})$, $\theta_{F(x)} = (d_b, d_{1-b}, S, \gamma)$, $RK_{F,x} = b$, and $(RT_{F,x}, \tau_{\theta_{F(x)}})$ where $RT_{F,x} \in \{d_b, d_{1-b}, \perp\}$
if $(\tau_{\theta_{F(x)}} = (\mathsf{accept}, S)$ **and** $g(RT_{F,x}) = g(m_0))$ **then** Output $\tilde{y} = 1$
else if $(\tau_{\theta_{F(x)}} = (\mathsf{accept}, S)$ **and** $g(RT_{F,x}) = g(m_1))$ **then** Output $\tilde{y} = 0$
else Output $\tilde{y} = \perp$

Alg. 9 $\{EK_{F,S'}\}$ or $\perp \leftarrow \mathsf{RPVC.Revoke}(\tau_{\theta_{F(x)}}, MK, PP)$

if $\tau_{\theta_{F(x)}} = (\mathsf{reject}, S)$ **then**
  $L_{\mathrm{Reg}}[S][1] = \{\epsilon\}$
  $L_{\mathrm{Rev}} = L_{\mathrm{Rev}} \cup S$
  Refresh $\mathbb{T}$
  $t \leftarrow \mathbb{T}$
  $UK^0_{L_{\mathrm{Rev}},t} \leftarrow \mathsf{ABE.KeyUpdate}(L_{\mathrm{Rev}}, t, MSK^0_{\mathrm{ABE}}, MPK^0_{\mathrm{ABE}})$
  $UK^1_{L_{\mathrm{Rev}},t} \leftarrow \mathsf{ABE.KeyUpdate}(L_{\mathrm{Rev}}, t, MSK^1_{\mathrm{ABE}}, MPK^1_{\mathrm{ABE}})$
  for all $S \in \mathcal{U}_{\mathrm{ID}}$ **do**
    Parse $EK_{F,S}$ as $(SK^0_{\mathrm{ABE}}, SK^1_{\mathrm{ABE}}, UK^0_{L_{\mathrm{Rev}},t-1}, UK^1_{L_{\mathrm{Rev}},t-1})$
    Update and send $EK_{F,S} = (SK^0_{\mathrm{ABE}}, SK^1_{\mathrm{ABE}}, UK^0_{L_{\mathrm{Rev}},t}, UK^1_{L_{\mathrm{Rev}},t})$
else
  output $\perp$

Theorem 1. *Given a revocable KP-ABE scheme secure in the sense of indistinguishability against selective-target with semi-static query attack (*IND-sHRSS*) [2] for a class of Boolean functions $\mathcal{F}$ closed under complement, an*

EUF-CMA *secure signature scheme and a one-way function g. Let* $\mathcal{RPVC}$ *be the RPVC scheme defined in Algorithms 1–9. Then* $\mathcal{RPVC}$ *is secure in the sense of selective semi-static Public Verifiability, selective semi-static Revocation, Vindictive Servers, Blind Verification and selective semi-static Vindictive Managers.*

Lemma 1. *The* $\mathcal{RPVC}$ *construction defined by Algorithms 1–9 is secure against Vindictive Servers (Game 3) under the same assumptions as in Theorem 1.*

Proof. Let $\mathcal{A}_{VC}$ be an adversary with non-negligible advantage against the Vindictive Servers game (Game 3) when instantiated by Algorithms 1–9. We show that an adversary $\mathcal{A}_{Sig}$ with non-negligble advantage δ in the EUF-CMA signature game can be constructed using $\mathcal{A}_{VC}$. $\mathcal{A}_{Sig}$ interacts with the challenger $\mathcal{C}$ in the EUF-CMA security game and acts as the challenger for $\mathcal{A}_{VC}$ in the security game for Vindictive Servers for a function F as follows. The basic idea is that $\mathcal{A}_{Sig}$ can create a VC instance and play the Vindictive Servers game with $\mathcal{A}_{VC}$ by executing Algorithms 1–9 himself. $\mathcal{A}_{Sig}$ will guess a server identity that he thinks the adversary will select to vindictively revoke. The signature signing key that would be generated during the $\mathsf{Register}$ algorithm for this server will be implicitly set to be the signing key in the EUF-CMA game and any $\mathsf{Compute}$ oracle queries for this identity will be forwarded to the challenger to compute. Then, assuming that $\mathcal{A}_{Sig}$ guessed the correct server identity, $\mathcal{A}_{VC}$ will output a forged signature that $\mathcal{A}_{Sig}$ may output as its guess in the EUF-CMA game.

1. $\mathcal{C}$ initializes $Q = \epsilon$ to be an empty list of messages queried to the $\mathsf{Sig.Sign}$ oracle and runs $\mathsf{Sig.KeyGen}(1^\kappa)$ to generate a challenge signing key $\overline{SK}$ and verification key $\overline{VK}$. $\mathcal{C}$ sends $\overline{VK}$ to $\mathcal{A}_{Sig}$.
2. $\mathcal{A}_{Sig}$ chooses a function F on which to instantiate $\mathcal{A}_{VC}$.
3. $\mathcal{A}_{Sig}$ initializes the revocation list $Q_{\mathrm{Reg}} = \epsilon$. Furthermore, it chooses a server identity from $\mathcal{U}_{\mathsf{ID}} \setminus \mathcal{A}_{VC}$ which will be denoted by $\overline{S}$.
4. $\mathcal{A}_{Sig}$ runs $\mathsf{RPVC.Setup}(1^\kappa)$ and $\mathsf{RPVC.FnInit}(F, MK, PP)$, as specified in Algorithms 1 and 2 and passes PK_F and PP to the VC adversary $\mathcal{A}_{VC}$.
5. $\mathcal{A}_{VC}$ may now perform oracle queries to $\mathsf{RPVC.FnInit}$ $\mathsf{RPVC.Register}$ $\mathsf{RPVC.Certify}$ and $\mathsf{RPVC.Revoke}$ which $\mathcal{A}_{Sig}$ handles by running Algorithms 2, 3, 4 and 9 respectively.
6. Eventually, $\mathcal{A}_{VC}$ finishes querying and declares the challenge input $x^\star$.
7. $\mathcal{A}_{Sig}$ runs $\mathsf{RPVC.ProbGen}$ on the challenge $x^\star$ as specified in Algorithm 5.
8. $\mathcal{A}_{VC}$ is given the values of $PK_F, PP, \sigma_{F,x^\star}, VK_{F,x^\star}$ and $RK_{F,x^\star}$. It is also given oracle access to the following functions. $\mathcal{A}_{Sig}$ simulates these oracles and maintains a state of the generated parameters for each query.
   - $\mathsf{FnInit}(\cdot, MK, PP)$: $\mathcal{A}_{Sig}$ runs this step as per Algorithm 2.
   - $\mathsf{Register}(\cdot, MK, PP)$: If, for a queried server S, $S = \overline{S}$ then return $\perp$. Otherwise, $\mathcal{A}_{Sig}$ makes queries to $\mathcal{O}^{\mathsf{Register}}(S, MK, PP)$. If S has not been registered before and therefore does not appear on the registration list Q_{Reg} then the oracle returns a signing key SK_S for S and adds the pair (S, SK_S) to Q_{Reg}. Otherwise, the stored signing key is returned.
   - $\mathsf{Certify}(\cdot, \cdot, MK, PP)$: $\mathcal{A}_{Sig}$ honestly runs Algorithm 4.
   - $\mathsf{Revoke}(\cdot, MK, PP)$: $\mathcal{A}_{Sig}$ operates as in Algorithm 9.

– $\mathsf{Register2}(\cdot, MK, PP)$: $\mathcal{A}_{Sig}$ responds in the same way as for standard $\mathsf{Register}$ queries above, but *always* returns $\perp$ and not a signing key.

$\mathcal{A}_{VC}$ eventually outputs a target server identity $\tilde{S}$.

9. If $\tilde{S} \neq \overline{S}$ then $\mathcal{A}_{Sig}$ outputs $\perp$ and stops. Else, $\mathcal{A}_{VC}$ continues with oracle access as in Step 8 as well as a $\mathsf{Compute}$ oracle. $\mathcal{A}_{VC}$ submits queries of the form $\mathcal{O}^{\mathsf{Compute}}(\sigma_{F,x}, EK_{F,S}, SK_S, PP)$ for its choice of server S and $\sigma_{F,x}$. If $S \neq \overline{S}$ then $\mathcal{A}_{Sig}$ simply follows Algorithm 6 using the decryption and signing keys generated during the oracle queries. Otherwise, $S = \overline{S}$ and $\mathcal{A}_{Sig}$ does not have access to the signing key $SK_{\overline{S}}$. Thus, he runs the $\mathsf{ABE.Decrypt}$ operations correctly to generate plaintexts d_0 and d_1, and submits $m = (d_0, d_1, \overline{S})$ as a $\mathsf{Sig.Sign}$ oracle query to $\mathcal{C}$. $\mathcal{C}$ adds m to the list Q and returns $\gamma \leftarrow \mathsf{Sig.Sign}(m, \overline{SK})$, which $\mathcal{A}_{Sig}$ uses to return $\theta_{F(x)} = (d_0, d_1, \overline{S}, \gamma)$.
10. $\mathcal{A}_{VC}$ finally outputs $\theta^\star$ which appears to be an invalid result computed by $\tilde{S}$. Thus, Verify will output a reject token for $\tilde{S}$ and $\mathsf{accept} \leftarrow \mathsf{Sig.Verify}((d_0, d_1, \tilde{S}), \gamma, \overline{VK})$. Thus, γ is a valid signature under key $\overline{SK}$.
11. $\mathcal{A}_{Sig}$ outputs $m^\star = (d_0, d_1, \tilde{S})$ and $\gamma^\star = \gamma$ to $\mathcal{C}$.

Note that due to Constraint 2 in Game 3, $\mathcal{A}_{VC}$ is not allowed to have made a query for $\mathcal{O}^{\mathsf{Compute}}(\sigma_{x^\star}, EK_{F,\tilde{S}}, SK_{\tilde{S}}, PP)$ and thus the forgery $(m^\star, \gamma^\star)$ output by $\mathcal{A}_{Sig}$ will satisfy the requirement in the EUF-CMA game that $m^\star \notin Q$. We argue that, assuming $\overline{S} = \tilde{S}$ (i.e. $\mathcal{A}_{Sig}$ correctly guessed the challenge identity) then $\mathcal{A}_{Sig}$ succeeds with the same non-negligible advantage δ as $\mathcal{A}_{VC}$. We assume that $n = |\mathcal{U}_{\mathsf{ID}}|$ is polynomial (else the KDC could not efficiently search the list L_{Reg}). The probability that $\mathcal{A}_{Sig}$ correctly guesses $\overline{S} = \tilde{S}$ is $\frac{1}{n}$ and

$$Adv_{\mathcal{A}_{Sig}} \geq \frac{1}{n} Adv_{\mathcal{A}_{VC}} \geq \frac{\delta}{n} \geq negl(\kappa)$$

We conclude that if $\mathcal{A}_{VC}$ has a non-negligible advantage in the Vindictive Servers game then $\mathcal{A}_{Sig}$ has the same advantage in the EUF-CMA game, but since the signature scheme is assumed EUF-CMA secure, $\mathcal{A}_{VC}$ may not exist. □

5 Conclusion

We have introduced the new notion of RPVC and provided a rigorous framework that we believe to be more realistic than the purely theory oriented models of prior work, especially when the KDC is an entity responsible for user authorization within a organization. We believe our model more accurately reflects practical environments and the necessary interaction between entities for PVC. Each server may provide services for many different functions and for many different clients. The first model of Parno et al. [10] considered evaluations of a single function, while their second allowed for multiple functions but required a more exotic type of ABE scheme. This allowed a single $\mathsf{ProbGen}$ stage to encode input for any function, whilst in our model, we also allow multiple functions but use a simpler ABE scheme that also permits the revocation functionality. We require $\mathsf{ProbGen}$ to be run for each unique $F(x)$ to be outsourced which we believe

to be reasonable. Additionally, in our model, any clients may submit multiple requests to any available servers, whereas prior work considered just one server.

We have shown that by using a revocable KP-ABE scheme we can revoke misbehaving servers such that they receive a penalty for cheating and that, by permuting elements within messages, we achieve output privacy (as hinted at by Parno et al. although seemingly with two fewer decryptions than their brief description implies). We have shown that this blind verification could be used when a manager runs a pool of servers and rewards correct work – he needs to verify but is not entitled to learn the result. We have extended previous notions of security to fit our new definitional framework, introduced new models to capture additional threats (e.g. vindictive servers using revocation to remove competing servers), and provided a provably secure construction.

We believe that this work is a useful step towards making PVC practical and provides a natural set of baseline definitions from which to add future functionality. For example, in future work we will introduce an access control framework (using our scheme as a black box construction) to restrict the set of functions that clients may outsource, or to restrict (using the blind verification property) the set of verifiers that may learn the output. In this scenario, the KDC entity may, in addition to certifying servers and registering clients, determine access rights for such entities. The full version of this paper is available online [1].

References

1. J. Alderman, C. Janson, C. Cid, and J. Crampton. Revocation in publicly verifiable outsourced computation. Cryptology ePrint Archive, Report 2014/640, 2014. `http://eprint.iacr.org/`.
2. N. Attrapadung and H. Imai. Attribute-based encryption supporting direct/indirect revocation modes. In M. G. Parker, editor, *IMA Int. Conf.*, volume 5921 of *Lecture Notes in Computer Science*, pages 278–300. Springer, 2009.
3. H. Carter, C. Lever, and P. Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014*, pages 266–275. ACM, 2014.
4. S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *TCC*, pages 499–518, 2013.
5. R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In T. Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.
6. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
7. S. Goldwasser, S. D. Gordon, V. Goyal, A. Jain, J. Katz, F. Liu, A. Sahai, E. Shi, and H. Zhou. Multi-input functional encryption. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 578–602. Springer, 2014.

8. V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*, pages 89–98. ACM, 2006.
9. M. Green, S. Hohenberger, and B. Waters. Outsourcing the decryption of ABE ciphertexts. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.
10. B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In R. Cramer, editor, *TCC*, volume 7194 of *Lecture Notes in Computer Science*, pages 422–439. Springer, 2012.
11. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.

A PVC using KP-ABE

Parno et al. [10] provide a instantiation using *Key-policy Attribute-based Encryption*[7] (KP-ABE) [8], for Boolean functions. Define a universe $\mathcal{U}$ of n attributes and associate $V \subseteq \mathcal{U}$ with a binary n-tuple (the *characteristic tuple* of V) where the ith place is 1 if and only if the ith attribute is in V. Thus, there is a natural one-to-one correspondence between n-tuples and attribute sets; we write A_x to denote the set associated with x. A function $F : \{0,1\}^n \to \{0,1\}$ is monotonic if $x \leqslant y$ implies $F(x) \leqslant F(y)$, where $x = (x_1, \ldots, x_n)$ is less than or equal to $y = (y_1, \ldots, y_n)$ if and only if $\forall i, x_i \leqslant y_i$. For a monotonic F, the set $\mathbb{A}_F = \{x \in \{0,1\}^n : F(x) = 1\}$ defines a monotonic access structure. Informally, for a Boolean function F, the client generates a private key $SK_{\mathbb{A}_F}$ using the KeyGen algorithm.

Given an input x, a client encrypts a random message m "with" A_x using the Encrypt algorithm and publishes $VK_{F,x} = g(m)$ where g is a suitable one-way function (e.g. a pre-image resistant hash function). The server decrypts the message using the Decrypt algorithm, which will either return m (when $F(x) = 1$) or $\perp$.

The server returns m to the client. Any client can test whether the value returned by the server is equal to $g(m)$. Note, however, that a "rational" malicious server will always return $\perp$, since returning any other value will (with high probability) result in the verification algorithm returning a reject decision. Thus, it is necessary to have the server compute both F and its "complement" (and for both outputs to be verified).

Note that, to compute the private key $SK_{\mathbb{A}_F}$, it is necessary to identify all minimal elements x of $\{0,1\}^n$ such that $F(x) = 1$. There may be exponentially many such x. Thus, the initial phase is indeed computationally expensive for the client. Note also that the client may generate different private keys to enable the evaluation of different functions.

[7] If input privacy is required then a predicate encryption scheme could be used in place of the KP-ABE scheme.