Access Control in Publicly Verifiable Outsourced Computation

James Alderman[*], Carlos Cid[†], Jason Crampton[‡], and Christian Janson[§]

Information Security Group, Royal Holloway, University of London

Abstract

Publicly Verifiable Outsourced Computation (PVC) allows devices with restricted resources to delegate expensive computations to more powerful external servers, and to verify the correctness of results. Whilst this is highly beneficial in many situations, it also increases the visibility and availability of potentially sensitive data, and thus we may wish to limit the set of entities with access to input data and results. Additionally, within an organization it is extremely unlikely that every user would have uncontrolled access to all functionality. It is also not always reasonable to publish the results of a sensitive computation. Thus there is a need to apply access control mechanisms in PVC environments.

In this work, we define a new framework for Publicly Verifiable Outsourced Computation with Access Control (PVC-AC) that applies cryptographic enforcement mechanisms to address these concerns, and we provide a provably secure instantiation using Key Assignment Schemes. We also discuss example policies of interest in this setting.

1 Introduction

Increasingly mobile devices are being used as general computing devices. There is also a trend towards cloud computing and enormous volumes of data ("big data") meaning that computations may require considerable resources. In short, there is increasingly a discrepancy between computing resources of end-user devices and the resources required to perform complex computations on large datasets. This discrepancy, coupled with the rise of software-as-a-service, means there is a need for a client device to be able to delegate a computation to a server.

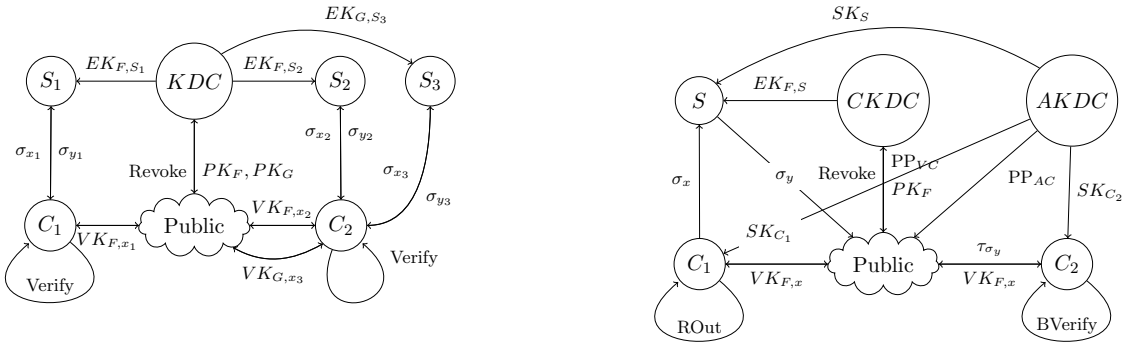
Consider, for example, a company that operates a "bring your own device" policy, allowing employees to use smartphones and tablets for work. Due to resource limitations, these devices may be unable to perform complex computations locally. Instead, a computation is outsourced over a network to a more powerful server (possibly outside the company, offering software-as-a-service, and untrusted) and the result of the computation is returned to the delegator. Another example arises in the context of battlefield communications where a squadron of soldiers is deployed with a reasonably light-weight computing device to 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.

[*]James.Alderman.2011@live.rhul.ac.uk
[†]Carlos.Cid@rhul.ac.uk
[‡]Jason.Crampton@rhul.ac.uk
[§]Christian.Janson.2012@live.rhul.ac.uk

Verifiable Outsourced Computation (VC), has recently attracted a lot of attention [13, 16, 8]. Alderman et al. [2] recently introduced Revocable Publicly Verifiable Outsourced Computation (RPVC) which includes a trusted Key Distribution Center (KDC) that is an authority on entities and performs expensive setup operations and key management duties. This leads to a more decentralised system architecture comprising a pool of delegators and a pool of servers, with the delegator not necessarily knowing the server chosen to perform a computation. Any delegator can submit a request for work (or job) to the server pool and some system-dependent mechanism allocates it to an available server (based on availability, suitability or on some bidding process if operating on a price per computation basis). This contrasts with prior models where the delegator chose a single server with whom to set up a VC system, and hence the server could be authenticated beforehand. Comparatively, then, delegators now have less control over the servers that may access their data or computation results. The RPVC model introduces Blind Verification such that anyone can verify correctness of a result, but only those holding an additional key may learn the value. This naturally lends itself to considering restrictions on who has access to this key and hence the output. A final consideration is to imagine a system (in the Manager model of [2]) that offers subscriptions to a computing platform, and the price paid corresponds to the set of available functions, time periods etc. The manager must ensure that only legitimate customers may access these tiers of service.

The contribution of this paper therefore is to define an access control framework providing greater control over the clients that may delegate a computation, the servers that may evaluate it, and the set of verifiers that may learn the output. This is motivated by the observation that within an organization it is extremely unlikely that all users have uncontrolled access to all functionality. In other words, some form of access control policy is required. In the context of PVC with Access Control (PVC-AC), this amounts to restricting those delegators that can outsource the computation of a function F to those that are authorized internally to compute F (if they had sufficient resources available). Practically, this means limiting those delegators that can prepare an input for delegation by insisting that the input is encrypted with a key appropriate to the function. Moreover, the server must be in possession of an appropriate key to decrypt the input and perform the computation. Thus we are controlling write access on the delegators and read access on the servers. In addition, we may wish to specify an access control policy for the output of a computation to restrict the set of clients that can verify correctness or read (learn the value of) the output. We can achieve this by encrypting the verification tokens with appropriate keys.

We define a new notion, PVC-AC, that extends the Attribute-based encryption RPVC scheme of Alderman et al. [2] to enforce graph-based access control policies. We provide a provably secure construction using Key Assignment Schemes (KASs). This provides a pragmatic blend of symmetric and asymmetric primitives where the symmetric KAS enables the efficient derivation of keys, each associated to a security label. Policies are specified in terms of these labels. As an authority on entities, the KDC sets up the KAS and issues keys to each entity corresponding to the highest security label for which they are authorised, and they may derive any lower keys required. These keys may cryptographically protect input values and verification keys such that only entities satisfying the specified policies may operate on them. Unauthorised servers or eavesdropper will not even learn the input values, thus exposure of data is limited to explicitly authorised (trusted) entities. Moreover, this restriction enforces access control policies on the part of the delegator device. To encrypt the inputs with a symmetric key associated with the policy, the delegator must be able to derive this key in the first place. Thus, the delegator's security clearance must be at least the classification of the function in order to derive this key from that issued by the KDC. Hence delegators may only outsource the evaluation of problems that they could (resources permitted) compute themselves.



(a) The operation of RPVC

(b) The operation of PVC-AC

Figure 1: The operation of RPVC and PVC-AC

We begin by briefly reviewing related work before giving a formal definition for PVC-AC, discussing relevant access control policies, introducing new notions of security that arise in this setting and finally giving an overview of our construction. The paper is self-contained: full details of the background, security games, instantiation and full security proofs can be found in the appendices.

We use the following notation.We write $y \leftarrow A(\cdot)$ for the action of running a probabilistic algorithm A on given inputs and assigning the result to an output y. We use PPT to denote probabilistic polynomial-time and say that $\text{negl}(\cdot)$ is a negligible function of its input. $\mathcal{A}^{\mathcal{O}}$ is used to denote the adversary $\mathcal{A}$ being provided with oracle access.

2 Background

2.1 Revocable Publicly Verifiable Outsourced Computation

A RPVC scheme comprises the algorithms RPVC.Setup, RPVC.FnInit, RPVC.Register, RPVC.Certify, RPVC.ProbGen, RPVC.Compute, RPVC.BlindVerify, RPVC.RetrieveOutput and RPVC.Revoke which correspond to the following steps, and as in Figure 1a:

- The KDC generates public parameters, issues personalised secret keys, and evaluation keys to servers and publishes function delegation information.
- To outsource the evaluation of $F(x)$, a delegator C, sends an encoded input σ_x to a server S, and publishes verification tokens for the computation.
- S uses σ_x and an evaluation key for F to produce an encoded output (sent to C, the manager, or published depending on the system architecture).
- Any entity can use the verification token to blind verify correctness of the output. The verifier may not learn the value of $F(x)$ if not in possession of the retrieval key. If S cheated they may report S to the KDC for revocation.
- If blind verification was successful, a party possessing the retrieval key $RK_{F,x}$ can recover $F(x)$.
- The KDC may revoke a cheating server to prevent it computing F in the future (and hence from receiving any reward for future work).

2.2 Key Assignment Schemes

A *partially ordered set* (*poset*) is a set L equipped with a reflexive, anti-symmetric and transitive binary relation $\leqslant$. Let U be a set of entities, O be a set of resources to be protected, and $(L, \leqslant)$ be a poset of security labels. Let $\lambda : U \cup O \to L$ be a labelling function assigning a security label to each entity and object. The tuple $(L, \leqslant, U, O, \lambda)$ then denotes an *information flow policy*. The policy requires that an entity $u \in U$ may read an object $o \in O$ if and only if $\lambda(u) \geqslant \lambda(o)$.

A *Key Assignment Scheme* (KAS) [1] provides a generic, cryptographic enforcement mechanism for such policies in which a unique cryptographic key is associated to each node (representing a security label) in $(L, \leqslant)$. A KAS eases the problem of key distribution by allowing a trusted center to distribute a single key to each entity, who may combine this with public information to derive additional keys. A well-known KAS construction (known as an *iterative key encrypting* (IKE) KAS [12]) publishes encrypted keys. In particular, for each $y < x$, such that no z exists with $y < z < x$, $\mathsf{Encrypt}_{\kappa_x}(\kappa_y)$ is published. Then for any $x > y$, the key for each node on a path from x to y can be derived (in an iterative fashion) by an entity that knows κ_x. For more detail, see Appendix A.1.

3 Definition of PVC-AC

Here we define the PVC-AC notion to enforce graph-based access control policies over delegators, servers and verifiers. Generally, we wish to impose restrictions on three activities. First, we wish to specify a (write) policy that determines the functions a client is authorised to delegate – this means restricting the inputs a client may correctly encrypt. Second, we specify a (read) policy that determines the functions a server may compute – that is, ciphertexts he may correctly decrypt. Lastly, we specify a (read) policy dictating the function outputs a verifier may read – again corresponding to decrypting. We use symmetric cryptographic primitives to provide the enforcement mechanism.

More specifically, for the first two cases, we associate each delegator C and server S with a set of functions $\lambda(C)$ and $\lambda(S)$ respectively. C is authorized to delegate the computation of $F(x)$ if $F \in \lambda(C)$; similarly, S may compute $F(x)$ if $F \in \lambda(S)$. We associate each function F with a singleton set $\lambda(F) = \{F\}$ and a key κ_F, and ensure that C can derive κ_F for all $F \in \lambda(C)$ (and similarly for S). This *correctness* criterion allows an authorized entity to encrypt or decrypt using $\lambda(F)$. The *security* criterion requires that a set of unauthorized entities cannot collude, and can be realised by a KAS. Verification policies work similarly with a set of functions $\lambda(V)$. We denote computation policies over delegators and servers by $\lambda^C(\cdot)$, and verification policies over delegators and verifiers by $\lambda^V(\cdot)$. When clear from the context, we simply use λ. Finally, $\mathcal{P}_C$ and $\mathcal{P}_V$ denote the poset encoding the computation and verification policy respectively.

We now give the definition of PVC-AC. We extend the functionality of the KDC from [2] to grant access control credentials to delegators, servers and verifiers. We may split the responsibilities between two KDCs: a *computation* KDC (CKDC) that generates function keys, and an *authorization* KDC (AKDC) that manages access control policies. The AKDC could be a trusted institution with details of entities such that it can grant relevant permissions and capabilities.

Definition 1. *A* Publicly Verifiable Outsourced Computation Scheme with Access Control (PVC-AC) *comprises the following algorithms:*

- $\mathsf{Setup}(1^\lambda) \to (\mathrm{PP}, \mathrm{MK})$: Comprises two subalgorithms:
  - $\mathsf{SetupVC}(1^\lambda) \to (\mathrm{PP}_{VC}, \mathrm{MK}_{VC})$: Run by CKDC to establish public parameters PP_{VC} and a master secret key MK_{VC} for PVC functionality.

  - $\mathsf{SetupAC}(1^\lambda, \mathcal{P}_C, \mathcal{P}_V) \rightarrow (\mathrm{PP}_{AC}, \mathrm{MK}_{AC})$: Run by AKDC to establish public parameters and a master secret to enforce access control policies.

- $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F) \rightarrow (PK_F, L_F)$: Run by the CKDC to generate a delegation key, PK_F, for a function F as well as a list L_F of available servers for evaluating F, which is initially empty.

- $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathrm{ID}, \lambda(\mathrm{ID})) \rightarrow \mathrm{SK}_{\mathrm{ID}}$: Run by the AKDC to generate a personalised key SK_{ID} for an entity with identifierID[1]. It outputs a secret key granting rights for the label $\lambda(\mathrm{ID})$ according to the computation or verification policy.

- $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S) \rightarrow (EK_{F,S}, L_F)$: CKDC generates an evaluation key $EK_{F,S}$ for a function F and server S, and S is added to L_F.

- $\mathsf{ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(F), \lambda^V(C), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}) \rightarrow (\sigma_x, VK_{F,x}, RK_{F,x})$: Run by a delegator C to outsource the computation of $F(x)$ to a server S. C may do so only if it satisfies the computation policy – that is $\lambda^C(C) \geqslant \lambda^C(F)$. It outputs an encoded input σ_x, a public verification key $VK_{F,x}$ to verify correctness, and an output retrieval key $RK_{F,x}$ which verifiers satisfying $\lambda^V(V) \geqslant \lambda^V(F)$ may use to read $F(x)$.

- $\mathsf{Compute}(\sigma_x, EK_{F,S}, SK_S, \lambda^C(S), \lambda^C(F), PP_{AC}) \rightarrow \sigma_y$: Run by a server S holding an evaluation key $EK_{F,S}$, SK_S and an encoded input σ_x to evaluate $F(x)$ and output an encoding, σ_y, of the result. This succeeds if and only if $\lambda^C(S) \geqslant \lambda^C(F)$, that is S satisfies the computation policy for this evaluation.

- $\mathsf{Verify}(\sigma_y, SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}) \rightarrow (\tilde{y}, \tau_{\sigma_y})$: Verification consists of two steps:

  - $\mathsf{VC.BlindVerify}(\mathrm{PP}_{VC}, \sigma_y, VK_{F,x}, L_F) \rightarrow (\mu, \tau_{\sigma_y})$: Run by *any* verifier to produce an intermediate output μ and a token $\tau_{\sigma_y} = (\mathsf{accept}, S)$ for a correct result, or $\tau_{\sigma_y} = (\mathsf{reject}, S)$ to signify a cheating server S.
  - $\mathsf{VC.RetrieveOutput}(SK_V, \mu, \tau_{\sigma_y}, VK_{F,x}, RK_{F,x}, \lambda^V(V), \lambda^V(F)) \rightarrow \tilde{y}$: Run by verifiers V in possession of the output retrieval key $RK_{F,x}$ and the output μ from $\mathsf{BlindVerify}$. If $\lambda^V(V) \geqslant \lambda^V(F)$ then V should be able to read the actual result $\tilde{y} = F(x)$ or $\perp$.

- $\mathsf{Revoke}(\mathrm{MK}_{VC}, \tau_{\sigma_y}, F, L_F) \rightarrow (\{EK_{F,S'}\}, L_F)$ or $\perp$: Run by the CKDC if a verifier reports a misbehaving server i.e. that Verify returned $\tau_{\sigma_y} = (\mathsf{reject}, S)$ (if $\tau_{\sigma_y} = (\mathsf{accept}, S)$ then this algorithm should output $\perp$). It revokes the evaluation key $EK_{F,S}$ of the server S thereby preventing any further evaluations of F. This is achieved by removing S from L_F (the list of servers for F) and issuing updated evaluation keys $EK_{F,S'}$ to all servers $S' \neq S$.

We say that a PVC-AC scheme is *correct* if for all functions $F \in \mathcal{F}$, inputs x, honestly generated parameters, and honestly registered entities C, S and V (delegator, certified server and verifier respectively)such that $\lambda^C(C), \lambda^C(S) \geqslant \lambda^C(F)$ and $\lambda^V(C), \lambda^V(V) \geqslant \lambda^V(F)$, if S honestly runs $\mathsf{Compute}$ for F on an encoding of x generated by C, then V running Verify on the output from S will almost certainly output accept and $F(x)$. That is, if all algorithms are run honestly by authorised parties then the verifier should almost certainly accept. More formally we can write

[1]In future algorithms this will be S, C or V to denote a server, delegator or verifier respectively.

Definition 2 (Correctness)**.** *A Publicly Verifiable Outsourced Computation Scheme with Access Control (PVC-AC) is* correct *for a family of functions $\mathcal{F}$ if for all functions $F \in \mathcal{F}$ and inputs x, where* $\mathrm{negl}(\cdot)$ *is a negligible function of its input:*

$$
\begin{aligned}
&\Pr[\mathsf{Setup}(1^\lambda) \rightarrow (\mathrm{PP} = (\mathrm{PP}_{VC}, \mathrm{PP}_{AC}), \mathrm{MK} = (\mathrm{MK}_{VC}, \mathrm{MK}_{AC})),\\
&\quad \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F) \rightarrow (PK_F, L_F),\\
&\quad \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathrm{C}, \lambda(\mathrm{C})) \rightarrow \mathrm{SK}_\mathrm{C},\\
&\quad \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathrm{S}, \lambda(\mathrm{S})) \rightarrow \mathrm{SK}_\mathrm{S},\\
&\quad \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathrm{V}, \lambda(\mathrm{V})) \rightarrow \mathrm{SK}_\mathrm{V},\\
&\quad \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S) \rightarrow (EK_{F,S}, L_F),\\
&\quad \mathsf{ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}) \rightarrow (\sigma_x, VK_{F,x}, RK_{F,x}),\\
&\quad \mathsf{Verify}(\mathsf{Compute}(\sigma_x, EK_{F,S}, SK_S, \lambda^C(S), \lambda^C(F), PP_{AC}),\\
&\quad SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}) \rightarrow (F(x), (\mathsf{accept}, S))]\\
&\quad = 1 - \mathrm{negl}(\lambda),
\end{aligned}
$$

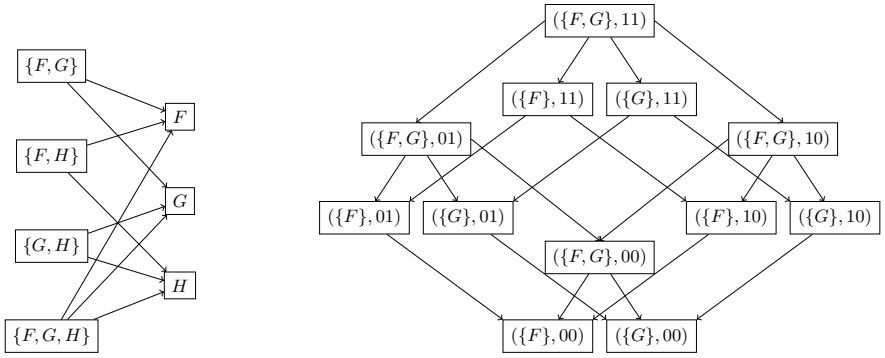
where $\lambda^C(C) \geqslant \lambda^C(F), \lambda^V(C) \geqslant \lambda^V(F), \lambda^C(S) \geqslant \lambda^C(F)$ and $\lambda^V(V) \geqslant \lambda^V(F)$ holds.

4 Policies

In this section we discuss the type of access control policies that may be of interest in an outsourced computation environment. We begin by examining simple policies over the choice of functions, before considering more fine-grained policies over sets of input values and other security posets. We also discuss policies to protect function outputs in the second part of this section. For ease of notation, we use λ to denote λ^C or λ^V if it is clear from the context. The policies we consider are graph-based policies, where "objects" to be protected are not data files, as in traditional access control policies, but function evaluations to be outsourced. The "user" population comprises the sets of delegators, $\mathcal{C}$, and computational servers, $\mathcal{S}$. We define a security function $\lambda : \mathcal{C} \cup \mathcal{S} \cup \mathcal{F} \rightarrow L$ where $\mathcal{F}$ is the family of functions that may be outsourced and $(L, \leqslant)$ is a poset of security labels. This function assigns a label from L to each delegator, server and function in the PVC-AC system, which represents the security classification of these entities and functions. Each delegator, C, is issued with a symmetric key corresponding to $\lambda(C)$ by the AKDC. Each server, S, is issued with two keys: one is the symmetric key issued by the AKDC and associated with $\lambda(S)$, enabling the server to derive further symmetric keys and decrypt delegated inputs; the second is a private key for a particular function, issued by the CKDC, which it uses to perform a computation. We restrict our focus to graph-based policies, as these have been shown to encompass many notions of access control that are desirable in practice including information flow policies, role-based access control and attribute-based access control [10].

4.1 Policies Over Functions

First note that in a RPVC scheme, the KDC implicitly provides some access control in that it certifies servers to perform specific functions by generating evaluation keys. However, there is no access control applied to delegators – *any* entity can outsource an evaluation of *any* function for which the KDC has published delegation information (essentially because asymmetric cryptographic primitives are used). In particular, a delegator may request a computation that the delegator itself is not authorised to perform.



(a) $L = (2^{\{F,G,H\}}, <)$ (b) $L = (2^{\{F,G\}} \times \{0,1\}^2, <)$

Figure 2: Example Posets

A simple access control policy we may enforce is to define the set of security labels L to be $2^{\mathcal{F}}$ (the power set of functions to be outsourced). Then, $\lambda(C) \subseteq \mathcal{F}$ defines the set of functions that a delegator C may outsource the evaluation of, $\lambda(S) \subseteq \mathcal{F}$ denotes the set of functions that a server S may compute, and $\lambda(F) = \{F\}$ denotes the label of the function $F \in \mathcal{F}$. Then, for any $x, y \in L$ we define an order relation $<$ such that $x < y$ if and only if $x \in \mathcal{F}, y \subseteq \mathcal{F}$ and $x \in y$. The corresponding Hasse diagram with $\mathcal{F} = \{F, G, H\}$ is shown in Figure 2a[2].

To outsource the computation of $F(x)$, C encrypts the encoding of x using the key κ_F. C can derive κ_F for this purpose if and only if $\lambda(C) \geqslant \lambda(F)$ i.e. if and only if $F \in \lambda(C)$. To compute $F(x)$, S decrypts the encrypted input value using the corresponding key κ_F. As before, S may do this if and only if $F \in \lambda(S)$. Note that as the input values are encrypted using a symmetric key chosen in accordance with the access control policy, only authorized servers may learn the input data. Any unauthorised server may not derive κ_F, and by the (IND-CPA) security of the encryption scheme, will learn nothing about the data.

4.1.1 Policies over Function Inputs.

In addition to limiting the set of functions that may be delegated and computed, we may wish to implement a more fine-grained access control policy determined by input values to functions. To do so, we redefine the security function such that "objects" are now considered to be pairs (F, x) where $F \in \mathcal{F}$ and $x \in \mathsf{Dom}(F)$. For ease of exposition we assume that for all $F \in \mathcal{F}$, the domain of F is $\mathsf{Dom}(F) = \{0,1\}^n$ for a positive, non-zero integer n.[3] Then, $\lambda : \mathcal{C} \cup \mathcal{S} \cup (\mathcal{F} \times \{0,1\}^n) \to L$.

We could, for example, let L be $2^{\mathcal{F}} \times \{0,1\}^n$. We assume a co-ordinatewise ordering on $\{0,1\}^n$: that is $(x_1, \ldots, x_n) \leqslant (y_1, \ldots, y_n)$ if and only if $x_i \leqslant y_i$ for all i. Then for two labels $(\mathcal{G}, x)$ and $(\mathcal{G}', x')$ in $2^{\mathcal{F}} \times \{0,1\}^n$, $(\mathcal{G}, x) \leqslant (\mathcal{G}', x')$ if and only if $\mathcal{G} = \{F\}$ for some $F \in \mathcal{F}$, $F \in \mathcal{G}'$ and $x \leqslant x'$. This poset could be used to ensure delegators and servers operate over a limited range of data values. Specifically, for a delegator or server z, we define $\lambda(z) = (\mathcal{G}, x)$, where $\mathcal{G} \subseteq \mathcal{F}$ and $x \in \{0,1\}^n$, and $\lambda(F) = (\{F\}, 0^n)$. Then C is authorized to outsource the computation of $G(y)$ for all $G \in \mathcal{G}$ and all $y \leqslant x$. The Hasse diagram for this poset with $\mathcal{F} = \{F, G\}$ and $n = 2$ is shown in Figure 2b.

Alternatively, we may let L be the power set $2^{\mathcal{F} \times \{0,1\}^n}$. This enables each function to be associated with different range of allowable inputs. Then, for two labels $\{(F_1, x_1), \ldots, (F_m, x_m)\}$ and $\{(F'_1, x'_1), \ldots, (F'_{m'}, x'_{m'})\}$,

$$\{(F_1, x_1), \ldots, (F_m, x_m)\} \leqslant \{(F'_1, x'_1), \ldots, (F'_{m'}, x'_{m'})\}$$

[2]We have not included nodes for the empty set in these posets.

[3]If this is not the case then we can define $n = \max_{F \in \mathcal{F}} |\mathsf{Dom}(F)|$ and then redefine all functions F with $\mathsf{Dom}(F) < n$ to satisfy the required property by, for example, adding fixed points $F(x) = x$ for all $x \in \{0,1\}^n \setminus \mathsf{Dom}(F)$.

if and only if $m = 1$ (the label is a single pair), $F_1 = F'_i$ for some i and $x_1 \leqslant x'_i$.

We remark that the first choice of poset presented in this section roughly corresponds to the idea of assigning a possible data range to each delegator and then authorizing them to outsource a set of functions on that data, whilst the second is more akin to authorizing delegators to outsource particular functions and then assigning each function a set of authorized input values. Finally, we note that the choice of $L = 2^{\mathcal{F}} \times 2^{\{0,1\}^n}$ extends this second case to a situation where each function can be associated with arbitrary sets of input values rather than just contiguous intervals.

4.1.2 Enforcing General Graph-based Policies.

We have seen how sets of functions and inputs can be mapped to a graph-based access control policyto restrict the functions a delegator may outsource and a server may compute. We now briefly discuss how to incorporate additional security labels that rely on the environment or external access control policies. We can use any poset of security labels L to classify each outsourced computation. For example, define $M = \{\texttt{Top-Secret}, \texttt{Secret}, \texttt{Classified}, \texttt{Unclassified}\}$ to be the Bell-LaPadula clearance levels [6] and K to be a set of need-to-know categories. Then we can enforce the security function $\lambda : \mathcal{C} \cup \mathcal{S} \cup \mathcal{F} \to L = M \times 2^K$ where K specifies the nature, and M specifies the sensitivity, of the function F. Then, to delegate or compute $F(x)$, we require that the entity's clearance level is at least the classification of the function: $\lambda(C) \geqslant \lambda(F)$. Similarly L could be a set of roles for a Role-based Access Control Policy [10].

We could also add additional criteria in the mapping to security labels. For example, we could add time intervals $\mathcal{T}$ as in Figure 4c by setting $\lambda : \mathcal{C} \cup \mathcal{S} \cup (\mathcal{F} \times \mathcal{T}) \to L$. We could then set $L = 2^{\mathcal{F} \times \mathcal{T}}$ as in the previous section, or if $L = M$ as above, then certain functions could be classified higher during certain times of the day. In place of a temporal poset $\mathcal{T}$ we could also apply a geo-spatial poset to classify function evaluations differently depending on location data e.g. a function may be more sensitive if being outsourced from a battlefield as opposed to within a secured building. Finally, we could extend the policies over function inputs to include characteristics of the input data. Let Z be a set of labels describing data types that functions may be computed over, and define $L = 2^{\mathcal{F} \times (\{0,1\}^n \times Z)}$ for example. This enables the same input to the same function to be classified differently and hence to require different authorisation from the delegator and the server. As a motivating example, consider a summation function over the integers. The semantic meaning of the integers in question may determine the overall classification of this computation – if the integers are populations of cities then this may not be classified at all, but troop deployments in different regions may be much more sensitive.

4.2 Policies over Outputs

It is also useful to apply access control to the act of verifying the results of a computation and in particular learning the output. Clearly, when considering computations over confidential data, it is not always appropriate to publish the results. As a trivial example, the outsourcing of the identity function could be used to "legitimately" leak confidential data in an unsecured PVC setting.

In general it may be considered that any verifier should at least have the access rights of the delegating or computing entity, that is $\lambda(C), \lambda(S) \leqslant \lambda(V)$. This encompasses the "no write down" property of traditional access control policies. Alternatively, the AKDC may decide that, for certain computations, the results are not as highly classified as the input data, or indeed the act of computing it. For example, the results of a computation over classified data may be included in a public document, despite the input data remaining classified The encoded output from the computation may be published with the verification key such that recipients

can verify the legitimacy of the published values. It may also be reasonable to expect that only authorised and trusted reviewers of the document should be able to verify, or that some (lower) form of clearance is still required to access the result. It may be the case that we require some level of trust (demonstrated through possession of a relevant key) in the verifier. The scheme of Alderman et al. [2] allowed for separate blind verification (to ensure accuracy) and output retrieval (to learn results). Here, we assume that all entities are able to blind verify a result, but the set of entities that may learn the actual output should be restricted[4].

4.2.1 Published Verification Token.

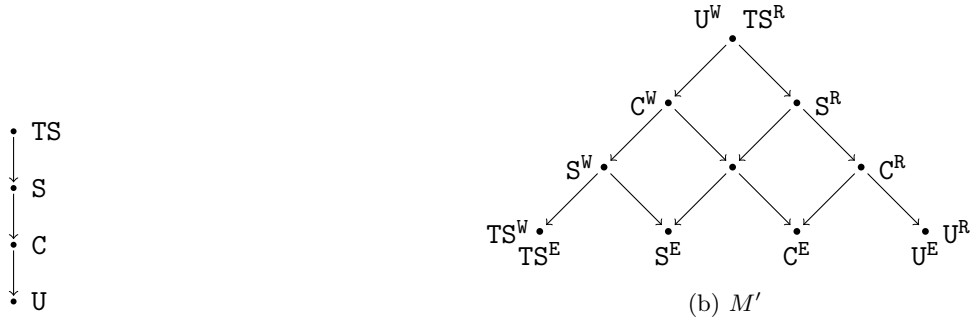
In the case of publishing a verification tokens, we can use similar posets and security functions as in Section 4. The user population is the set of delegators $\mathcal{C}$ and verifiers $\mathcal{V}$, and objects are encoded outputs. The security function is defined to be $\lambda^V : \mathcal{V} \cup \mathcal{C} \cup \mathcal{F} \to L$, where $\mathcal{F}$ is the family of functions to be outsourced and $(L, \leqslant)$ is a poset of security labels. Each verifier, V, is given the symmetric key $\kappa_{\lambda^V(V)}$ corresponding to $\lambda^V(V)$. Each delegator, C, is given $\kappa_{\lambda^V(C)}$. When encoding an input, C generates additional information $RK_{F,x}$ that enables output to be retrieved given the encoded output and the verification key. C will derive the required key for reading the output, $\kappa_{\lambda^V(F)}$ and use this to symmetrically encrypt $RK_{F,x}$. Note that C must at least have clearance to verify the result of any computation he outsources due to the use of a symmetric enforcement mechanism. This is reasonable since the delegator must have the right to compute any function that he outsources and hence (resources permitting) could certainly learn the result. As before we can extend the definition of the security function and of L to accommodate more fine-grained policies over outputs.

4.2.2 Enforcement of No Write Down Policies.

No Write Down is an important requirement of many access control policies. It means that an entity C may not write (encrypt) an object to a classification level $\lambda(o) < \lambda(C)$ i.e. write to a lower security level, as this could constitute a leak of classified data. It is, of course, possible in our setting for the writer to encrypt the object at his current clearance level and for any readers with higher access rights to derive the lower key to decrypt and read. However, we may want to protect a result at a higher classification level (write up) e.g. preparing a report for a manager. We now discuss two methods to achieve this. The first is straightforward from the prior sections. Consider a computation poset $\mathcal{P}_C = (L, \leqslant)$ and, by inverting the order relation, construct the verification poset $\mathcal{P}_V = (L, \geqslant)$ i.e. $x \leqslant y$ in $\mathcal{P}_V$ if and only if $y \leqslant x$ in $\mathcal{P}_C$. Then if an (independent) KAS is instantiated over each poset and a delegator is provided with corresponding keys in each, i.e. $\lambda^C(C) = \lambda^V(C)$, then key derivation allows the derivation of keys κ_i for $\lambda^C(i) \leqslant \lambda^C(C)$ and $\lambda^V(C) \geqslant \lambda^C(i)$.

An alternative solution (only applicable in some situations) requires a single poset. We begin with the relatively strong assumption that authorised writers of classified objects are also allowed to read objects at that level, and that there is a separation of duty between delegators and verifiers. As a motivating example, consider a PVC-AC scheme implemented by a company to use third-party computation servers. Then it may be the case that the access control policies should enforce privacy from the external servers and external verifiers, but delegators within the company may verify their colleagues' results. It may also occur that, due to the particular RPVC scheme employed (built on predicate encryption for example), access to the encoded input does not leak any sensitive information. For ease of explanation, let us consider only a totally ordered chain M of security labels such as the Bell-LaPadula chain, illustrated in Figure

[4]Note, we can use the same techniques to protect $VK_{F,x}$ to restrict blind verification.



(a) M (b) M'

Figure 3: M and it's corresponding interval based poset

3a. For each label $\mathtt{x} \in M$, we define a new label $\mathtt{x}^{\mathtt{E}}$ to denote an "encryption label". We then instantiate a KAS over the triangular poset $M' = T_{|M|}$ with this set of new labels as the leaf nodes, as shown in Figure 3b. We can then use this poset to differentiate between write policies and read policies over M, with labels denoted $\mathtt{x}^{\mathtt{W}}$ and $\mathtt{x}^{\mathtt{R}}$ respectively. To grant write access for a label $\mathtt{x} \in M$, we provide the entity with $\kappa_{\mathtt{x}^{\mathtt{W}}}$ which enables the derivation of keys $\kappa_{\mathtt{y}^{\mathtt{E}}}$ for all $\mathtt{y} \geqslant \mathtt{x}$ corresponding to writing up the chain of security labels. Conversely, to grant read access for $\mathtt{x} \in M$, we issue $\kappa_{\mathtt{x}^{\mathtt{R}}}$ which only enables derivation of $\kappa_{\mathtt{y}^{\mathtt{E}}}$ for all $\mathtt{y} \leqslant \mathtt{x}$ corresponding to reading down the chain of security labels. All objects are encrypted and decrypted with the set of encryption keys $\mathtt{x}^{\mathtt{E}}$ for $\mathtt{x} \in M$. Delegators are given *only* writing keys and verifiers are given *only* reading keys.

5 Security Definitions

We now introduce several security models capturing requirements of PVC-AC. Notions of Public Verifiability, Revocation and Vindictive Servers follow naturally from [2] with additional inputs for policy declarations. However, as these do not rely on the access control properties introduced in this work, the proofs follow. Additionally, we are concerned with preventing unauthorized entities from performing restricted actions or learning restricted information. We use o to denote an outsourced computation (including descriptors such as the function, input and environmental factors) such that $\lambda(o)$ encompasses the necessary clearance to operate on this evaluation. The notation $\mathcal{A}^{\mathcal{O}}$ denotes $\mathcal{A}$ being given oracle access to the functions $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$. Aside from $\mathsf{Register}$, each oracle simply runs the relevant algorithm. With the exception of Authorized Verification, each of these games uses the $\mathsf{Register}$ oracle specified in Oracle Query 1. Authorized Verification uses the modified $\mathsf{Register}$ oracle given in Oracle Query 2 which compares to the verification policy instead of the computation policy.

5.1 Authorized Outsourcing

In Game 1 we formalize the notion that a delegator may not outsource any computation for which he is not authorized – that is, any computation o where $\lambda^C(o) \not\leqslant \lambda^C(C)$. Clearly, within our framework we cannot make any guarantees about what an entity may do externally. For instance, we cannot enforce that a client does not perform any computation using an external server that is not subject to these controls, however we can enforce that in order to use the provided functionality (i.e. to contract an available server registered in this system) they must be able to prove authorisation. This seems to be a reasonable assumption, taking into consideration organisational boundaries, and any external control should perhaps be enforced

Game 1 $\mathbf{Exp}_{\mathcal{A}}^{AutOut}\left[\mathcal{PVC}_{\mathrm{AC}}, F, 1^{\lambda}\right]$:

$L = \epsilon, o = \perp$
$(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}) \leftarrow \mathsf{SetupVC}(1^{\lambda})$;
$(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}) \leftarrow \mathsf{SetupAC}(1^{\lambda}, \mathcal{P}_C, \mathcal{P}_V)$;
$\lambda^C(\mathcal{A}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$SK_{\mathcal{A}} \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathcal{A}, \lambda^C(\mathcal{A}))$;
$L = L \cup \lambda^C(\mathcal{A})$;
$o \leftarrow \mathcal{A}^{\mathcal{O}}(\lambda^C(\mathcal{A}), SK_{\mathcal{A}}, \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
for all $\lambda^C(v_i) \in L$ **do**
  if $\lambda^C(o) \leqslant \lambda^C(v_i)$ **then**
    return 0
  end if
end for
$SK_S \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, S, \lambda^C(S))$ *s.t.* $\lambda^C(S) \geqslant \lambda^C(o)$;
$SK_V \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, V, \lambda^V(V))$ *s.t.* $\lambda^V(V) \geqslant \lambda^V(o)$;
$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F)$;
$(EK_{F,S}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S)$;
$(\sigma_x, VK_{F,x}, RK_{F,x}) \leftarrow \mathcal{A}^{\mathcal{O}}(o, \lambda^C(o), \lambda^C(\mathcal{A}), \lambda^V(o), \lambda^V(\mathcal{A}), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, PK_F, L_F)$;
$\sigma_y \leftarrow \mathsf{Compute}(\sigma_x, EK_{F,S}, SK_S, \lambda^C(S), \lambda^C(o), \mathrm{PP}_{AC})$;
$(\tilde{y}, \tau_{\sigma_y}) \leftarrow \mathsf{Verify}(\sigma_y, SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(o), \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
If $((\tilde{y}, \tau_{\sigma_y}) \neq (\perp, (\mathsf{reject}, \mathcal{A})))$
  Return 1
Else Return 0

Oracle Query 1 $\mathcal{O}^{\mathsf{Register}}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, ID, \lambda(ID))$:

if $(o \neq \perp) \wedge \lambda(ID) \geqslant \lambda^C(o)$ **then**
  return $\perp$
end if
$L = L \cup \lambda(ID)$;
return $\mathsf{Register}(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}, ID, \lambda(ID))$

by limiting external communication channels (e.g. using a strict firewall). Similarly, we cannot enforce that an entity does not share key content with other entities, but we do ensure that such collusion does not enable access that either entity alone could not access. Additionally, due to the revocation functionality, it may be undesirable for a server to share key material as he must trust the additional server not to cheat.

The game proceeds as follows. First the challenger runs the setup procedures for the scheme and registers the adversary for the adversary's choice of security label. The adversary is given the public information, his secret key and oracle access, and outputs a choice of outsourced computation o to attack, including the function and input, F and x. We require that no security label that $\mathcal{A}$ has queried to the Register oracle may allow the derivation of a key for $\lambda^C(o)$ as this would be a trivial win, and we return 0 since the adversary has not managed to find a reasonable attack target. We also require that any ID that $\mathcal{A}$ queries to its oracle must previously have been queried to the Register oracle (which is in keeping with realistic operation). $\mathcal{C}$ sets up keys for F and registers two entities that are authorized to perform the computation and verification for o respectively. The adversary is then challenged, given all information that a real attacker may learn and oracle access, to output an encoded input that the Compute and Verify algorithms will accept – that is, an unauthorized adversary must produce an input that is accepted and computed on by honest entities.

Definition 3. *The* advantage *of a PPT adversary $\mathcal{A}$ making a polynomial number of queries q in the Authorized Outsourcing Experiment is defined as follows:*

$$Adv_{\mathcal{A}}^{\mathbf{AutOut}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathbf{AutOut}}[\mathcal{PVC}_{AC}, F, 1^{\lambda}] = 1]$$

A PVC-AC is secure against Authorized Outsourcing *for a function F, if for all PPT adver-*

saries $\mathcal{A}$, $Adv_{\mathcal{A}}^{\mathbf{AutOut}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) \leqslant \mathrm{negl}(\lambda)$.

5.2 Authorized Computation

Authorized Computation, in Game 2, proceeds similarly to Authorized Outsourcing and captures the notion that a computational result should only be considered valid if generated by an authorized party. The challenger sets up the system and registers the adversary for its choice of security label. $\mathcal{A}$ then chooses a target computation that it is not authorized to compute by *any* of the keys it holds. The challenger now simulates two entities: a delegator and a verifier that are authorized for this computation and certifies $\mathcal{A}$ as a server for F. The challenger creates an encoded input by running ProbGen on the adversary's target computation and gives this to $\mathcal{A}$ who must output an encoded output that is accepted by the verifier. Note that although the adversary chooses the input to the computation, and therefore can certainly compute $F(x)$, it still should not be able to convince the verifier to accept.

Game 2 $\mathbf{Exp}_{\mathcal{A}}^{AutComp}\left[\mathcal{PVC}_{\mathrm{AC}}, F, 1^{\lambda}\right]$:

$L = \epsilon, o = \perp$
$(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}) \leftarrow \mathsf{SetupVC}(1^{\lambda})$;
$(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}) \leftarrow \mathsf{SetupAC}(1^{\lambda}, \mathcal{P}_C, \mathcal{P}_V)$;
$\lambda^C(\mathcal{A}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$SK_{\mathcal{A}} \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathcal{A}, \lambda^C(\mathcal{A}))$;
$L = L \cup \lambda^C(\mathcal{A})$;
$o \leftarrow \mathcal{A}^{\mathcal{O}}(\lambda^C(\mathcal{A}), SK_{\mathcal{A}}, \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
for all $\lambda^C(v_i) \in L$ **do**
  if $\lambda^C(o) \leqslant \lambda^C(v_i)$ **then**
    return 0
  end if
end for
$SK_C \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, C, \lambda^C(C))$ *s.t.* $\lambda^C(C) \geqslant \lambda^C(o)$;
$SK_V \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, V, \lambda^V(V))$ *s.t.* $\lambda^V(V) \geqslant \lambda^V(o)$;
$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F)$;
$(EK_{F,\mathcal{A}}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, \mathcal{A})$;
$(\sigma_x, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(o), \lambda^V(C), \lambda^V(o), \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$\sigma_y \leftarrow \mathcal{A}^{\mathcal{O}}(\sigma_x, VK_{F,x}, o, \lambda^C(o), \lambda^C(\mathcal{A}), EK_{F,\mathcal{A}}, SK_{\mathcal{A}}, \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, PK_F, L_F, RK_{F,x})$;
$(\tilde{y}, \tau_{\sigma_y}) \leftarrow \mathsf{Verify}(\sigma_y, SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(o), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, RK_{F,x})$;
If $((\tilde{y}, \tau_{\sigma_y}) \neq (\perp, (\mathsf{reject}, \mathcal{A})))$
  Return 1
Else Return 0

Definition 4. *The* advantage *of a PPT adversary* $\mathcal{A}$ *making a polynomial number of queries* q *in the Authorized Computation Experiment is defined as follows:*

$$Adv_{\mathcal{A}}^{\mathbf{AutComp}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathbf{AutComp}}[\mathcal{PVC}_{AC}, F, 1^{\lambda}] = 1]$$

A PVC-AC is secure against Authorized Computation *for a function* F*, if for all PPT adversaries* $\mathcal{A}$*,*

$$Adv_{\mathcal{A}}^{\mathbf{AutComp}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) \leqslant \mathrm{negl}(\lambda).$$

5.3 Authorized Verification

In Game 3 we capture the notion that an unauthorized verifier should not be able to learn the output of the computation (in this section we are referring to verifiers performing the BlindVerify step only, and in particular not in possession of the output retrieval key). The game begins similarly to before, except here the adversary is not permitted to choose the target computation. Clearly if the adversary chose the input values then it can win trivially. Thus

Game 3 $\mathbf{Exp}_{\mathcal{A}}^{AutVerif}\left[\mathcal{PVC}_{\mathrm{AC}}, F, 1^{\lambda}\right]$:

$L = \epsilon, o = \perp$
$(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}) \leftarrow \mathsf{SetupVC}(1^{\lambda})$;
$(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}) \leftarrow \mathsf{SetupAC}(1^{\lambda}, \mathcal{P}_C, \mathcal{P}_V)$;
$\lambda^V(\mathcal{A}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$SK_{\mathcal{A}} \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathcal{A}, \lambda^V(\mathcal{A}))$;
$L = L \cup \lambda^V(\mathcal{A})$;
$o \xleftarrow{\$} \mathcal{O}(F)$ s.t. $\lambda^V(o) > \lambda^V(v_i)\ \forall \lambda^V(v_i) \in L$;
$SK_C \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, C, \lambda^C(C))$ *s.t.* $\lambda^C(C) \geqslant \lambda^C(o)$;
$SK_S \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, V, \lambda^C(S))$ *s.t.* $\lambda^C(S) \geqslant \lambda^C(o)$;
$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F)$;
$(EK_{F,S}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S)$;
$(\sigma_x, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(o), \lambda^V(C), \lambda^V(o), \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$\sigma_y \leftarrow \mathsf{Compute}(\sigma_x, EK_{F,S}, SK_S, \lambda^C(S), \lambda^C(o), \mathrm{PP}_{AC})$;
$y' \leftarrow \mathcal{A}^{\mathcal{O}}(\sigma_y, VK_{F,x}, RK_{F,x}, \lambda(o), SK_{\mathcal{A}}, \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, PK_F, L_F)$;
If $y' = F(x)$
    Return 1
Else Return 0

Oracle Query 2 $\mathcal{O}^{\mathsf{Register}}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, ID, \lambda(ID))$:

if $(o \neq \perp) \wedge \lambda(ID) \geqslant \lambda^V(o)$ **then**
    return $\perp$
end if
$L = L \cup \lambda(ID)$;
return $\mathsf{Register}(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}, ID, \lambda(ID))$

we choose a computation o uniformly at random from the set of all possible computations of F, denoted $\mathcal{O}(F)$. We use the notation $o \xleftarrow{\$} \mathcal{O}(F)$ to denote this operation, and require that $\mathcal{C}$ repeatedly samples until it finds an o such that $\lambda(o) \not\leqslant \lambda(v_i)$ for all $\lambda(v_i)$ queried to the $\mathsf{Register}$ oracle. If not such o may be found then the game ends with a loss for the adversary as they did not select a valid attack identity. The challenger simulates a delegator and a server, and provides the encoded output from the server to the adversary who should not be able to guess the value of $F(x)$.

We define the advantage for this game as follows, where we subtract the adversary's best guess based on knowledge of F (if F is balanced then this will be $\frac{1}{\mathsf{Ran}(F)}$):

Definition 5. *The* advantage *of a PPT adversary $\mathcal{A}$ making a polynomial number of queries q in the Authorized Verification Experiment is defined as follows:*

$$Adv_{\mathcal{A}}^{\mathbf{AutVerif}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathbf{AutVerif}}[\mathcal{PVC}_{AC}, F, 1^{\lambda}] = 1] - \max_{y \in \mathsf{Ran}(F)} \left(\Pr_{x \in \mathsf{Dom}(F)}[F(x) = y] \right).$$

A PVC-AC is secure against Authorized Verification *for a function F, if for all PPT adversaries $\mathcal{A}$,*

$$Adv_{\mathcal{A}}^{\mathbf{AutVerif}}(\mathcal{PVC}_{AC},\ F,\ 1^{\lambda}, q) \leqslant \mathrm{negl}(\lambda).$$

5.4 Weak Input Privacy

Here we define a notion of weak input privacy. This is not as strong as the usual input privacy considered in PVC settings where computational servers learn nothing about the data they are working on. Instead, here, we consider that servers (or other entities) that are not authorized to access (compute on) the input data may not learn x. If full input privacy is required then the KP-ABE primitive used as a black box in the construction could be replaced by a predicate encryption scheme for the same class of functions.

This is captured in Game 4, which begins much like the Authorized Verification game. Again the challenger chooses the computation to avoid a trivial win, and the (unauthorized) adversary is provided with the encoded input and must guess x.

Game 4 $\mathbf{Exp}_{\mathcal{A}}^{WeakIP}\left[\mathcal{PVC}_{\mathrm{AC}}, F, 1^{\lambda}\right]$:

$L = \epsilon, o = \perp$
$(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}) \leftarrow \mathsf{SetupVC}(1^{\lambda})$;
$(\mathrm{PP}_{AC}, \mathrm{MK}_{AC}) \leftarrow \mathsf{SetupAC}(1^{\lambda}, \mathcal{P}_C, \mathcal{P}_V)$;
$\lambda^C(\mathcal{A}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$SK_{\mathcal{A}} \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathcal{A}, \lambda^C(\mathcal{A}))$;
$L = L \cup \lambda^C(\mathcal{A})$;
$o \xleftarrow{\$} \mathcal{O}(F)$ s.t. $\lambda^C(o) > \lambda^C(v_i)\ \forall \lambda^C(v_i) \in L$;
$SK_C \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, C, \lambda^C(C))$ *s.t.* $\lambda^C(C) \geqslant \lambda^C(o)$;
$SK_V \leftarrow \mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, V, \lambda^V(V))$ *s.t.* $\lambda^V(V) \geqslant \lambda^V(o)$;
$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F)$;
$(EK_{F,\mathcal{A}}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, \mathcal{A})$;
$(\sigma_x, VK_{F,x}, RK_{F,x}) \leftarrow \mathsf{ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(o), \lambda^V(C), \lambda^V(o), \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$;
$x' \leftarrow \mathcal{A}^{\mathcal{O}}(\sigma_x, VK_{F,x}, \lambda(o), EK_{F,\mathcal{A}}, SK_{\mathcal{A}}, \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, PK_F, L_F)$;
If $x' = x$
    Return 1
Else Return 0

Definition 6. *The* advantage *of a PPT adversary $\mathcal{A}$ making a polynomial number of queries q in the Weak Input Privacy Experiment is defined as follows:*

$$Adv_{\mathcal{A}}^{\mathbf{WeakIP}}(\mathcal{PVC}_{AC}, F, 1^{\lambda}, q) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathbf{WeakIP}}[\mathcal{PVC}_{AC}, F, 1^{\lambda}] = 1] - \max_{x \in \mathsf{Dom}(F)} \left(\Pr_{y \in \mathsf{Ran}(F)}[F(x) = y] \right).$$

A PVC-AC is secure against Weak Input Privacy *for a function F, if for all PPT adversaries $\mathcal{A}$,*

$$Adv_{\mathcal{A}}^{\mathbf{WeakIP}}(\mathcal{PVC}_{AC},\ F,\ 1^{\lambda}, q) \leqslant \mathrm{negl}(\lambda).$$

6 Construction

6.1 Informal Overview

We give a construction of PVC-AC using a Key-Indistinguishable KAS and any secure (black-box) RPVC in Appendix 6.2. Informally:

1. $\mathsf{PVC_{AC}}$.SetupVC simply calls RPVC.Setup whilst $\mathsf{PVC_{AC}}$.SetupAC initialises two KASs, one for the security poset corresponding to computation policies, and one corresponding to verification policies[5].

2. $\mathsf{PVC_{AC}}$.FnInit initializes a list of servers L_F authorized to compute F.

3. $\mathsf{PVC_{AC}}$.Register assigns keys and secret information from the KAS corresponding to $\lambda(ID)$. If ID is a server, it also calls RPVC.Register.

4. $\mathsf{PVC_{AC}}$.Certify creates the evaluation key $EK_{F,S}$ that will enable S to compute F by calling RPVC.Certify.

5. $\mathsf{PVC_{AC}}$.ProbGen runs RPVC.ProbGen to create an intermediate problem instance σ'_x, an output retrieval key $RK_{F,x}$ and a verification key $VK_{F,x}$. It then derives the appropriate keys for the computation and verification policy, and uses them to encrypt σ'_x and $RK_{F,x}$ respectively.

[5]Note that if the verification policies coincide with the computation policies then we may run this only once

6. $\mathsf{PVC_{AC}.Compute}$ first derives the key required by the computation policy for this computation and decrypts (reads) the encoded input σ_x. It then runs $\mathsf{RPVC.Compute}$ to obtain an encoding σ_y of $F(x)$.

7. $\mathsf{PVC_{AC}.BlindVerify}$ simply calls $\mathsf{RPVC.BlindVerify}$ since the verification key is not protected by an access control policy in this work. $\mathsf{PVC_{AC}.RetrieveOutput}$, on the other hand, first derives the required KAS key, $\kappa_{\lambda^V(F)}$, according to the verification policy and uses this to decrypt (read) the output retrieval key $RK_{F,x}$. Finally it runs $\mathsf{RPVC.BlindVerify}$ on $RK_{F,x}$ to learn $F(x)$.

8. $\mathsf{PVC_{AC}.Revoke}$ simply runs $\mathsf{RPVC.Revoke}$ to revoke a misbehaving server and distribute updated evaluation keys.

6.2 Instantiation

For the construction of a publicly verifiable outsourced computation scheme we basically wrap a key assignment scheme around a RPVC scheme (for example, the KP-ABE construction given by Alderman et al. [2]). Note that the RPVC scheme is used in a black-box manner and thus any concrete instantiation could be used. Additionally, a non-revocable scheme such as that given by Parno et al. [16] could be used if desired.

Let $\mathcal{RPVC}$ = ($\mathsf{RPVC.Setup}$, $\mathsf{RPVC.FnInit}$, $\mathsf{RPVC.Register}$, $\mathsf{RPVC.Certify}$, $\mathsf{RPVC.ProbGen}$, $\mathsf{RPVC.Compute}$, $\mathsf{RPVC.BlindVerify}$, $\mathsf{RPVC.RetrieveOutput}$, $\mathsf{RPVC.Revoke}$) be a RPVC scheme, as defined by Alderman et al. [2], for a class of functions $\mathcal{F}$. Let $\mathcal{SE}$= ($\mathsf{SE.KeyGen}$, $\mathsf{SE.Encrypt}$, $\mathsf{SE.Decrypt}$) be an IND-CPA secure symmetric encryption scheme. Let $\mathcal{KAS}$ = ($\mathsf{KAS.MakeKeys}$, $\mathsf{KAS.MakeSecrets}$, $\mathsf{KAS.MakePublicData}$, $\mathsf{KAS.GetKey}$) be a Key Assignment Scheme whose keys are compatible with $\mathcal{SE}$. Finally, let $\mathcal{P}_C$ denote the poset encoding computation policies (e.g. $(L, \leqslant)$), and similarly let $\mathcal{P}_V$ denote the poset encoding verification policies (e.g. $(L, \geqslant)$)[6]. Then there is a PVC-AC scheme $\mathcal{PVC}_{\mathrm{AC}}$ = ($\mathsf{PVC_{AC}.SetupVC}$, $\mathsf{PVC_{AC}.SetupAC}$, $\mathsf{PVC_{AC}.FnInit}$, $\mathsf{PVC_{AC}.Register}$, $\mathsf{PVC_{AC}.Certify}$, $\mathsf{PVC_{AC}.ProbGen}$, $\mathsf{PVC_{AC}.Compute}$, $\mathsf{PVC_{AC}.Verify}$, $\mathsf{PVC_{AC}.Revoke}$) for the same class of functions $\mathcal{F}$ which is defined as follows in Algorithms 1-10.

Algorithm 1 $\mathsf{PVC_{AC}.SetupVC}(1^\lambda) \rightarrow (\mathrm{PP}_{VC}, \mathrm{MK}_{VC})$

1: Run $(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}) \leftarrow \mathsf{RPVC.Setup}(1^\lambda)$

Algorithm 2 $\mathsf{PVC_{AC}.SetupAC}(1^\lambda, \mathcal{P}_C, \mathcal{P}_V) \rightarrow (\mathrm{PP}_{AC}, \mathrm{MK}_{AC})$

1: $\kappa_{\mathcal{P}_C} \leftarrow \mathsf{KAS.MakeKeys}(1^\lambda, \mathcal{P}_C)$
2: $\omega_{\mathcal{P}_C} \leftarrow \mathsf{KAS.MakeSecrets}(1^\lambda, \mathcal{P}_C)$
3: $Pub_{\mathcal{P}_C} \leftarrow \mathsf{KAS.MakePublicData}(1^\lambda, \mathcal{P}_C)$
4: $\kappa_{\mathcal{P}_V} \leftarrow \mathsf{KAS.MakeKeys}(1^\lambda, \mathcal{P}_V)$
5: $\omega_{\mathcal{P}_V} \leftarrow \mathsf{KAS.MakeSecrets}(1^\lambda, \mathcal{P}_V)$
6: $Pub_{\mathcal{P}_V} \leftarrow \mathsf{KAS.MakePublicData}(1^\lambda, \mathcal{P}_V)$
7: $\mathrm{PP}_{AC} = (Pub_{\mathcal{P}_C}, Pub_{\mathcal{P}_V})$
8: $\mathrm{MK}_{AC} = (\kappa_{\mathcal{P}_C}, \omega_{\mathcal{P}_C}, \kappa_{\mathcal{P}_V}, \omega_{\mathcal{P}_V})$

Algorithm 3 $\mathsf{PVC_{AC}.FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F) \rightarrow (PK_F, L_F)$

1: $(PK_F, L_F) \leftarrow \mathsf{RPVC.FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F)$

[6]We use the notation $\kappa_{\mathcal{P}_C}$ to denote the set of all keys generated by the KAS for the computation poset $\mathcal{P}_C$ and for a label $\lambda^C(\mathrm{ID}) \in \mathcal{P}_\mathrm{C}$, $\kappa_{\lambda^C(\mathrm{ID})} \in \kappa_{\mathcal{P}_C}$ is the key associated to that label in the computation policy.

Algorithm 4 $\mathsf{PVC_{AC}.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \mathrm{ID}, \lambda(\mathrm{ID})) \rightarrow \mathrm{SK}_{\mathrm{ID}}$

if ID is a server **then**
  $SK'_{\mathrm{ID}} \leftarrow \mathsf{RPVC.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{ID})$
  $SK_{\mathrm{ID}} = (SK'_{\mathrm{ID}}, \kappa_{\lambda^C(\mathrm{ID})}, \omega_{\lambda^C(\mathrm{ID})})$
else
  if ID is a delegator **then**
    $SK_{\mathrm{ID}} = (\kappa_{\lambda^C(\mathrm{ID})}, \omega_{\lambda^C(\mathrm{ID})}, \kappa_{\lambda^V(\mathrm{ID})}, \omega_{\lambda^V(\mathrm{ID})})$
  else
    $SK_{\mathrm{ID}} = (\kappa_{\lambda^V(\mathrm{ID})}, \omega_{\lambda^V(\mathrm{ID})})$
  end if
end if

Algorithm 5 $\mathsf{PVC_{AC}.Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S) \rightarrow (EK_{F,S}, L_F)$

$(EK_{F,S}, L_F) \leftarrow \mathsf{RPVC.Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S)$

Algorithm 6 $\mathsf{PVC_{AC}.ProbGen}(x, SK_C, PK_F, \lambda^C(C), \lambda^C(F), \lambda^V(C), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}) \rightarrow (\sigma_x, VK_{F,x}, RK_{F,x})$

$(\sigma'_x, VK_{F,x}, RK'_{F,x}) \leftarrow \mathsf{RPVC.ProbGen}(x, PK_F)$
$\kappa_{\lambda^C(F)} \leftarrow \mathsf{KAS.GetKey}(\lambda^C(C), \lambda^C(F), \omega_{\lambda^C(C)}, \mathrm{PP}_{AC})$
$\kappa_{\lambda^V(F)} \leftarrow \mathsf{KAS.GetKey}(\lambda^V(C), \lambda^V(F), \omega_{\lambda^V(C)}, \mathrm{PP}_{AC})$
if $(\kappa_{\lambda^C(F)} \neq \perp)$ and $(\kappa_{\lambda^V(F)} \neq \perp)$ **then**
  $\sigma_x = (\lambda^C(F), \mathsf{SE.Encrypt}(\sigma'_x, \kappa_{\lambda^C(F)}))$
  $RK_{F,x} = (\lambda^V(F), \mathsf{SE.Encrypt}(RK'_{F,x}, \kappa_{\lambda^V(F)}))$
end if

Algorithm 7 $\mathsf{PVC_{AC}.Compute}(\sigma_x, EK_{F,S}, SK_S, \lambda^C(S), \lambda^C(F), PP_{AC}) \rightarrow \sigma_y$

Parse σ_x as $(\lambda^C(F), c)$
$\kappa_{\lambda^C(F)} \leftarrow \mathsf{KAS.GetKey}(\lambda^C(S), \lambda^C(F), \omega_{\lambda^C(S)}, \mathrm{PP}_{AC})$
if $\kappa_{\lambda^C(F)} = \perp$ **then**
  Output $\perp$
else
  $\sigma'_x \leftarrow \mathsf{SE.Decrypt}(c, \kappa_{\lambda^C(F)})$
  $\sigma_y \leftarrow \mathsf{RPVC.Compute}(\sigma'_x, EK_{F,S}, SK_S)$
end if

Algorithm 8 $\mathsf{PVC_{AC}.BlindVerify}(\mathrm{PP}_{VC}, \sigma_y, VK_{F,x}, L_F) \rightarrow (\mu, \tau_{\sigma_y})$:

$(\mu, \tau_{\sigma_y}) \leftarrow \mathsf{RPVC.BlindVerify}(\mathrm{PP}_{VC}, \sigma_y, VK_{F,x}, L_F)$

Algorithm 9 $\mathsf{PVC_{AC}.RetrieveOutput}(\mu, \tau_{\sigma_y}, VK_{F,x}, RK_{F,x}, \lambda^V(V), \lambda^V(F)) \rightarrow \tilde{y}$:

Parse $RK_{F,x}$ as $(\lambda^V(F), e)$
$\kappa_{\lambda^V(F)} \leftarrow \mathsf{KAS.GetKey}(\lambda^V(V), \lambda^V(F), \omega_{\lambda^V(V)}, \mathrm{PP}_{AC})$
if $\kappa_{\lambda^V(F)} = \perp$ **then**
  Output $\perp$
else
  $RK_{F,x}{}' \leftarrow \mathsf{SE.Decrypt}(e, \kappa_{\lambda^V(F)})$
  $\tilde{y} \leftarrow \mathsf{RPVC.RetrieveOutput}(\mu, \tau_{\sigma_y}, VK_{F,x}, RK'_{F,x})$
end if

Algorithm 10 $\mathsf{PVC_{AC}.Revoke}(\mathrm{MK}_{VC}, \tau_{\sigma_y}, F, L_F) \rightarrow (\{EK_{F,S'}\}, L_F)$ or $\perp$

$(\{EK_{F,S'}\}, L_F)$ or $\perp \leftarrow \mathsf{RPVC.Revoke}(\mathrm{MK}_{VC}, \tau_{\sigma_y}, F, L_F)$

6.3 Proof of Security

We now give a theorem and proof that the construction presented above is secure against the games presented in Section 5.

Theorem 1. *Given a RPVC scheme secure in the sense of Public Verifiability, Revocation and Vindictive Servers, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10*. Then*

$\mathcal{PVC}_{AC}$ is secure in the sense of Public Verifiability, Revocation, Vindictive Servers, Authorized Outsourcing, Authorized Computation, Weak Input Privacy, and Authorized Verification.

Informally, the security proofs follow from the Key-Indistinguishability of the KAS and the security of the underlying RPVC scheme. For each, we show that if an adversary exists, then we can break the Key-Indistinguishability of the KAS. We now present a formal proof for Theorem 1.

6.4 Public Verifiability, Revocation and Vindictive Servers

Lemma 1. *Given a secure RPVC scheme, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10*. Then $\mathcal{PVC}_{AC}$ is secure in the sense of Public Verifiability, Revocation and Vindictive Servers.*

Proof. The notions of Public Verifiability, Revocation and Vindictive Servers follow naturally from the full security proofs presented in [2] with additional inputs for policy declarations. However, as these do not rely on the access control properties introduced in this work, the proofs follow immediately by adjusting the notation to the present PVC-AC scheme. □

6.5 Authorized Outsourcing

Lemma 2. *Given a secure RPVC scheme, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10*. Then $\mathcal{PVC}_{AC}$ is secure in the sense of Authorized Outsourcing (Game 1).*

Proof. Assume that $\mathcal{A}_{VC}$ is an adversary with non-negligable advantage δ in the Authorized Outsourcing game. We show that we can use this to construct an adversary, $\mathcal{A}_{KI}$, with non-negligable advantage in the Key-Indistinguishability game (Game 5). Let $\mathcal{C}$ be the challenger for $\mathcal{A}_{KI}$ and let $\mathcal{A}_{KI}$ act as the challenger for $\mathcal{A}_{VC}$.

1. $\mathcal{C}$ begins by running
$$\kappa_{\mathcal{P}_C} \leftarrow \mathsf{MakeKeys}(1^\lambda, \mathcal{P}_C),$$
$$\omega_{\mathcal{P}_C} \leftarrow \mathsf{MakeSecrets}(1^\lambda, \mathcal{P}_C), \text{ and}$$
$$Pub_{\mathcal{P}_C} \leftarrow \mathsf{MakePublicData}(1^\lambda, \mathcal{P}_C).$$
It initialises the list Q to be empty and sends $Pub_{\mathcal{P}_C}$ to $\mathcal{A}_{KI}$.

2. $\mathcal{A}_{KI}$ first sets $L = \epsilon$ and $o = \perp$. Then it runs lines 4 to 7 of Algorithm 2 and sets $\mathrm{MK}_{AC} = (\perp, \perp, \kappa_{\mathcal{P}_V}, \omega_{\mathcal{P}_V})$. It also runs Algorithm 1 as given, and sends PP_{VC} and PP_{AC} to $\mathcal{A}_{VC}$.

3. Now, $\mathcal{A}_{VC}$ is provided with oracle access which $\mathcal{A}_{KI}$ can simulate as follows:

   (a) $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as specified in the construction as these rely on the underlying RPVC scheme only.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ will query for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ will follow Algorithm 4 but will make use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \perp)$, to which $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^C(ID)})$ as well as updating $L = L \cup \lambda^C(ID)$. Note that keys and secrets for verification policies are owned by $\mathcal{A}_{KI}$ already.

$\mathcal{A}_{VC}$ chooses a security label, $\lambda^C(\mathcal{A}_{VC})$, from the computation policy for itself and sends it to $\mathcal{A}_{KI}$.

4. To register $\mathcal{A}_{VC}$ as a delegator with this security label, $\mathcal{A}_{KI}$ will make use of its $\mathsf{Corrupt}$ oracle queries in the Key Indistinguishability game – that is, it will query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(\mathcal{A}_{VC}), \perp)$ in Game 5. $\mathcal{C}$ will add $\lambda^C(\mathcal{A}_{VC})$ to the list Q and return the relevant key and secret information $(\kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$. Finally, $\mathcal{A}_{KI}$ forwards $\mathrm{SK}_{\mathcal{A}_{\mathrm{VC}}} = (\kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$ to $\mathcal{A}_{VC}$, as well as updating the list to $L = L \cup \lambda^C(\mathcal{A}_{VC})$.

5. Now, $\mathcal{A}_{VC}$ is again provided with oracle access which $\mathcal{A}_{KI}$ can simulate as before. $\mathcal{A}_{VC}$ eventually chooses a computation o that it wishes to be challenged upon (which must be for the function F parameterised in the game, and will specify the input data x). If the chosen o does not satisfy the requirement that $\mathcal{A}_{VC}$ is not authorised to delegate it (i.e. that $\lambda^C(o) \not\leqslant \lambda^C(\mathcal{A}_{VC})$) then the game ends with a loss for the adversary.

6. $\mathcal{A}_{KI}$ sends $\mathcal{C}$ its choice of challenge security label in the Key Indistinguishability game $v^\star = \lambda^C(o)$. Notice that this choice is allowed since $\lambda^C(o)$ is not a descendent of any security labels given in a $\mathsf{Corrupt}$ oracle query (those labels added to Q). This is necessarily so since the only $\mathsf{Corrupt}$ queries thus far have been for the adversary's label $\lambda^C(\mathcal{A}_{VC})$ in Step 4, and those labels given in a $\mathsf{Register}$ query by $\mathcal{A}_{VC}$. Now, by the condition in Step 5 we know $\lambda^C(o)$ is certainly not less than or equal to $\lambda^C(\mathcal{A}_{VC})$. Also, by the restriction on $\mathcal{A}_{VC}$'s oracle queries, it is not allowed to query for any $\lambda^C(ID) \geqslant \lambda^C(o)$ as this would constitute a trivial win.

   $\mathcal{C}$ chooses a random bit b and returns $\kappa_{\lambda^C(o)}$ if $b = 0$ or chooses a random key from the keyspace otherwise, and sends this key, $\kappa^\star$, to $\mathcal{A}_{KI}$.

7. $\mathcal{A}_{KI}$ must now simulate a computational server and a verifier. It does this by first registering two such entities, S and V respectively, that are authorised to compute and verify o respectively. It sets $\lambda^C(S) = \lambda^C(o)$ and hence $\kappa_{\lambda^C(S)} = \kappa^\star$. Note that this is the only part of SK_S that is required, and in particular $\omega_{\lambda^C(S)}$ is not required (as this is only needed for deriving keys, and S will only need to use $\kappa^\star$).

   To register S, $\mathcal{A}_{KI}$ runs

$$SK'_S \leftarrow \mathsf{RPVC.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, S)$$

   and then sets $SK_S = (SK'_S, \kappa_{\lambda^C(S)}, \perp)$.

   To register V, $\mathcal{A}_{KI}$ sets $SK_V = (\kappa_{\lambda^V(V)}, \omega_{\lambda^V(V)})$.

8. $\mathcal{A}_{KI}$ runs

$$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F) \text{ and}$$
$$(EK_{F,S}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S)$$

   and sends (PK_F, L_F) to $\mathcal{A}_{VC}$.

9. $\mathcal{A}_{VC}$ again gets oracle access which $\mathcal{A}_{KI}$ responds to as follows:

   (a) As before $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as given in the construction.

(b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ queries for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ follows Algorithm 4 making use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \lambda^C(o))$. $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^C(ID)})$ if and only if $\lambda^C(ID) \not\geqslant \lambda^C(o)$ and updating the list L accordingly. $\mathcal{A}_{KI}$ can respond with verification keys and secrets since it already owns them.

$\mathcal{A}_{VC}$ eventually outputs what it hopes to be an encoded input σ_x, a verification key $VK_{F,x}$ and an output retrieval key $RK_{F,x}$ for the challenge computation o.

10. $\mathcal{A}_{KI}$ attempts to compute F on the provided encoded input, and to verify the result. To run $\mathsf{Compute}$, $\mathcal{A}_{KI}$ parses σ_x as $(\lambda^C(o), c)$ and runs

$$\sigma'_x \leftarrow \mathsf{SE.Decrypt}(c, \kappa^\star) \text{ and}$$

$$\sigma_y \leftarrow \mathsf{RPVC.Compute}(\sigma'_x, EK_{F,S}, SK_S).$$

To verify the result, $\mathcal{A}_{KI}$ runs

$$(\tilde{y}, \tau_{\sigma_y}) \leftarrow \mathsf{Verify}(\sigma_y, SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC})$$

11. If $b = 0$ then $\kappa^\star$ is a real key corresponding to $\lambda^C(o)$ and therefore the decryption of c will succeed (if $\mathcal{A}_{VC}$ performed successfully as we have assumed). Thus, σ'_x is a valid encoded input and so $\mathsf{Compute}$ and Verify will execute correctly, yielding $(\tilde{y}, \tau_{\sigma_y}) \neq (\perp, (\mathsf{reject}, \mathcal{A})))$. In this case, $\mathcal{A}_{KI}$ will output a guess $b' = 0$.

Otherwise, $\kappa^\star$ is a random key and the decryption should not give a valid encoded input that will permit a valid execution of $\mathsf{Compute}$ and Verify, and $\mathcal{A}_{KI}$ should output a guess $b' = 1$.

We conclude that $\mathcal{A}_{KI}$ at least succeeds whenever $\mathcal{A}_{VC}$ succeeds i.e. with non-negligable probability δ. However, our construction assumes that the KAS is secure in the sense of Key Indistinguishability, and hence $\mathcal{A}_{VC}$ cannot have a non-negligable success probability. □

6.6 Authorized Computation

Lemma 3. *Given a secure RPVC scheme, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10. *Then $\mathcal{PVC}_{AC}$ is secure in the sense of Authorized Computation (Game 2).*

Proof. The proof is partially similar to the proof of the previous lemma. Assume that $\mathcal{A}_{VC}$ is an adversary with non-negligable advantage δ in the Authorized Computation game. We show that we can use this to construct an adversary, $\mathcal{A}_{KI}$, with non-negligable advantage in the Key-Indistinguishability game. Let $\mathcal{C}$ be the challenger for $\mathcal{A}_{KI}$ and let $\mathcal{A}_{KI}$ act as the challenger for $\mathcal{A}_{VC}$.

1. $\mathcal{C}$ begins by running

$$\kappa_{\mathcal{P}_C} \leftarrow \mathsf{MakeKeys}(1^\lambda, \mathcal{P}_C),$$

$$\omega_{\mathcal{P}_C} \leftarrow \mathsf{MakeSecrets}(1^\lambda, \mathcal{P}_C), \text{ and}$$

$$Pub_{\mathcal{P}_C} \leftarrow \mathsf{MakePublicData}(1^\lambda, \mathcal{P}_C).$$

It initialises the list Q to be empty and sends $Pub_{\mathcal{P}_C}$ to $\mathcal{A}_{KI}$.

2. $\mathcal{A}_{KI}$ first sets $L = \epsilon$ and $o = \perp$. Then it runs lines 4 to 7 of Algorithm 2 and sets $\mathrm{MK}_{AC} = (\perp, \perp, \kappa_{\mathcal{P}_V}, \omega_{\mathcal{P}_V})$. It also runs Algorithm 1 as given, and sends PP_{VC} and PP_{AC} to $\mathcal{A}_{VC}$.

3. Now, $\mathcal{A}_{VC}$ is provided with oracle access which $\mathcal{A}_{KI}$ can simulate as follows:

   (a) $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as specified in the construction as these rely on the underlying RPVC scheme only.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ will query for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ will follow Algorithm 4 but will make use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \perp)$, to which $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^C(ID)})$ as well as updating $L = L \cup \lambda^C(ID)$. Note that keys and secrets for verification policies are owned by $\mathcal{A}_{KI}$ already.

   $\mathcal{A}_{VC}$ chooses a security label, $\lambda^C(\mathcal{A}_{VC})$, from the computation policy for itself and sends it to $\mathcal{A}_{KI}$.

4. To register $\mathcal{A}_{VC}$ as a server with this security label, $\mathcal{A}_{KI}$ will make use of its $\mathsf{Corrupt}$ oracle queries in the Key Indistinguishability game – that is, it will query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(\mathcal{A}_{VC}), \perp)$ in Game 5. $\mathcal{C}$ will add $\lambda^C(\mathcal{A}_{VC})$ to the list Q and return the relevant key and secret information $(\kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$. Additionally, $\mathcal{A}_{KI}$ runs $\mathsf{RPVC.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathcal{A}_{VC})$ to create $SK'_{\mathcal{A}_{VC}}$. Finally, $\mathcal{A}_{KI}$ sets $\mathrm{SK}_{\mathcal{A}_{\mathrm{VC}}} = (SK'_{\mathcal{A}_{VC}}, \kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$ as well as updating the list to $L = L \cup \lambda^C(\mathcal{A}_{VC})$.

5. Now, $\mathcal{A}_{VC}$ is again provided with oracle access which $\mathcal{A}_{KI}$ can simulate as before. $\mathcal{A}_{VC}$ eventually chooses a computation o that it wishes to be challenged upon (which must be for the function F parameterised in the game, and will specify the input data x). If the chosen o does not satisfy the requirement that $\mathcal{A}_{VC}$ is not authorised to delegate it (i.e. that $\lambda^C(o) \not\leqslant \lambda^C(\mathcal{A}_{VC})$) then the game ends with a loss for the adversary.

6. $\mathcal{A}_{KI}$ sends $\mathcal{C}$ its choice of challenge security label in the Key Indistinguishability game $v^\star = \lambda^C(o)$. Notice that this choice is allowed since $\lambda^C(o)$ is not a descendent of any security labels given in a $\mathsf{Corrupt}$ oracle query (those labels added to Q). This is necessarily so since the only $\mathsf{Corrupt}$ queries thus far have been for the adversary's label $\lambda^C(\mathcal{A}_{VC})$ in Step 4, and those labels given in a $\mathsf{Register}$ query by $\mathcal{A}_{VC}$. Now, by the condition in Step 5 we know $\lambda^C(o)$ is a descendant of $\lambda^C(\mathcal{A}_{VC})$. Also, by the restriction on $\mathcal{A}_{VC}$'s oracle queries, it is not allowed to query for any $\lambda^C(ID) \geqslant \lambda^C(o)$ as this would constitute a trivial win.

   $\mathcal{C}$ chooses a random bit b and returns $\kappa_{\lambda^C(o)}$ if $b = 0$ or chooses a random key from the keyspace otherwise, and sends this key, $\kappa^\star$, to $\mathcal{A}_{KI}$.

7. $\mathcal{A}_{KI}$ must now simulate a delegator and a verifier. It does this by first registering two such entities, C and V respectively, that are authorised to compute and verify o respectively. It sets $\lambda^C(C) = \lambda^C(o)$ and hence $\kappa_{\lambda^C(C)} = \kappa^\star$. Note that this is the only part of SK_C that is required, and in particular $\omega_{\lambda^C(C)}$ is not required (as this is only needed for deriving keys, and S will only need to use $\kappa^\star$).

   To register C, $\mathcal{A}_{KI}$ sets $SK_C = (\kappa^\star, \perp, \kappa_{\lambda^V(C)}, \omega_{\lambda^V(C)})$. Furthermore, to register V, $\mathcal{A}_{KI}$ sets $SK_V = (\kappa_{\lambda^V(V)}, \omega_{\lambda^V(V)})$.

8. $\mathcal{A}_{KI}$ runs
$$(PK_F, L_F) \leftarrow \mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F) \text{ and}$$
$$(EK_{F,\mathcal{A}_{VC}}, L_F) \leftarrow \mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, \mathcal{A}_{VC})$$
and sends $(PK_F, L_F F$ and $EK_{F,\mathcal{A}_{VC}}$ to $\mathcal{A}_{VC}$.

9. $\mathcal{A}_{KI}$ runs
$$(\sigma'_x, VK_{F,x}, RK'_{F,x}) \leftarrow \mathsf{RPVC.ProbGen}(x, PK_F)$$
and prepares the encoded input being $\sigma_x = (\lambda^C(o), \mathsf{SE.Encrypt}(\sigma'_x, \kappa^\star))$ which it sends to $\mathcal{A}_{VC}$. Furthermore $\mathcal{A}_{KI}$ prepares $RK_{F,x} = (\lambda^V(o), \mathsf{SE.Encrypt}(RK'_{F,x}, \kappa_{\lambda^V(o)}))$ which it also sends to $\mathcal{A}_{VC}$.

10. $\mathcal{A}_{VC}$ again gets oracle access which $\mathcal{A}_{KI}$ responds to as follows:

    (a) As before $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as given in the construction.

    (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ queries for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ follows Algorithm 4 making use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \lambda^C(o))$. $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^C(ID)})$ if and only if $\lambda^C(ID) \not\geqslant \lambda^C(o)$. $\mathcal{A}_{KI}$ can update the list L accordingly and respond with verification keys and secrets since it already owns them.

    $\mathcal{A}_{VC}$ eventually outputs what it hopes to be an encoded output σ_y for the challenge computation o.

11. $\mathcal{A}_{KI}$ attempts to verify the result on the provided encoded output. It runs
$$(\tilde{y}, \tau_{\sigma_y}) \leftarrow \mathsf{Verify}(\sigma_y, SK_V, VK_{F,x}, L_F, \lambda^V(V), \lambda^V(F), \mathrm{PP}_{VC}, \mathrm{PP}_{AC}, RK_{F,x}).$$

12. If $b = 0$ then $\kappa^\star$ is a real key corresponding to $\lambda^C(o)$ and if $\mathsf{ProbGen}$ was executed correctly then $\mathcal{A}_{VC}$ should decrypt successfully. Thus, σ_y is a valid encoded output and so Verify will execute correctly, yielding $(\tilde{y}, \tau_{\sigma_y}) \neq (\perp, (\mathsf{reject}, \mathcal{A})))$. In this case, $\mathcal{A}_{KI}$ will output a guess $b' = 0$.

    Otherwise, $\kappa^\star$ is a random key and the decryption should not be successful and should not give a valid encoded output that will permit a valid execution Verify, and $\mathcal{A}_{KI}$ should output a guess $b' = 1$.

We conclude that $\mathcal{A}_{KI}$ at least succeeds whenever $\mathcal{A}_{VC}$ succeeds i.e. with non-negligable probability δ. However, our construction assumes that the KAS is secure in the sense of Key Indistinguishability, and hence $\mathcal{A}_{VC}$ cannot have a non-negligable success probability. □

6.7 Weak Input Privacy

Lemma 4. *Given a secure RPVC scheme, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10*. Then $\mathcal{PVC}_{AC}$ is secure in the sense of Weak Input Privacy (Game 4).*

Proof. Assume that $\mathcal{A}_{VC}$ is an adversary with non-negligable advantage δ in the Authorized Computation game. We show that we can use this to construct an adversary, $\mathcal{A}_{KI}$, with non-negligable advantage in the Key-Indistinguishability game. Let $\mathcal{C}$ be the challenger for $\mathcal{A}_{KI}$ and let $\mathcal{A}_{KI}$ act as the challenger for $\mathcal{A}_{VC}$.

1. $\mathcal{C}$ begins by running
$$\kappa_{\mathcal{P}_C} \leftarrow \mathsf{MakeKeys}(1^\lambda, \mathcal{P}_C),$$
$$\omega_{\mathcal{P}_C} \leftarrow \mathsf{MakeSecrets}(1^\lambda, \mathcal{P}_C), \text{ and}$$
$$Pub_{\mathcal{P}_C} \leftarrow \mathsf{MakePublicData}(1^\lambda, \mathcal{P}_C).$$
It initialises the list Q to be empty and sends $Pub_{\mathcal{P}_C}$ to $\mathcal{A}_{KI}$.

2. $\mathcal{A}_{KI}$ first sets $L = \epsilon$ and $o = \perp$. Then it runs lines 4 to 7 of Algorithm 2 and sets $\mathrm{MK}_{AC} = (\perp, \perp, \kappa_{\mathcal{P}_V}, \omega_{\mathcal{P}_V})$. It also runs Algorithm 1 as given, and sends PP_{VC} and PP_{AC} to $\mathcal{A}_{VC}$.

3. Now, $\mathcal{A}_{VC}$ is provided with oracle access which $\mathcal{A}_{KI}$ can simulate as follows:

   (a) $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as specified in the construction as these rely on the underlying RPVC scheme only.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ will query for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ will follow Algorithm 4 but will make use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \perp)$, to which $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^V(ID)})$ as well as updating $L = L \cup \lambda^C(ID)$. Note that keys and secrets for verification policies are owned by $\mathcal{A}_{KI}$ already.

   $\mathcal{A}_{VC}$ will eventually output a choice of computational label $\lambda^C(\mathcal{A}_{VC})$.

4. To register $\mathcal{A}_{VC}$ as a server with this security label, $\mathcal{A}_{KI}$ will make use of its $\mathsf{Corrupt}$ oracle queries in the Key Indistinguishability game – that is, it will query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(\mathcal{A}_{VC}), \perp)$ in Game 5. $\mathcal{C}$ will add $\lambda^C(\mathcal{A}_{VC})$ to the list Q and return the relevant key and secret information $(\kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$. Additionally, $\mathcal{A}_{KI}$ updates the list to $L = L \cup \lambda^C(\mathcal{A}_{VC})$, runs $\mathsf{RPVC.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathcal{A}_{VC})$ and outputs $SK'_{\mathcal{A}_{VC}}$. Finally, $\mathcal{A}_{KI}$ forwards $\mathrm{SK}_{\mathcal{A}_{\mathrm{VC}}} = (SK'_{\mathcal{A}_{VC}}, \kappa_{\lambda^C(\mathcal{A}_{VC})}, \omega_{\lambda^C(\mathcal{A}_{VC})})$ to $\mathcal{A}_{VC}$.

5. $\mathcal{C}$ now chooses a computation o uniformly at random from the space of all possible computations of F that the adversary is not authorised to compute i.e. we require $\lambda^C(o) \not\leqslant \lambda^C(\mathcal{A}_{VC})$. Note that the choice of o will dictate the input data x. If not such o may be found then the game ends with a loss for the adversary as they did not select a valid attack identity.

6. $\mathcal{A}_{KI}$ sends $\mathcal{C}$ its choice of challenge security label in the Key Indistinguishability game $v^\star = \lambda^C(o)$. Notice that this choice is allowed since $\lambda^C(o)$ is not a descendent of any security labels given in a $\mathsf{Corrupt}$ oracle query (those labels added to Q). This is necessarily so since the only $\mathsf{Corrupt}$ queries thus far have been for the adversary's label $\lambda^C(\mathcal{A})$ in Step 4, and those labels given in a $\mathsf{Register}$ query by $\mathcal{A}_{VC}$. Now, by the condition in Step 5 we know $\lambda^C(o)$ is certainly not less than or equal to $\lambda^C(\mathcal{A})$. Also, by the restriction on $\mathcal{A}_{VC}$'s oracle queries, it is not allowed to query for any $\lambda^C(ID) \geqslant \lambda^C(o)$ as this would constitute a trivial win.

$\mathcal{C}$ chooses a random bit b and returns $\kappa_{\lambda^C(o)}$ if $b = 0$ or chooses a random key from the keyspace otherwise, and sends this key, $\kappa^\star$, to $\mathcal{A}_{KI}$.

7. $\mathcal{A}_{KI}$ must now simulate a client and a verifier. It does this by first registering two such entities, C and V respectively, that are authorised to compute and verify o respectively. It sets $\lambda^C(C) = \lambda^C(o)$ and hence $\kappa_{\lambda^C(C)} = \kappa^\star$. Note that this is the only part of SK_C that is required, and in particular $\omega_{\lambda^C(C)}$ is not required (as this is only needed for deriving keys, and S will only need to use $\kappa^\star$).

   To register C, $\mathcal{A}_{KI}$ sets $SK_C = (\kappa^\star, \perp, \kappa_{\lambda^V(C)}, \omega_{\lambda^V(C)})$. Furthermore, to register V, $\mathcal{A}_{KI}$ sets $SK_V = (\kappa_{\lambda^V(V)}, \omega_{\lambda^V(V)})$.

8. $\mathcal{A}_{KI}$ runs

$$(PK_F, L_F) \leftarrow \mathsf{RPVC.FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F),$$
$$(EK_{F,S}, L_F) \leftarrow \mathsf{RPVC.Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S),$$
$$(\sigma'_x, VK_{F,x}, RK'_{F,x}) \leftarrow \mathsf{RPVC.ProbGen}(x, PK_F) \text{ for } x \text{ specified in } o,$$
$$\sigma_x = (\lambda^C(o), \mathsf{SE.Encrypt}(\sigma'_x, \kappa^\star))$$
$$RK_{F,x} = (\lambda^V(o), \mathsf{SE.Encrypt}(RK'_{F,x}, \kappa_{\lambda^V(o)}))$$

   $\mathcal{A}_{KI}$ sends $(\sigma_y, VK_{F,x}, \lambda(o), SK_{\mathcal{A}}, PK_F, L_F)$ to $\mathcal{A}_{VC}$.

9. $\mathcal{A}_{VC}$ again gets oracle access which $\mathcal{A}_{KI}$ responds to as follows:

   (a) As before $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as given in the construction.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ queries for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ follows Algorithm 4 making use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^C(ID)}$ or $\omega_{\lambda^C(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^C(ID), \lambda^C(o))$. $\mathcal{C}$ will respond by adding $\lambda^C(ID)$ to the query list Q and returning $(\kappa_{\lambda^C(ID)}, \omega_{\lambda^C(ID)})$ if and only if $\lambda^C(ID) \not\geqslant \lambda^C(o)$ and updating the list L accordingly. $\mathcal{A}_{KI}$ can respond with verification policy keys and secrets since it already owns them.

   $\mathcal{A}_{VC}$ eventually outputs a guess, x', for x.

10. By the IND-CPA property of the symmetric encryption scheme we hide the attributes attached to the intermediate encodings in $\mathsf{ProbGen}$. If $b = 0$ then $\kappa^\star$ is a real key corresponding to $\lambda^C(o)$ and therefore the encryption of σ'_x will be performed using a valid key and will look correct to $\mathcal{A}_{VC}$. Therefore, to $\mathcal{A}_{VC}$ the system looks consistent and correct, and hence he will have non-negligible advantage δ to guess x correctly, as we assumed. If $x' = x$, $\mathcal{A}_{KI}$ will output a guess $b' = 0$.

    Otherwise, $\kappa^\star$ is a random key and the encryption of the intermediate encoded input σ'_x will not be correct, so $\mathcal{A}_{VC}$ may not possess the assumed advantage and $\mathcal{A}_{KI}$ should output a guess $b' = 1$.

We conclude that $\mathcal{A}_{KI}$ at least succeeds whenever $\mathcal{A}_{VC}$ succeeds i.e. with non-negligable probability δ. However, our construction assumes that the KAS is secure in the sense of Key Indistinguishability, and hence $\mathcal{A}_{VC}$ cannot have a non-negligable success probability.

□

6.8 Authorized Verification

Lemma 5. *Given a secure RPVC scheme, a Key-Indistinguishability secure KAS and an* IND-CPA *secure symmetric encryption scheme, let $\mathcal{PVC}_{AC}$ be the PVC-AC scheme defined in Algorithms* 1-10. *Then $\mathcal{PVC}_{AC}$ is secure in the sense of Authorized Verification (Game 3).*

Proof. Assume that $\mathcal{A}_{VC}$ is an adversary with non-negligable advantage δ in the Authorized Verification game. We show that we can use this to construct an adversary, $\mathcal{A}_{KI}$, with non-negligable advantage in the Key-Indistinguishability game (Game 5). Let $\mathcal{C}$ be the challenger for $\mathcal{A}_{KI}$ and let $\mathcal{A}_{KI}$ act as the challenger for $\mathcal{A}_{VC}$.

1. $\mathcal{C}$ begins by running
$$\kappa_{\mathcal{P}_V} \leftarrow \mathsf{MakeKeys}(1^\lambda, \mathcal{P}_V),$$
$$\omega_{\mathcal{P}_V} \leftarrow \mathsf{MakeSecrets}(1^\lambda, \mathcal{P}_V), \text{ and}$$
$$Pub_{\mathcal{P}_V} \leftarrow \mathsf{MakePublicData}(1^\lambda, \mathcal{P}_V).$$
It initialises the list Q to be empty and sends $Pub_{\mathcal{P}_V}$ to $\mathcal{A}_{KI}$.

2. $\mathcal{A}_{KI}$ first sets $L = \epsilon$ and $o = \perp$. Then it runs lines 1 to 3 of Algorithm 2 and sets $\mathrm{MK}_{AC} = (\kappa_{\mathcal{P}_C}, \omega_{\mathcal{P}_C}, \perp, \perp)$. It also runs Algorithm 1 as given, and sends PP_{VC} and PP_{AC} to $\mathcal{A}_{VC}$.

3. Now, $\mathcal{A}_{VC}$ is provided with oracle access which $\mathcal{A}_{KI}$ can simulate as follows:

   (a) $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as specified in the construction as these rely on the underlying RPVC scheme only.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ will query for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ will follow Algorithm 4 but will make use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^V(ID)}$ or $\omega_{\lambda^V(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^V(ID), \perp)$, to which $\mathcal{C}$ will respond by adding $\lambda^V(ID)$ to the query list Q and returning $(\kappa_{\lambda^V(ID)}, \omega_{\lambda^V(ID)})$ as well as updating $L = L \cup \lambda^V(ID)$. Note that keys and secrets for computation policies are owned by $\mathcal{A}_{KI}$ already.

   $\mathcal{A}_{VC}$ chooses a security label, $\lambda^V(\mathcal{A})$, from the verification policy for itself and sends it to $\mathcal{A}_{KI}$.

4. To register $\mathcal{A}_{VC}$ as a verifier with this security label, $\mathcal{A}_{KI}$ will make use of its $\mathsf{Corrupt}$ oracle queries in the Key Indistinguishability game – that is, it will query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^V(\mathcal{A}), \perp)$ in Game 5. $\mathcal{C}$ will add $\lambda^V(\mathcal{A})$ to the list Q and return the relevant key and secret information $(\kappa_{\lambda^V(\mathcal{A})}, \omega_{\lambda^V(\mathcal{A})})$. Finally, $\mathcal{A}_{KI}$ updates the list $L = L \cup \lambda^V(\mathcal{A})$.

5. $\mathcal{C}$ now chooses a computation o uniformly at random from the space of all possible computations of F that the adversary is not authorised to verify i.e. we require $\lambda^V(o) \not\leqslant \lambda^V(\mathcal{A}_{VC})$. Note that the choice of o will dictate the input data x. If not such o may be found then the game ends with a loss for the adversary as they did not select a valid attack identity.

6. $\mathcal{A}_{KI}$ sends $\mathcal{C}$ its choice of challenge security label in the Key Indistinguishability game $v^\star = \lambda^V(o)$. Notice that this choice is allowed since $\lambda^V(o)$ is not a descendent of any security labels given in a $\mathsf{Corrupt}$ oracle query (those labels added to Q). This is necessarily so since the only $\mathsf{Corrupt}$ queries thus far have been for the adversary's label $\lambda^V(\mathcal{A})$ in

Step 4, and those labels given in a $\mathsf{Register}$ query by $\mathcal{A}_{VC}$. Now, by the condition in Step 5 we know $\lambda^V(o)$ is not less than or equal to $\lambda^V(\mathcal{A})$. Also, by the restriction on $\mathcal{A}_{VC}$'s oracle queries, it is not allowed to query for any $\lambda^V(ID) \geqslant \lambda^V(o)$ as this would constitute a trivial win.

$\mathcal{C}$ chooses a random bit b and returns $\kappa_{\lambda^V(o)}$ if $b = 0$ or chooses a random key from the keyspace otherwise, and sends this key, $\kappa^\star$, to $\mathcal{A}_{KI}$.

7. $\mathcal{A}_{KI}$ must now simulate a delegator and a computational server. It does this by first registering two such entities, C and S respectively, that are authorised to compute and verify o respectively. It sets $\lambda^V(C) = \lambda^V(o)$ and hence $\kappa_{\lambda^V(C)} = \kappa^\star$. It sets

$$SK_C = (\kappa_{\lambda^C(C)}, \omega_{\lambda^C(C)}, \kappa^\star, \perp).$$

Note that $\omega_{\lambda^V(C)}$ is not required (as this is only needed for deriving keys, and C will only need to use $\kappa^\star$).

To register S, $\mathcal{A}_{KI}$ runs

$$SK'_S \leftarrow \mathsf{RPVC.Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, S)$$

and then sets $SK_S = (SK'_S, \kappa_{\lambda^C(S)}, \omega_{\lambda^C(S)})$.

8. $\mathcal{A}_{KI}$ runs

$$(PK_F, L_F) \leftarrow \mathsf{RPVC.FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F),$$
$$(EK_{F,S}, L_F) \leftarrow \mathsf{RPVC.Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, F, L_F, S),$$
$$(\sigma'_x, VK_{F,x}, RK'_{F,x}) \leftarrow \mathsf{RPVC.ProbGen}(x, PK_F) \text{ for } x \text{ specified in } o,$$
$$\sigma_x = (\lambda^C(o), \mathsf{SE.Encrypt}(\sigma'_x, \kappa_{\lambda^C(o)}))$$
$$RK_{F,x} = (\lambda^V(o), \mathsf{SE.Encrypt}(RK'_{F,x}, \kappa^\star)), \text{ and}$$
$$\sigma_y \leftarrow \mathsf{RPVC.Compute}(\sigma'_x, EK_{F,S}, SK_S).$$

$\mathcal{A}_{KI}$ sends $(\sigma_y, VK_{F,x}, RK_{F,x}, \lambda(o), SK_{\mathcal{A}}, PK_F, L_F)$ to $\mathcal{A}_{VC}$.

9. $\mathcal{A}_{VC}$ again gets oracle access which $\mathcal{A}_{KI}$ responds to as follows:

   (a) As before $\mathsf{FnInit}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot)$, $\mathsf{Certify}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ and $\mathsf{Revoke}(\mathrm{MK}_{VC}, \cdot, \cdot, \cdot)$ can all be run as given in the construction.

   (b) $\mathsf{Register}(\mathrm{PP}_{VC}, \mathrm{MK}_{VC}, \mathrm{MK}_{AC}, \cdot, \cdot)$: $\mathcal{A}_{VC}$ queries for an identity ID and associated security label $\lambda(ID)$. $\mathcal{A}_{KI}$ follows Algorithm 4 making use of $\mathsf{Corrupt}$ oracle queries to $\mathcal{C}$ whenever it requires a value of $\kappa_{\lambda^V(ID)}$ or $\omega_{\lambda^V(ID)}$. That is, $\mathcal{A}_{KI}$ will issue an oracle query $\mathcal{O}^{\mathsf{Corrupt}}(\lambda^V(ID), \lambda^V(o))$. $\mathcal{C}$ will respond by adding $\lambda^V(ID)$ to the query list Q and returning $(\kappa_{\lambda^V(ID)}, \omega_{\lambda^V(ID)})$ if and only if $\lambda^V(ID) \not\geqslant \lambda^V(o)$ and updating the list L accordingly. $\mathcal{A}_{KI}$ can respond with computational policy keys and secrets since it already owns them.

$\mathcal{A}_{VC}$ eventually outputs a guess, y', for $F(x)$.

10. If $b = 0$ then $\kappa^\star$ is a real key corresponding to $\lambda^V(o)$ and therefore the encryption of $RK'_{F,x}$ will be performed using a valid key and will look correct to $\mathcal{A}_{VC}$. Therefore, to $\mathcal{A}_{VC}$ the system looks consistent and correct, and hence he will have non-negligible advantage δ to guess $F(x)$ correctly, as we assumed. If $y' = F(x)$, $\mathcal{A}_{KI}$ will output a guess $b' = 0$.

    Otherwise, $\kappa^\star$ is a random key and the encryption of the output retrieval key will not be correct, so $\mathcal{A}_{VC}$ may not possess the assumed advantage and $\mathcal{A}_{KI}$ should output a guess $b' = 1$.

We conclude that $\mathcal{A}_{KI}$ at least succeeds whenever $\mathcal{A}_{VC}$ succeeds i.e. with non-negligable probability δ. However, our construction assumes that the KAS is secure in the sense of Key Indistinguishability, and hence $\mathcal{A}_{VC}$ cannot have a non-negligable success probability.

□

7 Conclusion

In conclusion, we have motivated the study of access control policies in the setting of outsourced computation, and we have shown how to apply and enforce graph-based access control policies over such computations using a symmetric enforcement primitive. We provided a formal definitional framework, a provably secure construction and discussed example policies of interest. Future work will consider alternative policy forms and enforcement mechanisms, such as using authentication protocols that enforce authorisation policies [3].

Acknowledgements

We thank Christopher Dearlove, Rachel Player and Gordon Procter for helpful discussions. 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 Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst.*, 1(3):239–248, 1983.

[2] J. Alderman, C. Cid, J. Crampton, and C. Janson. Revocation in publicly verifiable outsourced computation. Cryptology ePrint Archive, Report 2014/640, 2014. `http://eprint.iacr.org/`.

[3] J. Alderman and J. Crampton. On the use of key assignment schemes in authentication protocols. *CoRR*, abs/1303.4262, 2013.

[4] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[5] M. J. Atallah, M. Blanton, and K. B. Frikken. Efficient techniques for realizing geo-spatial access control. In F. Bao and S. Miller, editors, *ASIACCS*, pages 82–92. ACM, 2007.

[6] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corporation, 1973.

[7] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. Cryptology ePrint Archive, Report 2014/224, 2014. `http://eprint.iacr.org/`.

[8] S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *TCC*, pages 499–518, 2013.

[9] M. Clear and C. McGoldrick. Policy-based non-interactive outsourcing of computation using multikey FHE and CP-ABE. In P. Samarati, editor, *SECRYPT*, pages 444–452. SciTePress, 2013.

[10] J. Crampton. Cryptographic enforcement of role-based access control. In P. Degano, S. Etalle, and J. D. Guttman, editors, *Formal Aspects in Security and Trust*, volume 6561 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.

[11] J. Crampton. Practical and efficient cryptographic enforcement of interval-based access control policies. *ACM Trans. Inf. Syst. Secur.*, 14(1):14, 2011.

[12] J. Crampton, K. M. Martin, and P. R. Wild. On key assignment for hierarchical access control. In *CSFW*, pages 98–111. IEEE Computer Society, 2006.

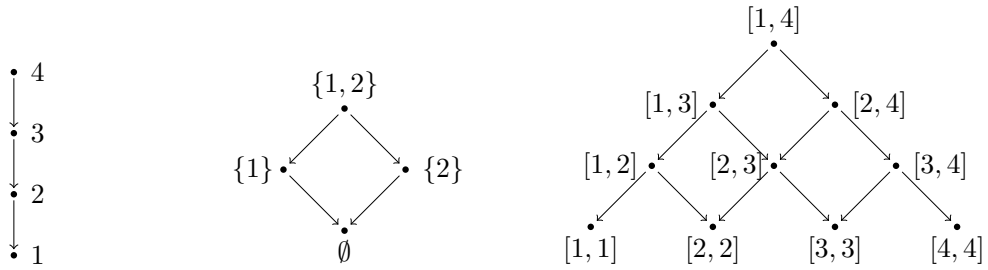
[13] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In T. Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[14] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[15] S. Goldwasser, V. Goyal, A. Jain, and A. Sahai. Multi-input functional encryption. Cryptology ePrint Archive, Report 2013/727, 2013. `http://eprint.iacr.org/`.

[16] 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.

[17] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.



(a) C_4 (b) Powerset $2^{\{1,2\}}$ (c) T_4

Figure 4: Example Hasse diagrams

A Background and Related Work

A.1 Key Assignment Schemes

Partially Ordered Set

A *partially ordered set*, or *poset*, is a set L equipped with a binary relation $\leqslant$ such that for all $x, y, z \in L$ the following conditions hold: $x \leqslant x$ (reflexivity); if $x \leqslant y$ and $y \leqslant x$ then $x = y$ (anti-symmetry); and if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$ (transitivity).

We may write $x < y$ if $x \leqslant y$ and $x \neq y$, and write $y \geqslant x$ if $x \leqslant y$. We say that x *covers* y, written $y \lessdot x$, if $y < x$ and no z exists in L such that $y < z < x$. The *Hasse Diagram* of a poset $(L, \leqslant)$ is the directed acyclic graph $H = (L, \lessdot)$ wherein vertices are labelled by the elements of L and an edge connects vertex v to w if and only if $w \lessdot v$, as in Figure 4. Figure 4 shows Hasse diagrams for the total order C_4, the powerset $2^{\{1,2\}}$ and the temporal poset T_4.

Let U be a set of entities, O be a set of resources to be protected, and $(L, \leqslant)$ be a poset of security labels. Let $\lambda : U \cup O \to L$ be a labelling function assigning a security label to each entity and object. The tuple $(L, \leqslant, U, O, \lambda)$ then denotes an *information flow policy* which can be represented by the H. Henceforth we shall refer to such policies as *graph-based access control policies*. The policy requires that information flow from objects to entities preserves the partial ordering relation; for instance an entity $u \in U$ may read an object $o \in O$ if and only if $\lambda(u) \geqslant \lambda(o)$. Equivalently, there must exist a directed path from $\lambda(u)$ to $\lambda(o)$ in H. Note that this statement is the *simple security property* of the Bell-LaPadula security model [6]. Thus an entity assigned clearance label x is prevented from accessing objects classified with label y if $y > x$. Posets of the form shown in Figure 4 have been used extensively as the basis for graph-based access control policies, notably in the Bell-LaPadula model [6] and in temporal access control [11].

Key Assignment Schemes

A *Key Assignment Scheme* (KAS) provides a generic, cryptographic enforcement mechanism for graph-based access control policies in which a unique cryptographic key is associated to each node (representing a security label) in H. Akl and Taylor [1] introduced KASs to manage the problem of key distribution by allowing a trusted center to distribute a single cryptographic key to each entity, who may then combine knowledge of this secret key with publicly available information in order to derive additional keys. More formally, a Key Assignment Scheme for an information flow policy $(L, \leqslant)$ is defined by the following four algorithms [12]:

- $\mathsf{MakeKeys}(1^\lambda, (L, \leqslant)) \to \kappa_L$: returns a labelled set of encryption keys $\{\kappa_x : x \in L\}$, denoted κ_L.

Game 5 $\mathbf{Exp}_{\mathcal{A}}^{KeyIndistinguishability}\left[1^{\lambda},(L,\leqslant)\right]$:

$\kappa_L \leftarrow \mathsf{MakeKeys}(1^{\lambda},(L,\leqslant))$
$\omega_L \leftarrow \mathsf{MakeSecrets}(1^{\lambda},(L,\leqslant))$
$Pub_{(L,\leqslant)} \leftarrow \mathsf{MakePublicData}(1^{\lambda},(L,\leqslant))$
$Q = \epsilon$
$v^{\star} \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Corrupt}}(\cdot,\perp)}(Pub_{(L,\leqslant)})$
for all $v_i \in Q$ **do**
  if $v^{\star} \in desc(v_i)$ **then**
    return 0
  end if
end for
$b \xleftarrow{\$} \{0,1\}$
if $b = 0$ **then**
  $k^{\star} = k_{v^{\star}}$
else
  $k^{\star} \xleftarrow{\$} \mathcal{K}$
end if
$b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Corrupt}}(\cdot,v^{\star})}(k^{\star})$
if $b' = b$ **then**
  return 1
else
  return 0
end if

Oracle Query 3 $\mathcal{O}^{\mathsf{Corrupt}}(v_i, v^{\star})$:

if $v_i \in Anc(v^{\star})$ **then**
  return $\perp$
end if
$Q = Q \cup v_i$
return $(\kappa_{v_i}, \omega_{v_i})$

- $\mathsf{MakeSecrets}(1^{\lambda},(L,\leqslant)) \rightarrow \omega_L$: returns a labelled set of secret values $\{\omega_x : x \in L\}$, denoted ω_L.
- $\mathsf{MakePublicData}(1^{\lambda},(L,\leqslant)) \rightarrow Pub_{(L,\leqslant)}$: returns a set of data $Pub_{(L,\leqslant)}$ that is published by the trusted setup authority
- $\mathsf{GetKey}(x, y, \omega_x, Pub_{(L,\leqslant)}) \rightarrow \kappa_y$: takes two nodes, $x, y \in L$, the secret information for x, ω_x, and the public information, $Pub_{(L,\leqslant)}$ and returns κ_y if $y \leqslant x$.

A well-known KAS construction (known as an *iterative key encrypting* (IKE) KAS [12]) publishes encrypted keys. In particular, for each directed edge (x, y) in the Hasse diagram, $\{\kappa_y\}_{\kappa_x}$ is published. Then for any $x > y$, there is a (directed) path in the Hasse diagram from x to y and the key associated with each node on that path can be derived (in an iterative fashion) by an entity that knows κ_x. A survey of existing generic schemes is given in [12] whilst further details of temporal access control and interval-based schemes can be found in [11] and [5]. A fundamental security property of a KAS is to be secure against *key indistinguishability* [4]: that is, given a set of keys $\kappa_{x_1}, \ldots, \kappa_{x_n}$, an adversary should not be able to distinguish between the key for a challenge node $x^{\star}$ (not a descendent of any x_i) and a randomly chosen key. This property is crucial if derived keys are to be used in other cryptographic protocols.

A.2 Verifiable Outsourced Computation

The concept of non-interactive verifiable computation was introduced by Gennaro et al. [13] 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 [13]:

1. C computes evaluation information EK_F that is given to S to enable it to compute F (pre-processing)
2. C sends the encoded input σ_x to S (input preparation)
3. S computes $y = F(x)$ using EK_F and σ_x and returns an encoding of the output σ_y to C (output computation)
4. C checks whether σ_y encodes $F(x)$ (verification)

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.

Parno et al. [16] introduced *Publicly Verifiable Computation* (PVC). The operation of a Publicly Verifiable Outsourced Computation scheme is illustrated in Figure **??**. In this setting, a single client C_1 computes EK_F, as well as publishing information PK_F that enables other clients to encode inputs, meaning that only one client has to run the expensive pre-processing stage. Each time a client submits an input x to the server, it may publish $VK_{F,x}$, which enables any other client to verify that the output is correct. It uses the same four algorithms as VC but KeyGen and ProbGen are now required to 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.

Clear et al. [9] discussed the introduction of access control to outsourced computation, but only considered policies over delegators (assuming semi-honest servers) in a non-verifiable, multi-client outsourced computation setting. Their construction used a homomorphic ciphertext-policy ABE scheme and fully homomorphic encryption. Our work uses KP-ABE and symmetric enforcement primitives, and considers policies over servers and verifiers as well in a publicly verifiable setting.

A.3 Other Related Work

Gennaro et al. [13] formalized the problem of *non-interactive* verifiable computation in which there is only one round of interaction between the client and the server each time a computation is performed and introduced a construction based on Yao's Garbled Circuits [17] which provides a "one-time" Verifiable Outsourced Computation allowing a client to outsource the evaluation of a function on a single input. However it is insecure if the circuit is reused on a different input and thus this cost cannot be amortized, and the cost of generating a new garbled circuit is approximately equal to the cost of evaluating the function itself. To overcome this, the authors additionally use a fully homomorphic encryption scheme [14] to re-randomize the garbled circuit for multiple executions on different inputs. In independent and concurrent work, Carter et al. [7] introduce a third party to generate garbled circuits for such schemes but require this entity to be online throughout the computations and models the system as a secure multi-party computation between the client, server and third-party. We do not believe this solution is practical in all situations since it is conceivable that a trusted entity is not always available to take part in computations, for example in the battlefield scenario discussed in Section 1. Here, the KDC

could be physically located within a high security base or governmental building and field agents may receive relevant keys before being deployed, but actual computations are performed using more local available servers and communications links. It may not be feasible, or desirable, for a remote agent to contact the headquarters and maintain a communications link with them for the duration of the computation. In addition, the KDC could easily become a bottleneck in the system and limit the number of computations that can take place at any one time, since we assume there are many servers but only a single (or small number of) trusted third parties.

Some works have also considered the multi-client case in which the input data to be sent to the server is shared between multiple clients, and notions such as input privacy become more important. Choi et al. [8] extended the garbled circuit approach [13] using a proxy-oblivious transfer primitive to achieve input privacy in a non-interactive scheme. Recent work of Goldwasser et al. [15] extended the construction of Parno et al. [16] to allow multiple clients to provide input to a functional encryption algorithm.