Enforcing Confinement in Distributed Storage and a Cryptographic Model for Access Control

Shai Halevi[*] Paul A. Karger[*] Dalit Naor[†]

June 8, 2005

Abstract

This work is concerned with the security of the standard T10 OSD protocol, a capability-based protocol for object stores designed by the OSD SNIA working group. The Object Store security protocol is designed to provide access control enforcement in a distributed storage setting such as a Storage Area Network (SAN) environment. In this work we consider in particular the ability of the OSD protocol to enforce *confinement*, which is the property that even misbehaving participants can not leak secret information across predefined boundaries.

We observe that being a "pure capability" protocol, the plain vanilla OSD protocol is incapable of enforcing confinement. We show, however, that given a trustworthy infrastructure for authentication and secure channels, the protocol can be used in a manner that achieves the desired property (and does not require any change in the message format). Thus we demonstrate that object stores can in principle be used in a standard fashion in applications that require protection against leakage of secret data.

Having identified a problem and proposed a solution, we proceed to prove formally that the proposed protocol indeed meets all its security goals. In the process we refine common cryptographic models in order to be able to reason about confinement, and then devise a precise model for a distributed capability-based access-control mechanism. To our knowledge, this is the first time such a model for access-control is defined, and defining it highlights what can and cannot be achieved by such mechanisms.

[*]IBM T. J. Watson Research Center, Hawthorne, NY, USA. shaih@alum.mit.edu, karger@watson.ibm.com.
[†]IBM Haifa Research Laboratory, Haifa, Israel. dalit@il.ibm.com

Contents

1 Introduction

Access control is the process of determining what objects can be accessed by what subjects (alternatively what information can be sent to what processes). In particular, secrecy models for access control are concerned with ensuring that secret objects are not leaked to subjects that are not cleared to see them. The confinement problem, defined by Lampson [Lam73], is to determine whether there exists a series of operations that will ultimately leak information to an unauthorized user. A system is said to enforce confinement if no such leakage can occur.

This work is concerned with the security of the standard T10 OSD security protocol [Web04], and in particular its ability to enforce confinement. Networked Storage is a classic example of a distributed system that needs an access-control mechanism. The distribution of storage over the network allows a higher level of data sharing and better manageability, but requires protection of the data. For networked file systems and Network Attached Storage (NAS) access to data is always mediated by the file server (or the NAS box), which makes it possible for these "choke points" to enforce the access control policy. This should be contrasted with Storage Area Networks (SANs), whose main benefit is exactly the removal of this single "choke point". In principle, SANs allow each client non-mediated access to the data, thus removing the centralized file server from the critical I/O path. This means, however, that there is no longer a single entity in the system that is capable of mediating all accesses and enforcing the access-control policy. A SAN environment may still have a centralized server that handles locking, placement of data, etc. But the actual I/O operations are done over the storage area network, with the client directly accessing the storage device. A misbehaving client can therefore completely bypass the centralized server (and every access-control decision that is made by that server), and instead go directly to the storage devices.

Object Storage technology addresses exactly this problem by providing an architecture and a protocol for enforcing access control in a SAN environment. The object store protocol is a capability-based protocol. The common setting is of a client that first contacts the decision-making server with an access request for some object, and later contacts the keeper of the object for the actual access. The decision maker provides the client with a "cryptographically secured" capability (or credential) that the client can later present to the object server that keeps the object, and the object server can check the validity of the credential before granting access.

Since the I/O path between the client and the object server is performance critical, an important objective of the protocol is to allow efficient validation of the request by the object store. In particular, the Object Store Device (OSD) security protocol from the T10 standard *does not require* that the object server knows the client's identity on every request. Rather, the object server only needs to verify that the credential was granted by the trusted security/policy server, and the request and credentials have not been modified. This makes it possible for a client process to freely delegate a legitimate credential to other client processes. (In fact, credential delegation was viewed as a desired feature of the object store protocol, as it allows an application to separate obtaining the credentials from using them.)

We observe, however, that the unrestricted delegation makes the OSD security protocol an instance of a "pure capability system", and it is well known that such systems are inherently incapable of enforcing confinement. On the other hand we show that when a trustworthy authentication infrastructure is available, a small modification to the protocol can transform the (CAPKEY method of the) OSD security protocol from a "pure capability system" to one that enforces confinement. This is achieved without affecting the underlying message format. Essentially, the security server needs to embed the client name in the capability (and the capability already has a field that can be used for that purpose), and the object server needs to compare the name on the capability to the name associated with the channel on which the request arrived. This fix can be implemented as a vendor-specific extension of the protocol and thus can be used with standards-compliant OSDs.

After describing this solution we proceed to prove formally that it indeed solves the problem. Since the OSD protocol relies on cryptography to achieve its goals, we need to analyze it in some cryptographic model. We thus provide a precise definition of a "secure capability-based access-control mechanism". Roughly, a mechanism is deemed secure if it looks just like a trusted party that both issues the capabilities and then checks them.

However, before we can formalize this idea and prove security of the protocol, we must make some modification to the workings of common cryptographic models. Specifically, the standard convention in most models is that all the misbehaving participants are coordinated and controlled by a single entity (called *the adversary*). Clearly, this convention makes it impossible to reason about confinement.[1] We therefore describe a different convention (in Section 4.1) that more carefully accounts for communication among misbehaving players and between players and "the network". Then we can prove that the (CAPKEY method in the) OSD protocol indeed realizes our notion of security. Also, we prove that our notion of security is strong enough to support confinement (by adapting to our setting the definition of probabilistic non-interference due to Backes and Pfitzmann [BP04]).

WHAT'S NEW. We see two contributions in this paper. On a practical level, we identified a problem with respect to confinement in a plain vanilla instantiation of the OSD protocol and show how it can be fixed without changing the message format. On a theoretical level, we provide a robust cryptographic definition for security of a capability-based access-control mechanism, and prove that the CAPKEY method of the OSD protocol realizes that definition. Moreover, this definition is refined enough to capture the difference between the vanilla protocol that cannot enforce confinement and the modified protocol that can.

PAPER ROADMAP. The paper is organized as follows. Section 2 describes relevant access-control concepts. Section 3 describes object storageand the OSD protocol itself, explain why the OSD protocol does not enforce confinement as is, and how it can be used in a manner that enforces confinement. In Section 4 we formulate our notion of security by defining a specific cryptographic model for capability-based access-control, and in the appendix we sketch a proof of security for the modified OSD protocol.[2] Finally, in Section 5 we demonstrate that our security notion is sufficient to enforce confinement.

Sections 2 and 3 are sufficient to understand the first contribution of the paper regarding confinement in distributed storage. Sections 3.3, 4, and 5 describe our definitional contributions.

Acknoledgements

The first author thanks Ran Canetti, Yael Tauman Kalai, and Manoj Prabhakaran for enlightening discussions about formal modeling of information-flow considerations in cryptographic settings.

2 Access control

Access-control models can be partitioned into two major categories: secrecy models that are concerned with ensuring that secret objects are not leaked to subjects that are not cleared to see them,

[1]The security of the CAPKEY method was already analyzed by Azagury et al. [ACF$^+$02], but that analysis was carried out in a common cryptographic model, and therefore did not capture issues of confinement.

[2]We comment that the proof itself is rather straightforward. It is specifying the model that is non-trivial.

and integrity models that deal with ensuring that potentially harmful objects are not accessed by vulnerable programs.[3] In this work we concentrate on the simplest case of secrecy models, and we will not consider integrity models any further.[4] For a survey on security models, see in [Lan81].

2.1 Preventing information disclosure

Lampson has defined in [Lam73] the *confinement problem* as determining whether there exists a series of operations that will ultimately leak information to an unauthorized user. Although the generic form of this problem is undecidable [HRU76], the focus in most access-control systems is on constructing specific systems for which it can be shown that no such unauthorized leakage occurs.

Preventing attempts of outsiders to directly gain access to unauthorized information is fairly starightforward in principle. However, the problem then becomes preventing leakage by "unsuspecting insiders". Specifically, many programs in the system may contain modifications that, when executed by legitimate users, may attempt to leak information to other unauthorized users. (Such surreptitious modifications are called Trojan horses, viruses, worms, etc.) The issue is preventing leakage of secret objects to other unauthorized processes, even if the process holding these objects is infected with a Trojan horse. (See Lipner [Lip75]).

In a distributed system, a Trojan horse can leak information via either *network channels*, *covert storage channels*, *covert timing channels*, or *side channels*. Using a network channel is the simplest type of leakage, consisting of simply sending this information over the network. Information can be leaked through a covert storage channel by changing the values of any of the state variables of the system. (Examples of potential covert storage channels include contents of files, names of files, amount of disk space used, etc.) Both network and covert storage channels are characterized by the fact that the Trojan horse uses the prescribed interfaces of the system in order to leak information.

By contrast, information can be leaked through covert timing channels and side channels by exploiting side effects of the system that may be visible to other processes. For example, a Trojan horse could encode information into deliberate modifications of the system page-fault rate, or it can modify the time interval between sent packages to encode this information. (Other side channels may include varying the power consumption of the physical machine, or causing it to emit different types of radiation.)

The distinctions between these different types of channels can be argued at great length [Wra91], and indeed, the Trusted Computer Security Evaluation Criteria (TCSEC) [Dep85], also called the Orange Book, encouraged such distinctions by requiring storage channel protection at level B2, but not timing channel protection until level B3. However, such distinctions are of no importance to a sophisticated attacker who will use whatever techniques will work, regardless of the level of evaluation.

There have been two general techniques for restricting capabilities, so that confinement can be ensured. Either the delegation of capabilities to other users must be restricted, as done by Hydra [WLH81] and PSOS [NBF$^+$80] or the ID of the user of a capability must be checked at time of use to ensure that the capability holder is actually authorized to use the capability. Karger [Kar88] showed how checking IDs can leave traditional capability delegation unrestricted, support additional access control models (such as access control lists), and with proper caching, not increase the performance costs over traditional unrestricted capabilities. Limiting capability delegation has an additional disadvantage that it is essential to show that no possible path exists in which an

[3]A third possible category is protection against denial of service, but very few models have dealt with denial of service. Gligor's work [YG90] is an exception.

[4]For more information on integrity models, see Biba [Bib77] and Schellhorn, et. al. [SRS$^+$00].

unauthorized process can get a copy of a particular capability. This can be difficult, particularly in a distributed environment. If any failure occurs anywhere, then that capability could be used improperly. By contrast, delaying the checking until time of use (with caching to reduce performance costs) requires only showing that the checking is properly done. Furthermore, checking at time of use also makes revocation of capabilities much easier.

Most modern capability systems restrict the use of capabilities for confinement. EROS [SW00] limits delegation. IBM's iSeries does additional checks at time of use [Sol01].

Most of these systems were designed for the case of a "stand-alone machine", where the assumption is that all the different processors have access to a common (trustworthy) hardware/software platform, and that platform is capable of interfering with every access.

In this work, we are interested in a *distributed* capability systems, where objects (and reference monitors) may be geographically distributed, and need to communicate with the decision-makers over unreliable links (that may even be adversarially controlled in some cases). A number of distributed capability systems have been designed over the years, including Amoeba [Mul85] and the Monads [APW86]. Indeed, the Kerberos ticket system [SNS88] was a distributed capability system.

Most systems today use *discretionary access controls*, where the access rights to an object may be determined at the discretion of the owner of the object. (These are based on the fully general Lampson access matrix [Lam74].) Such mechanisms, however, are typically vulnerable to unauthorized disclosure of information by misbehaving processes.[5] Thus, *mandatory access controls* have been developed to deal with this problem. The distinguishing feature of mandatory access controls is that a security officer may constrain the owner of an object in determining who may have access rights to that object. Most mandatory access controls have been based on lattice security models. These models were first developed at the MITRE Corporation by Bell and LaPadula [BL73] and at Case Western Reserve University by Walter et al. [WOR$^+$74].[6] A lattice secrecy model consists of a set of *access classes* that are partially ordered. A simple example of such a lattice is the common list of sensitivity levels, unclassified, confidential, secret, and top secret.

Lattice models define a policy for ensuring confinement, consisting of two simple rules: The *simple security property* says that to read an object, the access class of the object must be less than or equal to the access of the subject trying to read it. The *confinement property* requires that to write to an object, the access class of the object must be greater than or equal to the access of the subject trying to write to it. (The effect of enforcing the confinement property is that a Trojan horse at top secret cannot leak objects to lower classes, since any object that it write will also be marked top secret.)

2.2 Capability systems

Roughly, a capability system is one where the access-control decisions are separated from the enforcement of these decisions. In our setting we have a "client process" that first contacts a "manager process" with an access request for some object, and later contacts the "server" that keeps that object for the actual access. To ensure that the server grants the request if and only if it was deemed allowed, the manager provides the client with a *capability* that the client can attach to the access request, and the server can check the capability before granting access. Capability systems were first suggested in [DVH66]. See also [AP67, Geh82, Lev83].

However, it was shown by Karger and Herbert [KH84] and Boebert [Boe84] that systems such

[5]The one exception is the strict need-to-know model developed at Case Western. [WOR$^+$74].

[6]The non-discretionary models were based on earlier work described in [Wei69] and [Lob86, pages 147–148].

as in [DVH66] are inherently incapable of solving the confinement problem. The problem arises because in these systems the possession of a capability is both necessary and sufficient to gain access to an object. Furthermore, capabilities can be freely passed between clients. To illustrate the problem, consider two Trojan horse processs, one at high secrecy level and another at low secrecy level (denoted H and L, respectively), that are cooperating in order to leak high-secrecy object to the low-secrecy spy. First, the receiver L obtains a write capability to an agreed-upon low-secrecy object O, then L copies the write-capability into the object O itself. Next, the Trojan horse H obtains a read capability to object O, and use it to read the write capability off the object O. Now H is in possession of a write capability for the low-secrecy object O, and it can copy high-secrecy objects into O, thus making these objects available to the receiver L.

There have been two general techniques for restricting capabilities, so that confinement can be ensured. Either the delegation of capabilities to other users must be restricted, as done by Hydra [WLH81] and PSOS [NBF$^+$80] or the ID of the user of a capability must be checked at time of use to ensure that the capability holder is actually authorized to use the capability. Karger [Kar88] showed how checking IDs can leave traditional capability delegation unrestricted, support additional access control models (such as access control lists), and with proper caching, not increase the performance costs over traditional unrestricted capabilities. Limiting capability delegation has an additional disadvantage that it is essential to show that no possible path exists in which an unauthorized process can get a copy of a particular capability. This can be difficult, particularly in a distributed environment. If any failure occurs anywhere, then that capability could be used improperly. By contrast, delaying the checking until time of use (with caching to reduce performance costs) requires only showing that the checking is properly done. Furthermore, checking at time of use also makes revocation of capabilities much easier.

Most modern capability systems restrict the use of capabilities for confinement. EROS [SW00] limits delegation. IBM's iSeries does additional checks at time of use [Sol01]. Most of these systems were designed for the case of a "stand-alone machine", where the assumption is that all the different processors have access to a common (trustworthy) hardware/software platform, and that platform is capable of interfering with every access.

In this work, we are interested in a *distributed* capability systems, where objects (and reference monitors) may be geographically distributed, and need to communicate with the decision-makers over unreliable links (that may even be adversarially controlled in some cases). A number of distributed capability systems have been designed over the years, including Amoeba [Mul85] and the Monads [APW86]. Indeed, the Kerberos ticket system [SNS88] was a distributed capability system.

3 Object storage and the OSD protocol

Networked Storage allows clients to access storage over the network, as well as storage devices to be connected over a storage area network (SAN). While this provides a higher level of data sharing, it also necessitates protection of the data. In networked file systems and network-attached storage (NAS), data sharing and coordination among multiple clients is mediated by the file server or the NAS box. These "trusted entities" are therefore capable of making the access-control decisions as well as enforcing them.

Storage Area Networks go one step further in that they allow in principle *direct access* to the data by the client, thus removing the centralized file server from the critical I/O path. A client may first approach a centralized entity (e.g. an NFS server) for locking, placement of the data etc., but then it performs the I/O operation over the storage area network by directly accessing the storage device. Thus, the entity that makes the access control decision is no longer in a position ot enforce it. Moreover, in common SANs today, the storage device is not capable of any access control, which leaves the storage system wide open to attacks by misbehaving clients. Today, this issue is handled mostly my means of physical security, with Fibre Channel SANs that are deployed in a relatively closed environment. But as SANs evolve from Fibre Channel to IP networks (e.g. via iSCSI), securing shared data is likely to become a key factor. Object Storage technology addresses exactly this problem.

What is an object store? An object store (ObS) or Object Storage Device (OSD), is a new abstraction for a storage device that moves low-level storage functions into the storage device itself, raising the level of abstraction presented by the storage device to its users. Instead of presenting the abstraction of a logical array of unrelated blocks, an object store appears as a collection of objects. An individual object is a container of storage exposing an interface similar to a file. Users of an object store (e.g., the file system) operate on data by performing operations such as creating an object, reading/writing at a logical location in the object, and deleting the object, all using a standard object interface [SNI].

Increasing the access granularity from blocks to objects allows the object store to also *enforce access at the object level.* Namely, the object store may allow or disallow an operation (like read/write) initiated by a certain client to be performed on an object. The ability to enforce access control at the storage device is a key characteristics of an object store, allowing non-mediated shared access to shared storage in a secure manner.

The concept of object storage was originated in the Network-Attached Storage Devices (NASD) project [GNA+97, GNA+96] at CMU, with a security model by [Gob99], and evolved significantly since then. The first standardization effort of an OSD specification is embodied over the SCSI protocol [SNI]. It is realized as a new set of SCSI commands [Web04], which became an approved T10 standard in September of 2004. This standard defines a security model and specifies a security protocol that accompanies every OSD command.

3.1 The OSD security protocol

The object store security model is a capability-based access control system composed of three types of entities: clients/hosts, object stores/servers, and a security/policy manager. Below we simply refer to them as *clients*, *servers*, and *the manager.* The object store trust model assumes that the servers are trusted components that maintain integrity for the data while stored. The manager is also a trusted component that implements a certain access control policy. The application clients, however, are untrusted, and the main security goal of the protocol is to prevent misbehaving clients from accessing objects that they are not entitled to.

Since the I/O path between the client and server is performance critical, an important objective of the protocol is to allow the efficient validation of access request by the object store. To enforce access-control, all the commands to the object store must be accompanied by a valid credential that allows the host to perform the requested operation. The object-store standard [Web04] distinguishes between a *capability* (which is just a set of rights to perform operations on objects) and a *credential* which is a cryptographically secured capability. The standard also defines the set of allowed operations, namely Create object, Remove object, Read, Write, Append, Set Attribute, Get Attribute, and Set Key. There are other device management commands that are not relevant to our discussion.

In the protocol, the manager generates credentials for authorized clients, clients send credentials with their commands, and servers validate the credentials presented by clients before granting access. The OSD security protocol specifies how to bound the capability to the request so that the object store can verify that the request is authorized by the manager. The standard protocol defines three different methods to perform the validation, depending on what network security infrastructure is assumed. In this work we only consider the CAPKEY method [Web04, 4.10.4.3], that assumes a network security infrastructure for secure channels (such as IPSec). The relevant parts of the *OSD T10 standard protocol* specify the format and semantics for the credential, and also the format and semantics of commands between clients and servers. (The standard protocol views the manager as yet another client with certain privileges.)

Capabilities and credentials. The basic structure that is manipulated by the standard protocol is a capability. This is a set of fields that specify what commands are allowed, on what storage construct, expiration time, etc. The exact structure is specified in [Web04, 4.9.2], and can be described as

$$\mathit{Cap} \equiv [\texttt{ExpiryTime}, \texttt{Audit}, \texttt{Discriminator}, \texttt{ObjCreationTime}, \texttt{ObjType}, \texttt{Permissions}, \texttt{ObjDescriptor}],$$

where `ExpiryTime` is the expiration time, `ObjCreationTime`, `ObjType`, and `ObjDescriptor` identify the storage construct to which this capability applies, `Discriminator` is a nonce, and `Permissions` encodes the set of allowed operations on the storage construct. The `Audit` field is a 20-byte field that is described as a "vendor specific value that the security manager may use to associate the capability and credential with a specific application client" [Web04, 4.9.2.2.1]. We assume for the moment that the `Audit` field is set to zero.

The standard also defines a credential as a pair $\mathit{Cred} \equiv [\mathit{Cap}, \mathit{CKey}]$, where Cap is a capability as above and CKey is a secret information that is associated with the capability and defined as $\mathit{CKey} \equiv PRF_K(\mathit{Cap})$. Here PRF is a pseudo-random function (specifically HMAC-SHA1), and K is a secret key that is shared between the manager and the server on which the storage construct resides.[7]

Access requests. The standard also specifies the format and semantics of commands to the servers. There are several types of commands, depending on the specific operations that are requested, but for our purposes we only distinguish between commands that are issued by a "regular client" and commands that are issued by (a client that represents) the manager. The CAPKEY method stipulates that these commands are sent over secure channels such as given by IPSec.

Client commands. The exact format of these commands is defined by the OSD T10 standard [Web04, 5.2]. The client appends two new fields to every command: a capability as above that permits the requested operation, and a validation tag that is computed using the CKey field in the credential. Specifically, in the CAPKEY method the validation tag is computed as $\mathit{VTag} \equiv MAC_{\mathit{CKey}}(\mathit{SecToken})$, where MAC is a keyed message-authentication-code (which is also implemented using HMAC-SHA1), and $\mathit{SecToken}$ is a value that is unique to this combination of client, server, and the particular link on which they communicate [Web04, 7.5.3].[8]

The extended request therefore has the structure $[\mathit{Request}, \mathit{Cap}, \mathit{VTag}]$. Upon receipt of an extended request the server computes $\mathit{CKey} = PRF_K(\mathit{Cap})$ and verifies that the `ExpiryTime` in the capability Cap is still in the future, that the validation tag is correct, $\mathit{VTag} = MAC_{\mathit{CKey}}(\mathit{SecToken})$ and that the `Permissions` field in Cap matches the request. It processes the request if all three conditions are met, and rejects it otherwise.

[7] More accurately, in the OSD protocol there is a hierarchy of secret keys shared between the object store and the security manager. Here, the term K is used in a generic manner to denote any key in the hierarchy.

[8] The standard requires that *SecToken* be a random value, but in theory it could also be a nonce.

Manager commands. Two of the commands that are standardized by the OSD protocol can be used by the manager to revoke credentials: these are the SET ATTRIBUTE and SET KEY commands. The OSD protocol as standardized allows any client to issue these commands if it has a credential for them, but in a "reasonable deployment" there will be a specialized client that is run at the manager site, and only this client will ever get such credentials.[9] In this paper we therefore consider only such "reasonable deployments", and assume that these commands are issued by the manager itself.

The SET ATTRIBUTE can be used to revoke the credential of a single storage construct (e.g., a single object). The server maintains for each object a policy/access tag value, that can be set by the SET ATTRIBUTE command. This value also appears as part of the field `ObjDescriptor` that is included in the capability [Web04, 4.9.2.2.2]. This tag is therefore considered to be part of the object ID, and changing it changes the object ID and therefore invalidates all the credentials that were issued with the previous tag value. Future capabilities that are issued after the SET ATTRIBUTE command should include the new policy/access tag value. The SET KEY command can be used to revoke all the credentials for a server, by resetting the secret key K that is shared between the security manager and the object store. [10]

3.2 Credential Delegation

We observe that the security protocol between the server and the client from above *does not* authenticate the client's identity on every request. Rather, the server only verifies that (1) the credential was granted by the trusted security/policy manager (2) the request and credentials have not been modified. This allows a client with a valid credential to pass it on to another client. Specifically, a client C that obtains a credential $[\mathit{Cap}, \mathit{CKey}]$ that was issued to another client C' can use this credential just as well. The server will grant the same access right to C as it would to C'. (In fact, *credential delegation* is a design feature of the object store protocol, and it has uses in several environments.)

However, this means that *possession of a valid credential is both necessary and sufficient to gain access to an object*, which makes the basic OSD protocol an instance of a *pure capability* protocol. As explained in Section 2, this means that the basic protocol is inherently incapable of enforcing confinement. We now show how to augment the basic protocol (without changing the message format) to make it suitable for enforcing confinement.

Recall that the capability includes a 20-byte `Audit` field that is left for "vendor specific" extensions. One can use this field for a value that identifies one specific client, thereby binding the capability to that client. For example, one could set the value of the `Audit` field to $PRF_K(\mathit{ClientName})$, where *ClientName* is a string that is unique for each client and is known to the manager and the server. Specifically, an implementation that uses a secure-channel mechanism like IPSec to authenticate the communication can use for *ClientName* the name that was used to establish the secure channels (e.g. the IPSec public key of that client). Of course, for this to work the client must use the same name to authenticate itself to the manager and to the server.

When the server validates a request, it will also check that the `Audit` field in the credential matches the client name that is associated with the secure channel on which the request arrived. This means that to delegate a credential, the client must delegate also the ability to open a secure

[9]The SET ATTRIBUTE command is not only a manager's command, it can be used by any client, depending on the attribute that is being set

[10]As we explained above, a server will typically share more than one key with the manager, and a SET KEY command may only revoke the credentials that were issued relative to one key.

channel in its name. (Roughly, this means delegating its IPSec secret key.) If we have a trustworthy implementation of the secure channel mechanism (e.g., via secure IPSec cards in a "bump-in-a-wire" configuration), then even misbehaving clients cannot delegate their IPSec secret keys, and therefore no delegation of credentials is possible.

As this does not change the message format, it allows in principle standard-compliant OSDs to be used as trusted distributed storage elements in systems that require protection against leakage of secret data. (This holds provided that the OSD communicates with its clients and manager over secure channels.)

3.3 The protocol that we analyze

We now present a somewhat simplified interface that makes the CAPKEY method of the OSD protocol more amenable to analysis. We believe that this simplified interface captures faithfully all the relevant aspects of the CAPKEY method. (But note that we only model the access-control aspects of the protocol, and completely abstract out all the storage aspects of it.) We denote this simplified protocol by $\mathcal{P}_{\text{acc}}$, and this is the protocol that we analyze formally.

Infrastructure. The protocol $\mathcal{P}_{\text{acc}}$ is designed to work over secure channels. Namely, it assumes that participants have access to a message transmission mechanism, where a participant$P$ can send messages to another participant P' in such a way that only P' gets these messages (if anyone). To use this mechanism, P must know the name of P' and it is assumed that all the participants that send messages to P' use the same name for it. In a few more details, when P sends messages to P', a "misbehaving network" can see that messages are sent (and their length) and can also drop messages, but cannot do anything else. Moreover, we assume that the channels from the manager to the servers are *reliable*, which means that the network cannot even drop messages on these channels.[11] In addition to the message transmission mechanism, the protocol $\mathcal{P}_{\text{acc}}$ also assumes that every client-server pair (C_i, S_j) share a public random value R_{ij}, to be used as `SecToken` values.[12]

Interfaces. The protocol $\mathcal{P}_{\text{acc}}$ exposes four interfaces. Specifically, a Capability message from the manager to a client, an Access request from client to server, and two messages Revoke and RevokeAll from the manager to a server. In more details, the interfaces and their intended semantics are as follows:

Capability(S_j, CAP) from the manager to client C_i, where $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$. Here n is an opaque handle, t is the expiration time of this handle, and $\mathsf{oid}, \mathsf{ops}$ denote the object and allowed operations for which this handle can be used.

Delivers (S_j, CAP) to client C_i, and also adds this pair as a valid capability in the system.

Access$(\mathsf{request}, CAP)$ from client C_i to server S_j, where $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$ as above. Delivers to server S_j either $(\mathsf{request}, (t, \mathsf{oid}, \mathsf{ops}))$ if (CAP, S_j) is a valid capability or $(\mathsf{request}, \text{"invalid"})$ otherwise (or sometimes just returns "invalid" to C_i).

[11]This assumption is needed since we need revocations to have "guaranteed arrival". What it means in practice is that if there is ever a network partition between the manager and a server during a revocation message, the manager needs to alert an operator.

[12]There is no security need for the R_{ij}'s to be random, but the standard specifies that they are, so we model them as such.

$\mathsf{Revoke}(\mathsf{oid})$ from the manager to server S_j. Does not deliver anything to S_j, but has the side effect of invalidating all the capabilities that correspond to object oid on server S_j.

$\mathsf{RevokeAll}$ from the manager to server S_j. Does not deliver anything to S_j, but has the side effect of invalidating all the capabilities of all the objects on server S_j.

(Note that these interfaces do not have a "guaranteed delivery". Namely, whenever it says "delivers X" it actually means "either delivers X or nothing at all".)

Implementation. For each object oid on server S_j that the manager knows about, it keeps a tag $\tau_{j,\mathsf{oid}}$. Each tag is initially set to zero. Similarly each server S_j keeps a tag for each object that it maintains. Given the secure channels, the protocol $\mathcal{P}_{\text{acc}}$ is as follows:

$\mathsf{Capability}(S_j, CAP)$ from the manager to client C_i, where $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$. If the manager never sent a key to S_j (see $\mathsf{RevokeAll}$ below), then it ignores this query. Otherwise, the manager computes $Audit = PRF_{K_{S_j}}(\mathsf{Audit}, C_i)$, and $Cap = (t, Audit, n, \mathsf{oid}, \tau_{\mathsf{oid}}, \mathsf{ops})$, $CKey = PRF_{K_{S_j}}(Cap)$ and sends $(CKey, Cap, S_j)$ to C_i over the secure channel.

Upon receipt of $(CKey, Cap, S_j)$, client C_i records that tuple, sets $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$, and outputs the pair (S_j, CAP) to the higher-level protocol.[13]

$\mathsf{Access}(\mathsf{request}, CAP)$ from client C_i to server S_j, where $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$. Client C_i looks up a recorded tuple $(CKey, Cap, S_j)$ with $Cap = (t, Audit, n, \mathsf{oid}, \tau_{\mathsf{oid}}, \mathsf{ops})$ that matches the fields in CAP. If found, C_i uses the public random value R_{ij} that it shares with S_j to compute $v = MAC_{CKey}(R_{ij})$, and sends to S_j the message $(\mathsf{request}, Cap, v)$. If C_i does not find Cap it returns "invalid" on its output interface.

Upon receipt of $(\mathsf{request}, Cap, v)$, server S_j computes $CKey = PRF_{K_{S_j}}(Cap)$ and verifies that $v = MAC_{CKey}(R_{ij})$. S_j also parses $Cap = (t, Audit, n, \mathsf{oid}, \tau_{\mathsf{oid}}, \mathsf{ops})$ and checks that τ_{oid} is the same as the tag value that it keeps for oid. If all the checks pass then it outputs $(\mathsf{request}, (t, \mathsf{oid}, \mathsf{ops}))$ to the higher-level protocol.[14] If some of the checks fail, then S_j outputs $(\mathsf{request}, \text{"invalid"})$ to the higher-level protocol.

$\mathsf{Revoke}(\mathsf{oid})$ from the manager to server S_j. The manager picks a new random value for the tag τ_{oid}, and sends to S_j a message $(\text{Set-Attribute}, \mathsf{oid}, \tau_{\mathsf{oid}})$ with the new tag value. Upon receipt of this message S_j updates its own tag value.

$\mathsf{RevokeAll}$ from the manager to server S_j. The manager picks a new random value for K_{S_j} and sends to S_j a message $(\text{Set-Key}, K_{S_j})$. Upon receipt of this message S_j replace its secret key with the new K_{S_j}.

In the next section we define an abstract access-control functionality $\mathcal{F}_{\text{acc}}$, and prove that the protocol $\mathcal{P}_{\text{acc}}$ with the assumed infrastructure as above securely realizes this functionality.

Theorem 1 (informal) *The protocol $\mathcal{P}_{\text{acc}}$ securely realizes the functionality $\mathcal{F}_{\text{acc}}$ in the hybrid world with trustworthy secure message transmission, with respect to environments in which the manager is never corrupted.*

[13]The higher-level protocol in our case would probably be the actual storage system, e.g. a file system.

[14]Since we do not model any storage or timing aspects, we leave it to the higher-level protocol to verify that t is consistent with the local time at the server and the request is consistent with the given $\mathsf{oid}, \mathsf{ops}$.

Comments

1. We said above that the protocol $\mathcal{P}_{\text{acc}}$ assumes secure channels, but in fact the CAPKEY method only requires that the client–server channels be authenticated. (That is, the client–server channels need not encrypt the traffic, see [ACF^{+}02]). In our case, however, we care about confinement, so we assume that these channels are both authenticated and encrypted to prevent misbehaving clients from using them to leak information.

2. The reader may notice that the abstraction $\mathcal{F}_{\text{acc}}$ in the next section has another interface PurgeCap that is not exposed by the protocol $\mathcal{P}_{\text{acc}}$ from above. Roughly, this interface allows a client to forget a capability that it got from the manager (say, it the client discovers that it is no longer valid). The protocol $\mathcal{P}_{\text{acc}}$ as above realizes the functionality $\mathcal{F}_{\text{acc}}$ with or without PurgeCap, since it just never uses this interface. But we added this extra interface to the abstraction to be able to model also protocols that forget about invalid credentials.[15]

3. In the T10 OSD standard protocol, every object belongs to a *partition*, where a partition is an independent security unit which has it own keys, so RevokeAll revokes all the capabilities of all the objects in a given partition. This means that a server S_j in $\mathcal{P}_{\text{acc}}$ is mapped to a partition in the standard protocol (rather than to an entire store).

4 A Cryptographic model for capability-based access control

Our formal model in staged in the framework for universally-composable security (UC framework) of Canetti [Can01], which is based on the paradigm of abstract-world/real-world formulation due to Goldreich et al. [GMW87]. According to the abstract/real paradigm, the formal modeling proceeds by describing two probabilistic games, referred to as the *real world* and the *abstract world*.[16] The real world is meant to capture the protocol flows and the capabilities of a "real world attacker", the abstract world is meant to capture what we think of as a secure system, and the notion of security asserts that a protocol is secure if these two worlds are essentially equivalent.

4.1 The UC framework

In the UC framework, the players in the real world are the legitimate participants (usually denoted $P_1, P_2, \ldots$), *the adversary* A that models an attacker, and *the environment* E that models "all the observable aspects" of the protocol (such as the higher-level protocols at the legitimate participants and any other activity that may happen concurrently with a run of the protocol). All these players are formally modeled as (efficient, probabilistic) message-driven programs.

The actions in this game should capture all the interfaces that the various participants can utilize in an actual run of the protocol in a deployed system. In particular, the capabilities of A in this game should capture all the interfaces that a real-life attacker can utilize in an attack on the system. For example, A can see and modify network traffic. (In fact, it is usually assumed that A *is the network*: Every bit that a legitimate participant P_i sends on the network is routed to A, and every bit that P_i receives from the network is coming from A.) Also, A can sometimes take control over some of the legitimate participants. The interfaces of E include providing all the inputs to the legitimate participants in the protocol and getting all the outputs back from them. Also, E

[15]The issue is that if a client is later corrupted, the attacker will see more data if invalid credentials are not forgotten.

[16]In the cryptographic literature, the "abstract world" is usually called the "ideal world".

is in general allowed to communicate with the adversary A. (This last aspect is meant to capture potential interactions in which the higher-level protocols are leaking things to the adversary, etc.)

In the abstract-world model, we have all the participants of the real-world model and in addition we pretend that there is a completely trusted party that is accessible to everyone, and this trusted party is performing all the tasks that are required of the protocol. To describe a specific abstract-world model, one simply writes the code that this trusted third party should run, thereby specifying the expected functionality of the protocol. This trusted third party is usually called "the abstract functionality" and denoted by $\mathcal{F}$. In this abstract world, instead of invoking the protocol on some inputs, the legitimate participants give these inputs to $\mathcal{F}$. Then $\mathcal{F}$ records the inputs, produces the correct outputs based on the specification, and these outputs are handed back to the players. In fact, the legitimate players in the abstract world are reduced to nothing more than a communication channel between E and $\mathcal{F}$, since all the work is done by $\mathcal{F}$ itself. The functionality $\mathcal{F}$ can still interact with the adversary, but only to the extent that the intended security allows. For example, it can "leak" to the adversary things that should be publicly available for anyone (e.g., public keys and ciphertexts).

We note that writing the code of $\mathcal{F}$ is often non-trivial: It is only too easy to write a functionality that describes "what we want" but is not realizable. An example is a functionality for secure message transmission that withholds from the adversary any knowledge about the fact that a message was sent. Such functionality can only be realized if the "real world" has some mechanism to prevent traffic analysis. Hence, in typical formulations for secure message transmission, the code for the trusted party specifies that the adversary should be informed whenever a message is sent (and typically the adversary is also given the power to order messages dropped).

After formally defining the real and abstract worlds, the security notion in this framework asserts that a protocol is secure if "no environment can distinguish between these world". Formally, a protocol Π is said to securely realize the functionality $\mathcal{F}$, if for any adversary A in the real-world game there exists an adversary A' in the idea-world game, such that no environment E can distinguish between interacting with A and Π in the real world and interacting with A' and $\mathcal{F}$ in the abstract world.

Hybrid models and composition. In addition to the real and abstract worlds, an important concept in the UC framework is that of a "hybrid model". A hybrid world contains elements from both the real and the abstract models. Specifically, it includes some abstract functionality $\mathcal{F}$, but it also has legitimate participants that run a real protocol (rather than the degenerate participants of the abstract world that are only channels between E and $\mathcal{F}$).

Hybrid models are used in the UC framework for several things. For example, when we have a sub-protocol Π inside a bigger protocol Γ, we can analyze the sub-protocol Π separately, proving that it realizes some functionality F_Π. Then it is sufficient to analyze Γ in a hybrid model in which calls to Π are replaced by calls to the functionality F_Π. This, in a nutshell, is the universal-composition theorem [Can01, Theorem 6]. Hybrid models are also used to describe trusted infrastructure that is assumed to be available to the players. For example, to model an execution environment in which all participants has access to real-time clocks, we can analyze the protocol in a hybrid model with a functionality that implements such real-time clocks.

Corruptions and communication. To capture the notion of misbehaving participants, we typically give the adversary the power to corrupt players. Corruptions are formally modeled by a special *corrupt interface* of the adversary. The standard convention in the UC frameowrk (as well as most all other cryptogaphic models) is that in the real world, the adversary gains full control over the corrupted players. That is, the adversary can see the entire state of the corrupted player, and can replace that state with an arbitrary state of the adversary's choosing. Thereafter, the adversary has complete control over all the interfaces of the player. In effect, a corrupted player "belongs to the adversary". (The adversary in the abstract world has the same corrupt interface, but the effect of using this interface is entirely up to the functionality $\mathcal{F}$. Namely, it is the code of $\mathcal{F}$ that tells it what to do when receiving a query $\mathsf{Corrupt}(P_i)$ from the adversary.)

It is clear, however, that this corruption convention in the real world preclude any information-flow constraints, and in particular inherently cannot be used to reason about confinement. Instead, we suggest here a convention in which when a player P is corrupted, it just sends its entire state on its output interface to the "higher level protocol" at P. Thereafter, P relinquish control to the higher-level protocol and serves just as a channel that conveys the messages from the higher-level protocol to whatever interfaces that P has to communicate over. This is justified by viewing the entire node P as being controlled by a rogue code (call it a virus V_P). When this happens, V_P can learn the state and control the actions of every protocol that runs at the node P.

Note that with this convention, corruption by itself does not provide the adversary with additional communication channels. However, if we stick to the view of "the adversary is the network" and let V_P have a direct access to A, then V_P can still send its entire state to A and no confinement is possible. To prevent this, we modify the formal model by denying the participants direct access to the adversary/network. Of course, to be of any use the participants should be given other means of communication. This is done by working in an appropriate hybrid model, where participants have access to some functionality that implements a mediated network access. In our case, we use the "trusted communication mechanism" that is assumed by the protocol $\mathcal{P}_{\mathrm{acc}}$ from Section 3.3.

4.2 Our real-world model

The participants in our real-world model (other than the environment E and the adversary A) are a set of clients $C_1, C_2, \ldots, C_n$, a set of servers $S_1, S_2, \ldots, S_m$, and the manager M. As explained above, the participants C_i, S_j and M in our real-world model do not have direct access to the network, and instead they have access to a functionality that implements a mediated access to secure channels.

This functionality, denoted $\mathcal{F}'_{\mathrm{smt}}$, is very similar to the standard functionality of secure message transmission (see, e.g., [Can01, Section 6.3]), except that it treats "corrupted" participant the same as "uncorrupted" ones. Namely, *it forces all the participants to send their messages over secure channels*, even if they are corrupted and would like to leak their secrets in the clear over the network.[17] The formal description of that functionality is in Figure 1. Two other interesting aspects of it are discussed next:

- $\mathcal{F}'_{\mathrm{smt}}$ *forbids direct communication* between clients or between servers. That is, we model a communication infrastructure where clients cannot directly communicate with each other, but only with servers. They can still use the storage system as a communication medium, but cannot directly communicate via the network. We chose this model only to simplify the presentation, and because it is sufficient to express the concerns that we address in this work.

  We note that the model is easy to extend to a setting where participants can communicate with each other under some information-flow restrictions that are set by a policy. (Of course, the access-control policy and the communication policy have to agree on the same information-flow restrictions, or else you get no confienment.)

[17] As explained earlier, this functionality can be realized in practice by using IPSec in a "bump-in-the-wire" configuration where a trustworthy hardware maintains all the keys and does all the cryptography.

Secure-transmission functionality, $\mathcal{F}'_{\text{smt}}$

$\mathcal{F}'_{\text{smt}}$ interacts with the adversary A', n clients $C_1, \ldots, C_n$, m servers $S_1, \ldots, S_m$ and an manager M.
When receiving a message (S_j, X) from M, $\mathcal{F}'_{\text{smt}}$ reports $(M, S_j, |X|)$ to the adversary A' and delivers X to S_j from M.
On any other message (P', X) from participant P, $\mathcal{F}'_{\text{smt}}$ reports $(P, P', |X|)$ to the adversary A'. If A' replies with "proceed" then $\mathcal{F}'_{\text{smt}}$ does the following: If P, P' are either both clients or both servers, then $\mathcal{F}'_{\text{smt}}$ ignores this message. Else, $\mathcal{F}'_{\text{smt}}$ sends (P, X) to participant P'.

Figure 1: $\mathcal{F}'_{\text{smt}}$, message-transmission functionality for access-control.

- *$\mathcal{F}'_{\text{smt}}$ guarantees reliable delivery from the manager to the servers.* This aspect was discussed in Section 3.3. We note that extending the model to handle dropped messages from the manager to the servers is quite involved. In particular, one has to specify both in the protocol and in the functionality how the manager and servers react to such events.

Corrupted manager and servers. In our proof of security below we assume that the manager is never corrupted. Namely, we only prove that the protocol $\mathcal{P}_{\text{acc}}$ realizes the functionality $\mathcal{F}_{\text{acc}}$ with respect to the restricted "real world" model in which the manager is never corrupted. On the other hand, the clients and servers can be arbitrarily corrupted.

Common random string. Although there is no real need for it, the standard specifies that the system must use random values for the `SecToken`'s between clients and servers. (These values could be public, however). Hence we assume that we have in the real world a collection of values R_{ij}, one for each client-server pair, and these values are publicly known. Formally, this makes the protocol $\mathcal{P}_{\text{acc}}$ rely on the common-random-string model.

4.3 The access-control abstraction

We now define our abstract-world model by formally specifying $\mathcal{F}_{\text{acc}}$, the abstract functionality for capability-based access-control. A formal specification of $\mathcal{F}_{\text{acc}}$ is given in Section 4.4.

Intuitively, we are trying to model the situation where all the client requests go through a trusted gateway that consults the access-control policy and enforces its decision. This model is similar to a trusted file server, except that the actual files can still be kept on several storage servers, and we do not try to hide what storage server is serving what request. In other words, the abstract model includes the same manager and storage servers as in the "real world", but the clients can no longer contact the servers directly. Instead, any request to access an object in the "abstract world" is presented to the functionality, who contacts the appropriate storage server only if this request should be granted.

Corrupted participants. When a participant in the abstract world is corrupted, the adversary A' alerts the functionality $\mathcal{F}_{\text{acc}}$ to the corruption and also gives $\mathcal{F}_{\text{acc}}$ some "state" to pass to that player. This represents allowing the "higher level protocol" (which would be a virus at this point) to use all the state of the "lower level" protocol (the protocol $\mathcal{P}_{\text{acc}}$ in our case).

The functionality $\mathcal{F}_{\text{acc}}$ treats corrupted and non-corrupted players similarly, except that the corrupted players have richer interfaces than the non-corrupted ones. For example, the higher-level protocol of a non-corrupted server S_j sees only the "relevant parts" of access requests from clients, and only if these requests pass the checks of the OSD protocol. In particular, the higher-level protocol is never presented the validation tag that is sent by the client as part of the request in the protocol, since this tag is for "internal consumption" by the OSD protocol itself. If the server is corrupted, however, then it can see all the requests that are sent its way (validation tag included), even those requests that were supposed to be rejected by the OSD protocol.

When a participant is corrupted, therefore, the functionality records that fact, and from then on it interacts with that participant using the "richer" interfaces. Technicality, upon corruption of a participant P, $\mathcal{F}_{\text{acc}}$ gets from the adversary A' a stateful program Π_P that can translate between the "simple" interface that $\mathcal{F}_{\text{acc}}$ knows how to handle to the "richer" interface that the corrupted player expects.

Note that just like any other participant, also the corrupted participants can send requests to the functionality, and the functionality always reports to the adversary the length of the requests that it gets. This means that even in the abstract world, *corrupted players can communicate to the adversary via timing/traffic-analysis channels.* This models the fact that in an open network, corrupted players can always communicate to the network via timing/traffic-analysis channels, and the OSD protocol does not protect against such leakage.

4.4 Formal description of $\mathcal{F}_{\text{acc}}$

$\mathcal{F}_{\text{acc}}$ is interacting with clients $C_1, \ldots, C_n$, storage servers $S_1, \ldots, S_m$, the manager M and an adversary A'.

$\mathcal{F}_{\text{acc}}$ maintains for each server S_j a list of valid capabilities, and for each client a list of capabilities "that the client knows about", both initially empty. Also, $\mathcal{F}_{\text{acc}}$ maintains a list of object counters that is initially empty, and a counter c_j for every server S_j that is initially set to zero.[18] $\mathcal{F}_{\text{acc}}$ reacts (a-synchronously) to the following queries:

Capability. A query (C_i, Q) from M where $Q = \mathsf{Capability}(S_j, (n, t, \mathsf{oid}, \mathsf{ops}))$. $\mathcal{F}_{\text{acc}}$ recalls the counter value c_j for server S_j. If $c_j = 0$ then $\mathcal{F}_{\text{acc}}$ ignores this query.[19]

Otherwise, $\mathcal{F}_{\text{acc}}$ reports $(M, C_i, |Q|)$ to A'. If A' replies with "proceed" then $\mathcal{F}_{\text{acc}}$ recalls the counter value $c_{j,\mathsf{oid}}$ for object oid on server S_j (or adds a new counter $c_{j,\mathsf{oid}} = 0$ if it is not on the list yet). $\mathcal{F}_{\text{acc}}$ sets $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$, adds (C_i, CAP) to the capability list of server S_j, adds $(S_i, CAP, c_{j,\mathsf{oid}}, c_j)$ to the list of client C_i, and delivers (S_j, CAP) to C_i from M.

Access. A query (S_j, Q) from C_i where $Q = \mathsf{Access}(\mathsf{request}, CAP)$. $\mathcal{F}_{\text{acc}}$ parses $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$, checks that the capability list of C_i contains a tuple $(S_j, CAP, \star, \star)$, and if not it returns "invalid" to C_i.

If it finds such tuple then it reports $(C_i, S_j, |Q|)$ to A'. If A' replies with "proceed" then $\mathcal{F}_{\text{acc}}$ checks that the valid capability list of S_j contains the tuple (C_i, CAP). If so, $\mathcal{F}_{\text{acc}}$ delivers $(\mathsf{request}, (t, oid, \mathsf{ops}))$ to S_j on helaf of C_i. If not, $\mathcal{F}_{\text{acc}}$ instead delivers $(\mathsf{request}, \text{"invalid"})$ to S_j on helaf of C_i.

Revocation. A query (S_j, Q) from M where $Q = \mathsf{RevokeAll}$ or $Q = \mathsf{Revoke}(\mathsf{oid})$. $\mathcal{F}_{\text{acc}}$ reports $(M, S_j, |Q|)$ to A'.

[18]The main purpose of the client capability list and the counters is to help generate the client state when the client is corrputed.

[19]This essentially represents the manager not yet sharing a key with that server.

If $Q = \mathsf{RevokeAll}$ then $\mathcal{F}_{\text{acc}}$ increases the server counter c_j and removes everything from the valid capability list of S_j.

If $Q = \mathsf{Revoke}(\mathsf{oid})$ then $\mathcal{F}_{\text{acc}}$ increases the object counter $c_{j,\mathsf{oid}}$ and removes from the valid capability list of S_j all the entries of the form $(\star, (\star, \star, \mathsf{oid}, \star))$.

Forgetting capabilities. A query $\mathsf{PurgeCap}(S_j, n)$ from C_i. $\mathcal{F}_{\text{acc}}$ removes from the list of C_i any tuple of the form $(S_j, CAP, \star, \star)$ where CAP has nonce value n.

Corruptions. The description above assumes that no one is corrupted; we need the following modifications to handle corruption: On a query $\mathsf{Corrupt}(P, \Pi)$ from A', where P is either S_j or C_i and Π a program, $\mathcal{F}_{\text{acc}}$ records the fact that P is corrupted and associates with P the program $\Pi_P = \Pi$.

If $P = S_j$ is a server then $\mathcal{F}_{\text{acc}}$ sets $s = \Pi_{S_j}(\mathsf{Corrupt}, c_j)$ and delivers $(\mathsf{Corrupt}, s)$ to S_j. If $P = C_i$ is a client, then for every tuple (S_j, CAP, c, c') in the list of C_i, $\mathcal{F}_{\text{acc}}$ sets $Cred = \Pi_{C_i}(\mathsf{Capability}, S_j, CAP, c, c')$ and delivers $(\mathsf{Corrupt}, Cred_1, Cred_2, \ldots)$ to C_i. (This represents the fact that the virus at the higher-level protocol at P can use also the state of the OSD protocol.) Thereafter, whenever a participant P submits a query (P', X), the functionality $\mathcal{F}_{\text{acc}}$ does the following:

- If both P and P' are corrupted, and if P, P' are neither noth clients nor both servers, then $\mathcal{F}_{\text{acc}}$ reports $(P, P', |X|)$ to A', and if A' replies with "proceed" then it forwards X to P' without any further processing.
- If neither P nor P' are corrupted, $\mathcal{F}_{\text{acc}}$ proceeds as described above.
- If P is corrupted and P' is not, then it runs the program Π_P to get $Q = \Pi_P(P', X)$. Then $\mathcal{F}_{\text{acc}}$ processes Q as above.
- If P is not corrupted but P' is, then $\mathcal{F}_{\text{acc}}$ proceeds similar to above, except for the following changes:

  A $\mathsf{Capability}$ query to a corrupted C_i. After inserting (C_i, CAP, c, c') to the capability list of client C_i, $\mathcal{F}_{\text{acc}}$ sets $R = \Pi_{C_i}(\mathsf{Capability}, CAP, S_j, c, c')$ and delivers R to C_i.

  An Access query by non-corrupted C_i to a corrupted S_j. If the capability list of C_i contains a tuple (S_j, CAP, c, c'), then $\mathcal{F}_{\text{acc}}$ reports $(C_i, S_j, |Q|)$ to A', and if A' replies with "proceed" then $\mathcal{F}_{\text{acc}}$ sets $R = \Pi_{S_j}(\mathsf{Access}, CAP, c, c')$ and delivers R to S_j. (If no such tuple is found then $\mathcal{F}_{\text{acc}}$ still returns "invalid" to C_i without delivering anything to S_j.)

  Queries $Q = \mathsf{RevokeAll}$ or $Q = \mathsf{Revoke}(\mathsf{oid})$ to a corrupted server S_j. $\mathcal{F}_{\text{acc}}$ reports $(M, S_j, |Q|)$ to A', increases the relevant counter c, sets $R = \Pi_{S_j}(Q, c)$ and returns R to S_j.

COMMENTS. **1.** We already remarked above that the protocol $\mathcal{P}_{\text{acc}}$ realizes the functionality $\mathcal{F}_{\text{acc}}$ only under the assumption that the manager is never corrupted. Although it is possible in principle to write a weaker notion of security that we can prove $\mathcal{P}_{\text{acc}}$ to realize without this extra assumption, this would complicate the definition quite a bit. (For example, we would have to worry about a corrupted manager that issues an invalid capability to a non-corrupted client, who later tries to use that capability with a corrupted server.) Since achieving confinement anyway depends crucially on the manager not being corrupted, we decided to forgo this more general but more complicated definition.

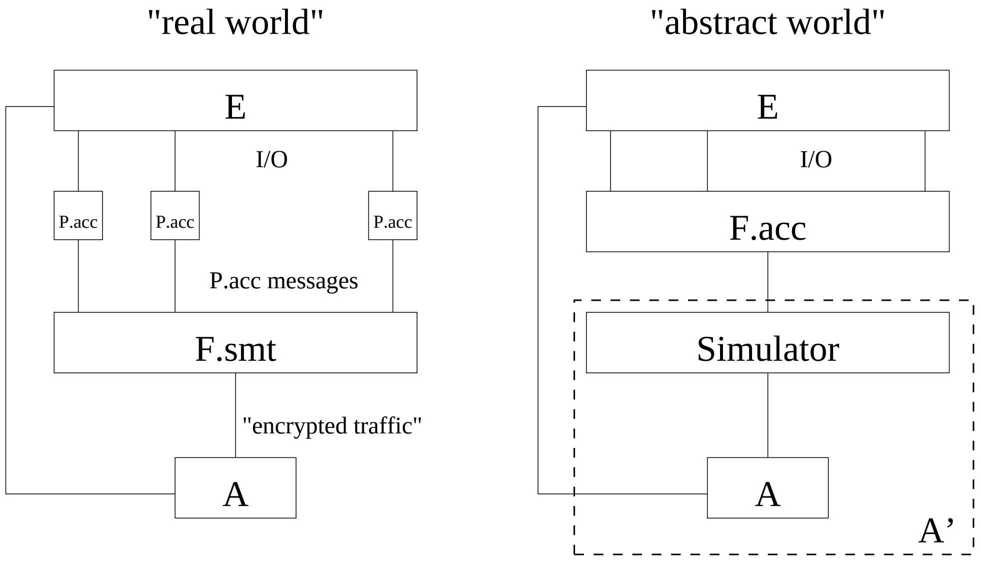

Figure 2: An illustration of the simulator.

2. The server counter c_j in the definition above is only needed to handle cases where the server S_j is corrupted. Note that if S_j is corrupted from the beginning then the environment knows all its keys, and thus when a client C_i is corrupted, the environment can check what credential was generated with respect to what key. Hence it is important that this information will be available somewhere also in the abstraction so that the simulator can simulate the state of C_i. Of course, we could get rid of the c_j's if we assume that the servers are never corrupted (just like we assumed it for the manager), but here we think that the small modification in $\mathcal{F}_{\text{acc}}$ is worth the extra generality of handling also corrupted servers.

4.5 $\mathcal{P}_{\text{acc}}$ securely realizes $\mathcal{F}_{\text{acc}}$

Now that we defined formally both models, we can finally state and prove a theorem saying that the protocol $\mathcal{P}_{\text{acc}}$ is indeed secure according to our notion.

Theorem 1 *Assume that PRF is a cryptographically secure pseudo-random function and that MAC is a cryptographically secure message-authentication code. Then the protocol $\mathcal{P}_{\text{acc}}$ securely realizes the functionality $\mathcal{F}_{\text{acc}}$ in the hybrid world with common random string and $\mathcal{F}'_{\text{smt}}$, with respect to environments in which the manager is never corrupted.*

Proof Sketch The proof is actually quite straightforward given the formal layout from above. Fix some real-world adversary A and we describe an ideal-world adversary A' such that no environment E can distinguish between interacting with A and $\mathcal{P}_{\text{acc}}$ (in the world with common random string and $\mathcal{F}'_{\text{smt}}$) and interacting with A' and $\mathcal{F}_{\text{acc}}$.

Description of A'. As usual, A' uses A as a black box, simulating for it the “real world”. Specifically, A' needs to generate the “encrypted traffic” that A expects to see from $\mathcal{F}'_{\text{smt}}$, and it needs to interact with $\mathcal{F}_{\text{acc}}$ in such a way that the I/O that E sees in both worlds is the same. See an illustration of the two worlds in Figure 2.

A' begins by choosing random R_{ij}'s and passing them to A. Then it also chooses a “master key” mk for PRF. To generate the “encrypted traffic”, every time that A' gets a message (P, P', ℓ)

from $\mathcal{F}_{\rm acc}$, it figures out the query type based on the length, then computes the length ℓ' of the corresponding message in the protocol $\mathcal{P}_{\rm acc}$ and sends to A (P, P', ℓ'), which is what A expects to see from $\mathcal{F}'_{\rm smt}$. If A replies with "proceed" then A' also replies with "proceed" to $\mathcal{F}_{\rm acc}$.

When A asks to corrupt a client C_i, A' prepares a program Π_{C_i} to pass to $\mathcal{F}_{\rm acc}$ as follows: The program Π_{C_i} is given the master key mk. When Π_{C_i} runs on an input $(\mathsf{Capability}, S_j, (n, t, \mathsf{oid}, \mathsf{ops}), c, c')$ it computes the "server key" $K_{S_j} = PRF_{mk}(\mathsf{ServerKey}, S_j, c')$, and tag $\tau_{j,\mathsf{oid}} = PRF_{mk}(\mathsf{Tag}, S_j, \mathsf{oid}, c)$,[20] then compute $Audit = PRF_{K_{S_j}}(\mathsf{Audit}, C_i)$, sets $Cap = (t, Audit, n, \mathsf{oid}, \tau_{\mathsf{oid}}, \mathsf{ops})$, computes $CKey = PRF_{K_{S_j}}(\mathsf{CKey}, Cap)$ and outputs $Cred = [Cap]$. (The collection of credentials is what $\mathcal{F}_{\rm acc}$ sends to the higher-level protocol in the corrupted C_i as the current state of the local $\mathcal{P}_{\rm acc}$ protocol.) The program Π_{C_i} also needs to translates from $\mathcal{P}_{\rm acc}$ messages that the corrupted C_i sends to queries that $\mathcal{F}_{\rm acc}$ understands, and from replies that $\mathcal{F}_{\rm acc}$ returns to $\mathcal{P}_{\rm acc}$ messages that the corrupted C_i expects. This is done in the obvious way by stripping the extra fields on the way from C_i to $\mathcal{F}_{\rm acc}$ and computing them back on the way from $\mathcal{F}_{\rm acc}$ to C_i (using the master key mk).

When A asks to corrupt a server S_j, A' prepares a program Π_{S_j} that knows the master key mk as follows: $\Pi_{S_j}(\mathsf{Corrupt}, c)$ returns the "server key" $K_{S_j} = PRF(S_j, c)$, and later Π_{S_j} translate between $\mathcal{P}_{\rm acc}$ messages and queries for $\mathcal{F}_{\rm acc}$ in the obvious way.

Indistinguishability. Proving that E cannot distinguish between the "real world" with A and the "abstract world" with A' is also quite straightforward. One first notices that the view of E in the "abstract world" is consistent: namely even if all the clients are corrupted then the values that they get, say, for *Ckey* in their credentials will always match the key K_{S_j} that S_j was supposed to be using for computing these values (and that E may learn if it corrupts S_j). The reason is that all the programs Π_P are using the same master key mk and compute all the keys in the same manner. (This is the only argument that is not entirely standard in this proof.)

It remains to show that the environment cannot distinguish the various "random fields" that it sees from random, and (most importantly) that it gets the same responses from $\mathcal{P}_{\rm acc}$ on access requests as it would from the functionality $\mathcal{F}_{\rm acc}$. This is proven by going through a few "mental experiments" that describe related games: we begin by replacing the applications of PRF with applications of a truly random function (first only w.r.t. the master key, then with respect to other keys as well), and proving that distinguishing the simulation from the resulting mental experiments implies breaking the security of the PRF function.

Then we argue that $\mathcal{P}_{\rm acc}$ grants access if and only if $\mathcal{F}_{\rm acc}$ does. (This, after all, is the main point of this exercises.) Clearly, if $\mathcal{F}_{\rm acc}$ grants accesss then all the relevant fields match, in which case the validation tag in $\mathcal{P}_{\rm acc}$ will be accepted. In the other direction, we argue that the case in which $\mathcal{P}_{\rm acc}$ grants access but $\mathcal{F}_{\rm acc}$ refuses corresponds to breaking the security of the MAC function. (This last argument is nearly identical to the one in [ACF+02], where the same protocol was analyzed.) ■

5 $\mathcal{F}_{\rm acc}$ can enforce confinement

Having specified an abstraction and proved that $\mathcal{P}_{\rm acc}$ realizes that abstraction, we now show that it is possible to use the abstraction to enforce confinement, and so it follows that $\mathcal{P}_{\rm acc}$ can be used to enforce confinement. Specifically, we consider a "hybrid world" in which all parties have access to the abstract access-control functionality $\mathcal{F}_{\rm acc}$ as defined in Section 4. We consider a system

[20]If $c = 0$ then Π_{C_i} sets $\tau_{\mathsf{oid}} = 0$ rather than computing it with PRF.

that uses $\mathcal{F}_{\text{acc}}$ to implement a distributed storage system, with the manager using a lattice-based access-control policy, following both the "simple security" and the "confinement" rules as described in Section 2.1. We further assume that:

(a) Neither the manager nor any of the storage servers are corrupted, and

(b) The high-secrecy data is not leaked via timing or traffic-analysis channels, nor via the protocols above the storage system.

Under these conditions, we prove that the system achieves confinement.

The rest of this section is organized as follows: In Section 5.1 we describe a toy distributed storage system that implements access control via access to $\mathcal{F}_{\text{acc}}$. Then in Section 5.2 we formally define confinement in our cryptographic setting, essentially using the definition of "probabilistic non-interference" due to Backes and Pfitzmann [BP04, BP03] as adapted to our settings, and prove that under the conditions (a) and (b) from above, our toy storage system achieves confinement.

5.1 $\mathcal{P}_{\text{st}}$ — a toy storage system

Recall that in our protocol $\mathcal{P}_{\text{acc}}$ we completely ignored the storage aspects of the OSD protocol, concentrating only on its access-control aspects. We now describe a very simple distributed storage system that uses the interfaces of $\mathcal{P}_{\text{acc}}$ (or $\mathcal{F}_{\text{acc}}$) to do access control. Formally, we describe a "storage protocol" that operates on top of the "access control mechanism" $\mathcal{F}_{\text{acc}}$. We denote this protocol $\mathcal{P}_{\text{st}}$. We believe that the protocol $\mathcal{P}_{\text{st}}$, although simplistic, is consistent with the storage aspects of the OSD standard [Web04] (i.e., $\mathcal{P}_{\text{acc}}$ can be implemented using the OSD commands). But we did not verify this carefully.

Infrastructure. The protocol $\mathcal{P}_{\text{st}}$ assumes that all the participants have access to the access-control functionality $\mathcal{F}_{\text{acc}}$, as well as to the secure message transmission mechanism $\mathcal{F}'_{\text{smt}}$ that can be used to pass storage messages.

Interfaces. The interfaces of $\mathcal{P}_{\text{st}}$ are roughly those of a (flat) file system. In principle only the clients need to get inputs from the higher level protocols (since the manager and servers are only "implementation details" at this level), and these inputs are the usual variety of Open, Read, Write, etc. In practice there will be interfaces also for the manager and servers, to handle administrative tasks, quotas, revocations, etc., but we omit all of these for simplicity.

The interfaces of $\mathcal{P}_{\text{st}}$, all of them invoked by clients, are the following: Create returns an identifier oid for a new object; $\mathsf{Open}(\mathsf{oid}, \mathsf{ops})$ returns a handle n that can be used to access the object; $\mathsf{Read}(n)$ returns the content of the object whose handle is n; $\mathsf{Write}(n, \mathsf{data})$ replaces the content of the object whose handle is n by the content of the string data; and $\mathsf{Delete}(n)$ removes the object hose handle is n from the storage system.

We comment that $\mathcal{P}_{\text{st}}$ is only a toy protocol, and it is not at all usable in practice. For example, it assumes that every operation is atomic, with a Read returning the entire object and a Write overwriting the entire object. Also many aspects of the implementation as described below are ridiculous from a practical point of view (e.g., some crucial acknowledgments are never returned). Still, it serves to demonstrate the main point of this paper, namely that the CAPKEY method of the OSD protocol can in principle be used to enforce confinement (when used with the Audit tag and over trustworthy secure channels).

Implementation. The system is parametrized by a lattice L and a "clearance mapping" $\mathsf{clr} : \{C_1, \ldots, C_n\} \to L$ that assigns a clearance level in L to every client C_i. The manager M keeps a table of objects in existence, which is initially empty. M begins by issuing $\mathsf{RevokeAll}$ queries to all the servers (since $\mathcal{F}_{\text{acc}}$ does not handle $\mathsf{Capability}$ queries when the server S_j has counter value 0). Thereafter, the interfaces are implemented as follows:

Create. Client C_i sends a message to M (using $\mathcal{F}'_{\text{smt}}$) asking to create an object.

The manager M consults its table to find the server S_j with the least occupied space. Then it makes a query $(C_i, \mathsf{Capability}(S_j, (n, t, 0, \mathsf{Create})))$ to $\mathcal{F}_{\text{acc}}$. Here 0 is a special object-ID that is used for creation of new objects, n is a random value that is used as nonce, and t is set to five minutes in the future (by the manager's local clock).

Upon receipt of the result $(S_j, (n, t, 0, \mathsf{Create}))$ from $\mathcal{F}_{\text{acc}}$, the client C_i makes up a new object-ID oid (say, at arandom), sets $\mathsf{request} = (\mathsf{Create}, \mathsf{oid})$ and $CAP = (n, t, 0, \mathsf{Create})$ and makes a query $(S_j, \mathsf{Access}(\mathsf{request}, CAP))$ to $\mathcal{F}_{\text{acc}}$.

When $\mathcal{F}_{\text{acc}}$ delivers $((\mathsf{Create}, \mathsf{oid}), (0, \mathsf{Create}))$ to server S_j on behalf of C_i, S_j verifies that it has no existing object with ID (C_i, oid). If there is such an existing object, S_j sends a message $(\mathsf{oid}, \text{“existing”})$ to C_i using $\mathcal{F}'_{\text{smt}}$. Otherwise it sends a message $((C_i, \mathsf{oid}), \text{“created”})$ for C_i, and also send the same message to M.

Upon receipt of these messages, client C_i returns (C_i, oid) to the higher level protocol, and the manager adds the tuple $(S_j, (C_i, \mathsf{oid}), 0, \mathsf{clr}(C_i))$ to its table of objects, denoting an object with ID (C_i, oid) on server S_j with initial size zero and level $\mathsf{clr}(C_i)$ in the lattice. (If the table already has an object (C_i, oid) at server S_j then M does nothing.)

$\mathsf{Open}(\mathsf{oid}, \mathsf{ops})$, with ops a non-empty subset of $\{\mathsf{Read}, \mathsf{Write}, \mathsf{Delete}\}$. C_i sends a message to M, asking for a capability for object oid and operations ops. The manager M checks that it has an object oid on server S_j in some level ℓ in the lattice. It also checks that (a) $\ell \leq \mathsf{clr}(C_i)$ if ops includes Read, (b) $\ell \geq \mathsf{clr}(C_i)$ if ops includes either Write of Delete, and (c) that the size of the object is non-negative. (A negative size signals a deleted object.)

If all the checks succeed, then M makes a query $\mathsf{Capability}(S_j, (n, t, \mathsf{oid}, \mathsf{ops}))$ to $\mathcal{F}_{\text{acc}}$ for C_i, where n is random and t is five minutes in the future. Upon receipt of the result $(n, t, \mathsf{oid}, \mathsf{ops}, S_j)$ from $\mathcal{F}_{\text{acc}}$, the client C_i sets $CAP = (n, t, \mathsf{oid}, \mathsf{ops})$, records (S_j, CAP) and returns n to the higher-level protocol.

$\mathsf{Read}(n)$. Client C_i looks for a recorded pair $(S_j, CAP = (n, t, \mathsf{oid}, \mathsf{ops}))$ with the right value of n. If no such pair is found or if ops does not include Read, it returns “invalid” to the higher-level protocol. Else it sets $\mathsf{request} = (\mathsf{Read}, \mathsf{oid})$ and makes a query $\mathsf{Access}(\mathsf{request}, CAP)$ to $\mathcal{F}_{\text{acc}}$ for S_j.

When $\mathcal{F}_{\text{acc}}$ delivers $((\mathsf{Read}, \mathsf{oid}), (t, \mathsf{oid}, \mathsf{ops}))$ to server S_j from C_i, it verifies that t is in thefuture and ops includes Read and that it has an object oid. If one of the first two checks fails it sedns $(\mathsf{Read}, \mathsf{oid}, \text{“unauthorized”})$ to C_i, and if the last check fails it sends $(\mathsf{Read}, \mathsf{oid}, \text{“absent”})$. If all succeed then S_j sends to C_i $(\mathsf{Read}, \mathsf{oid}, \mathsf{data})$ where data is the content of object oid. When C_i gets the reply from S_j, it outputs it to the higher-level protocol.

$\mathsf{Write}(n, \mathsf{data})$. Similar to above, C_i looks for a matching capability (S_j, CAP) and makes a query $\mathsf{Access}(\mathsf{request}, CAP)$ with $\mathsf{request} = (\mathsf{Write}, \mathsf{oid}, \mathsf{data})$. When S_j gets $((\mathsf{Write}, \mathsf{oid}, \mathsf{data}), (t, \mathsf{oid}, \mathsf{ops}))$ it makes the same checks with the same two error messages if they fail.

If all succeed then S_j replaces the content of object oid by the string data, and sends to the manager M a message $(\mathsf{Write}, \mathsf{oid}, |\mathsf{data}|)$. The manager records the size of object oid in its table.

$\mathsf{Delete}(n)$**.** Handled similarly to Write, except that if all goes well then S_j removes the object from its storage space and sends $(\mathsf{Delete}, \mathsf{oid})$ to the manager, who sets the new size of object oid in its table to -1.

5.2 Formalizing confinement

We now formalize our notion of confinement and then prove that $\mathcal{P}_{\mathrm{acc}}$ satisfies it. Our definition is an adaptation to our setting of the notion of "probabilistic non-interference" due to Backes and Pfitzmann [BP04, BP03]. We comment that the definitions of Backes and Pfitzmann are staged in the reactive simulatability framework of Pfitzmann and Weidner [PW00, PW01] rather than in the UC framework. However, these two frameworks are nearly interchangeable and in particular all the aspects that are of interest to us in this work behave exactly the same in both.

Before presenting the definition of confinement, we recall that the UC framework stipulates a special "environment machine" E that controls the I/O interfaces of all the legitimate participants (see Section 4.1). To define confinement we consider a setting in which the environment is split into two machines, E_H and E_L, such that E_H controls the I/O interfaces of the "high secrecy" clients and E_L controls the I/O interfaces of the "low secrecy" clients.

To formalize the requirement that no secrets are leaked via the higher-level protocols, we insist that E_H cannot send messages to E_L or to the adversary. (On the other hand, we let E_H receive messages from both the adversary and E_L, and we let E_L and the adversary exchange messages freely.) We now consider an experiment in which E_H is given a random input bit b, and eventually E_L halts with a guess b' for that input bit, and we require that the probability of $b = b'$ is not significantly more than 1/2.

A little more formally, consider a prescribed system $\mathcal{S} = \{P_1, \ldots, P_n\}$, and let $\mathcal{H} : \mathcal{L}$ be a partition of the set of participants (i.e., $\mathcal{H} \subseteq \mathcal{S}$ and $\mathcal{L} = \mathcal{S} \setminus \mathcal{H}$). We would like to say that there is no information-flow from $\mathcal{H}$ to $\mathcal{L}$ in $\mathcal{S}$ (or that $\mathcal{L}$ cannot interfere with $\mathcal{H}$) if for any adversary A and environments E_H, E_L as above, such that E_H controls the I/O of $\mathcal{H}$ and E_L controls the I/O of $\mathcal{L}$, it holds that $\Pr[b = b'] \approx 1/2$.

Traffic-oblivious systems. The condition above cannot be met in our setting where we allow traffic analysis, since E_H can leak to A the value of the bit b by sending either a short message when $b = 0$ or a long one when $b = 1$. We therefore restrict ourselves only to cases where traffic analysis is not used. We say that a complete system $(\mathcal{S}, A, E_H, E_L)$ is *traffic-oblivious* if whenever we fix the randomness that is used by the entire system except for the bit b, the traffic pattern of the execution of $(\mathcal{S}, A, E_H(0), E_L)$ is the same as the traffic pattern of the execution of $(\mathcal{S}, A, E_H(1), E_L)$.[21]

Then we say that there is no information-flow from $\mathcal{H}$ to $\mathcal{L}$ in $\mathcal{S}$ *upto traffic analysis* if for any A, E_H, E_L as above, such that E_H controls the I/O of $\mathcal{H}$, E_L controls the I/O of $\mathcal{L}$, and $(\mathcal{S}, A, E_H, E_L)$ is traffic-oblivious, it holds that $\Pr[b = b'] \approx 1/2$.

We comment that stronger definitions are certainly possible. For example we can have n random bits as inputs to E_H, allow leakage of $n-1$ bits via either the high-level protocol or timing channels, and require that the last bit will still be unguessable. We expect $\mathcal{P}_{\mathrm{st}}$ to meet these stronger notions, but formalizing these notions is very messy, so we did not attempt it at this time.

[21]The traffic pattern means who sent messages to whom and of what length. This can be made formal, but the formalization does not add anything here, so we leave it on an intuitive level.

5.3 $\mathcal{P}_{\rm st}$ achives confinement

Consider a system $\mathcal{S} = \{C_i, \ldots, C_n, S_1, \ldots, S_m, M\}$ that implements the protocol $\mathcal{P}_{\rm st}$ with respect to some lattice L and clearance mapping $\mathsf{clr} : \{C_1, \ldots, C_n\} \rightarrow L$. For every client C_i we consider a partition $\mathcal{H}_i : \mathcal{L}_i$ of the client set to those that are allowed to read what C_i writes and those that are not. Namely, $\mathcal{H}_i = \{C_{i'} : \mathsf{clr}(C_{i'}) \geq \mathsf{clr}(C_i)\}$ and $\mathcal{L}_i = \{C_1, \ldots, C_n\} \setminus \mathcal{H}_i$. Then we prove the following lemma:

Lemma 2 *For every partition $\mathcal{H}_i : \mathcal{L}_i$ of $\mathcal{S}$ as above, there is no information flow from $\mathcal{H}_i$ to $\mathcal{L}_i$ in $\mathcal{S}$ upto traffic analysis.*

Proof Sketch The proof is straightforward (mostly because there is no cryptography involved in the workings of $\mathcal{F}_{\rm acc}$). Consider a specific partition $\mathcal{H}_i : \mathcal{L}_i$, adversary A and environments E_H, E_L, such that the complete system $(\mathcal{S}, A, E_H, E_L)$ is traffic oblivious, and fix all the randomness in the system other than the input bit b of E_H. We show that the views of E_L in the cases $b = 0$ and $b = 1$ are identical, and it follows that $\Pr[b = b'] = 1/2$.

We can assume w.l.o.g. that E_L and A are the same machine (since they are allowed to talk freely and we quantify over all A, E_L). So we consider the complete system $(\mathcal{S}, E_H, E_L)$. Note that since we fixed all the randomness, then the runs of $(\mathcal{S}, E_H(0), E_L)$, $(\mathcal{S}, E_H(1), E_L)$ are deterministic. By traffic oblivious-ness it follows that what the functionalities reports to A is the same in both runs. So we need to show that also what E_L sees on the output interfaces of the clients in $\mathcal{L}$ is the same.

Assume toward contradiction that E_L sees something different on the output interfaces of some clients in $\mathcal{L}$, and consider the first event in which this happens. Since the view of E_L up to the distinguishing event if the same in both runs, it means that the inputs that the clients in $\mathcal{L}$ had upto that point where the same, so they made exactly the same queries with the same data to the functionalities.

Also, since we assume that M and the S_j's are not corrupted, then the servers in $\mathcal{F}_{\rm acc}$ enforce all the decisions of M, and these decisions are made according to a lattice-based policy, which is known to enforce confinement. Thus what the clients in $\mathcal{L}$ see as replies to their queries to all the servers and the manager is independent of the actions of clients in $\mathcal{H}$.[22]

It remains to verify that E_L does not learn anything about b when a client in $\mathcal{L}_i$ is corrupted. Notice that the only new things that E_L sees when corrupting a client C are (computed by Π_C from known information and) the counters $c_{j,\mathsf{oid}}$ and c_j in the capabilities that were issued to C. In the simplistic $\mathcal{P}_{\rm st}$ from above there are no revocations, so these values are always fixed and in particular E_L does not learn anything from them. (But this aspect remains true also for more realistic protocol, as long as revocation of capabilities for lower-secrecy objects are never influenced by the contents of higher-secrecy objects.) ■

References

[ACF$^+$02] Alain Azagury, Ran Canetti, Michael Factor, Shai Halevi, Ealan Henis, Dalit Naor, Rinetzky Noam, Ohad Rodeh, and Julian Satran. A two layered approach for securing an object store network. In *Proceedings of the First International IEEE Security in Storage Workshop*, pages 10–23, Greenbelt, MD, 11 December 2002.

[22]One can also check that the different error messages do not leak information, since the validity of capabilities is verified before checking if the object actually exists.

[AP67] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 5.1–5.10, Gatlinburg, TN, USA, October 1967.

[APW86] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.

[Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD–TR–76–372, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, April 1977.

[BL73] David E. Bell and Leonard J. LaPadula. Secure computer systems (mathematical foundations, a mathematical model, a refinement of the mathematical model). Technical Report ESD–TR–73–278, Vol. I-III, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, November 1973.

[Boe84] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the* 7^{th} *DoD/NBS Computer Security Conference*, pages 291–293, Gaithersburg, MD, USA, 24–26 September 1984. National Bureau of Standards.

[BP03] Michael Backes and Birgit Pfitzmann. Intransitive non-interference for cryptographic purposes. In *IEEE Symposium on Security and Privacy (OAKLAND"03)*, pages 140–152, 2003.

[BP04] Michael Backes and Birgit Pfitzmann. Computational probabilistic non-interference. *International Journal of Information Security*, 3(1):42–60, October 2004. Prelimniriay version in ESORICS'02.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 136–145, Las Vegas, NV, 14–17 October 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[Dep85] Department of defense trusted computer system evaluation criteria. DOD 5200.28–STD, Washington, DC, December 1985.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[Geh82] Edward F. Gehringer. *Capability Architectures and Small Objects.* UMI Research Press, Ann Arbor, MI, USA, 1982.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, New York, NY, 1987.

[GNA$^+$96] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU–CS–96–142, Carnegie Mellon

University, Pittsburgh, PA, 26 September 1996. `http://www.pdl.cmu.edu/PDL-FTP/NASD/TR96-142.pdf`.

[GNA$^+$97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Erik Riedel, David Rochberg, and Jim Zelenka. Filesystems for network-attached secure disks. Technical Report CMU–CS–97–118, Carnegie Mellon University, Pittsburgh, PA, July 1997. `http://www.pdl.cmu.edu/PDL-FTP/NASD/CMU-CS-97-118.pdf`.

[Gob99] Howard Gobioff. *Security for a High Performance Commodity Storage Subsystem.* Ph. d. dissertation, CMU–CS–99–160, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, July 1999. `http://www.pdl.cmu.edu/PDL-FTP/NASD/hbg_thesis.pdf`.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[Kar88] Paul A. Karger. *Improving Security and Performance for Capability Systems.* Ph. d. dissertation, Technical Report No. 149, Computer Laboratory, University of Cambridge, Cambridge, England, October 1988.

[KH84] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 2–12, Oakland, CA, USA, 29 April – 2 May 1984. IEEE Computer Society.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lam74] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974. Proceedings of the Fifth Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, NJ, USA, March 1971, pp. 437–443.

[Lan81] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[Lev83] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, MA, USA, 1983.

[Lip75] Steven B. Lipner. A comment on the confinement problem. *Operating Systems Review*, 9(5):192–196, November 1975. Proceedings of the Fifth Symposium on Operating Systems Principles, Unversity of Texas at Austin, Austin, TX, USA, 19–21 November 1975.

[Lob86] Jerome Lobel. *Foiling the System Breakers.* McGraw-Hill Book Company, New York, NY, USA, 1986.

[Mul85] Sape J. Mullender. *Principles of Distributed Operating System Design.* Ph. d. dissertation, Vrije Universiteit te Amsterdam, Amsterdam, The Netherlands, 31 October 1985. Published by Mathematisch Centrum, Amsterdam.

[NBF$^+$80] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. A provably secure operating system: The system, its applications, and proofs. Computer Science Laboratory Report CSL–116, SRI International, Menlo Park, CA, USA, May 7 1980.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In *ACM Conference on Computer and Communications Security (ACM-CCS'00)*, pages 245–254, Athens, Greece, November 2000. ACM.

[PW01] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, pages 184–201, Oakland, CA, May 2001. IEEE.

[SNI] SNIA - Storage Networking Industry Association. *OSD: Object Based Storage Devices Technical Work Group*. `http://www.snia.org/tech_activities/workgroups/osd/`.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988. `http://citeseer.ist.psu.edu/steiner88authentication.html`.

[Sol01] Frank G. Soltis. *Fortress Rochester : The Inside Story of the IBM Iseries*. 29th Street Press, Lewisville, TX, 2001.

[SRS$^+$00] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart cards. In *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes in Computer Science*, pages 17–36, Toulouse, France, 4–6 October 2000. Springer-Verlag.

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the eros confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, 14–17 May 2000.

[Web04] R. O. Weber. *SCSI Object-Based Storage Device Commands (OSD), Document Number: ANSI/INCITS 400-2004*. InterNational Committee for Information Technology Standards (formerly NCITS), December 2004. `http://www.t10.org/drafts.htm`.

[Wei69] Clark Weissman. Security controls in the ADEPT–50 time sharing system. In *AFIPS Conference Proceedings, Volume 35, 1969 Fall Joint Computer Conference*, pages 119–133, Montvale, NJ, USA, 1969. AFIPS Press.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, NY, USA, 1981.

[WOR$^+$74] K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, and D. G. Shumway. Primitive models for computer security. Technical Report ESD–TR–74–117, Case Western Reserve University, Cleveland, OH, HQ Electronic Systems Division, Hanscom AFB, MA, USA, 23 January 1974.

[Wra91] John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7, Oakland, CA, 20–22 May 1991.

[YG90] Che-Fn Yu and Virgil D. Gligor. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.