A Component Security Infrastructure

Yu David Liu Scott F. Smith
Department of Computer Science
The Johns Hopkins University
{yliu, scott}@cs.jhu.edu

June 25, 2002

Abstract

This paper defines a security infrastructure for access control at the component level of programming language design. Distributed components are an ideal place to define and enforce significant security policies, because components are large entities that often define the political boundaries of computation. Also, rather than building a security infrastructure from scratch, we build on a standard one, the SDSI/SPKI security architecture [EFL+99].

1 Introduction

Today, the most widely used security libraries are lower-level network protocols, such as SSL, or Kerberos. Applications then need to build their own security policies based on SSL or Kerberos. What this ends up doing in practice is making the policy narrow and inflexible, limiting the design space of the implementation and thus the final feature set implemented for users. A component-level policy using a security infrastructure, on the other hand, can both (1) abstract away from keys and encryption to concepts of principal, certificate, and authorization; and, (2) abstract away from packets and sockets to service invocations on distributed components. This is the appropriate level of abstraction for most application-level programming, and creation of such an architecture is our goal.

It is well known that components [Szy98] are useful units upon which to place security policies. But, while there has been quite a bit of recent research integrating security into programming languages, little of this has been at the component level. Industry has been making progress securing components. The CORBA Security Specification [OMG02] layers existing security concepts and protocols on top of CORBA. It is designed for interoperability between components using different existing protocols such as Kerberos, SSL, etc, and not using a higher-level security architecture. Microsoft's DCOM uses a very simple ACL-based security layer.

In this paper we define a new component-layer security model built on top of the SDSI/SPKI security infrastructure [RL96, EFL+99, CEE+01]. By building on top of an existing security infrastructure we achieve two important gains: open standards developed by cryptographers are more likely to be correct than a new architecture made by us; and, a component security model built on SDSI/SPKI will allow the component security policies to involve other non-component SDSI/SPKI principals. We have chosen SDSI/SPKI in particular because it is the simplest general infrastructure which has been proposed as an Internet standard. Other architectures which could also be used to secure components include the PolicyMaker/KeyNote trust management systems [BFK99], and the Query Certificate Manager (QCM) extension [GJ00]. The most widely used PKI today, X.509 [HFPS99], is too hierarchical to secure the peer-to-peer interactions that characterize components.

Although we have found no direct precedent for our approach, there is a significant body of related work. PLAN is a language for active networks which effectively uses the PolicyMaker/KeyNote security infrastructure [HK99]; PLAN is not component-based. Several projects have generalized the Java Applet model to support mobile untrusted code [GMPS97, BV01, HCC+98]. These projects focus on mobile code, however, and not on securing distributed component connections; and, they do not use existing security infrastructures.

Components we informally characterize a software component. Our characterization here differs somewhat from the standard one [Szy98] in that we focus on the behavior of distributed components at *run-time* since that is what is relevant to dynamic access control.

1. Components are named, addressable entities, running at a particular location (take this to be a machine and a process on that machine);
2. Components have services which can be invoked;
3. Components may be distributed, *i.e.* services can be invoked across the network;

The above properties hold of the most widespread component systems today, namely CORBA, DCOM, and JavaBeans.

1.1 Fundamental Principles of Component Security

We now define some principles we believe should be at the core of secure component architectures. We implement these principles in our model.

Principle 1.1 *Each Component should be a principal in the security infrastructure, with its own public/private key pair.*

This is the most fundamental principle. It allows access control decisions to be made directly between components, not via proxies such as users, etc (but, users and organizations should also be involved in access control policies).

Principle 1.2 *As with other principals, components are known to outsiders by their public key.*

Public keys also serve as universal component names, since all components should have unique public keys.

Principle 1.3 *Components each have their own secured namespace for addressing other components (and other principals).*

By localizing namespaces to the component level, a more peer-to-peer, robust architecture is obtained. Components may also serve public names to outsiders.

Principle 1.4 *Components may be private—if they are not registered in nameservers, then since their public keys are not guessable, they are hidden from outsiders.*

This principle introduces a capability-like layer in the architecture: a completely invisible component is a secured component. Capabilities are also being used in other programming-language-based security models [Mil, vDABW96].

1.2 The Cell Component Architecture

We are developing a security architecture for a particular distributed component language, *Cells* [RS02]. A brief definition of cells is as follows.

> *Cells* are deployable containers of objects and code. They expose typed linking interfaces (*connectors*) that may import (*plugin*) and export (*plugout*) classes and operations. Via these interfaces, cells may be dynamically linked and unlinked, locally or across the network. Standard client-server style interfaces (*services*) are also provided for local or remote invocations. Cells may be dynamically loaded, unloaded, copied, and moved.

So, cells have run-time services like DCOM and CORBA components, but they also have *connectors* that allow for persistent connections. Persistent connections can be made across the Internet. This is particularly good for security because a connection is secured at set-up, not upon every transaction. This is analogous to SSH: once a secure session is in place, it can stay up for hours or days and perform many transactions without additional authorization checks. See Figure 1 for an example of two `Chatter` cells which are connected across the Internet via a persistent `Chat` connection. They can communicate by the `send` and `receive` operations.

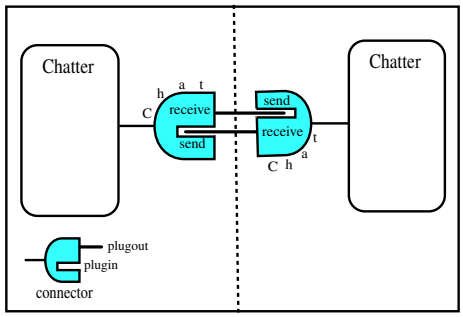

Figure 1: Two Chatter cells communicating over the network via Chat connector

There is a *president cell* in charge of cells at each location. The president can veto any access request and revoke any live connection at its location.

In this paper, we are not going to focus much on the cell-specific aspects, so the paper can be viewed as a proposal for a general component security architecture. The one exception is we do assume that each location has a single president cell which is in charge of all cells at that location, and even this assumption is not critical.

We use a simple, denotational model of cells here, which suffices for definition of the security model. We are currently implementing JCells, a modification of Java which incorporates the cell component architecture [Lu02].

1.3 The SDSI/SPKI Infrastructure

We very briefly review the SDSI/SPKI architecture, and give an overview of how we use it as a basis for the cell security architecture.

SDSI/SPKI is a peer-to-peer (P2P) architecture, a more flexible format than the centralized X.509 PKI [HFPS99]. Each SDSI/SPKI *principal* has a public/private key pair, and optional additional information such as its location and servers from which its certificate may be retrieved. *Certificates* are information signed by principals; forms of certificate include principals (giving the public key and optional information), group membership certificates (certifying a principal is a member of a certain group), and *authorization certificates* authorizing access. *Name servers* are

secure servers which map names (strings) to certificates. There is no *a priori* centralized structure to names—each server can have an arbitrary mapping. An *extended name* is a name which is a chain of name lookups; for instance asking one name server for "Joe's Sally" means to look up "Joe" on that name server, and then asking Joe's name server to look up "Sally". Access control decisions can be based either on direct lookup in an ACL, or via presentation of authorization certificates. In access control decisions using certificates, these certificates may be optionally delegated and revoked. See [RL96, EFL+99, CEE+01] for details.

We use SDSI/SPKI both for access control, and for a name service which allows cells to learn about each other securely. As mentioned above, each cell is registered as a SDSI/SPKI principal, and can issue certificates which fit the SDSI/SPKI format. Each cell can also serve SDSI/SPKI names. We currently don't generate certificates using the precise S-expression syntax of SDSI/SPKI, but the syntax is not an important issue (and, an XML format is probably preferred in any case). We incorporate SDSI/SPKI extended names, authorization certificates, delegation, and revocation models.

2 Cells

In this section we give a denotational model of cells and cell references (local or remote references to cells). Each cell has a public/private key and is thus a SDSI/SPKI principal. The public key not only serves for security, it serves to identify cells as no two cells should share a public key.

2.1 The Cell Virtual Machine (CVM)

The Cell Virtual Machine (CVM) is where cells run, in analogy to Java's JVM. The CVM is responsible for tasks including cell loading, unloading, serializing, deserializing and execution. Each CVM is represented by a particular cell, called its *president*. The president is a cell with extra service interfaces and connectors to implement the CVM's responsibilities. By reifying the CVM as a president cell, our architecture can be homogeneously composed of cells. It implicitly means CVM's are principals (identified by their president cells), which allows for CVM-wide security policies. In composing security policies, the president cell serves as a more abstract notion of location than the network-protocol-dependent locations used in existing security architectures such as Java's. This separation of low-level network protocols from the security architecture is particularly well-suited to mobile devices: even if network locations change from time to time, security policies can stay the same.

For the purposes of defining the security architecture, a simple denotational model of cells suffices: we ignore most of the run-time internals such as objects, classes and serialization, and focus on the access control policies for cells. We will define cells $c \in \mathbb{C}$ and cell references $cr \in \mathbb{CR}$. First we define the public and private keys of a cell.

2.2 Cell Identifiers CID and Locations LOC

All cells are uniquely identified by their *cell identifier*, *CID*.

Definition 2.1 *A cell identifier* $CID \in \mathbb{CID}$ *is the public key associated with a cell. A corresponding secret key* CID^{-1} *(*$\in \mathbb{CID}^{-1}$*) is also held by each cell.*

The *CID* is globally unique since it must serve as the identity of cells. All messages sent by a cell are signed by its CID^{-1}, and thus which will be verifiable given the *CID*.

When a cell is first loaded, its *CID* and CID^{-1} are automatically generated. Over the lifecycle of a cell, these values will not change regardless of any future unloading, loading, running, or migrating the cell may undergo. Since the *CID* is thus long-lived, it is sensible to make access control decisions based directly on cell identity.

Each loaded cell is running within a particular CVM. CVM's are located at network locations *LOC* which can be taken to be IP addresses.

2.3 Cell Denotations

We now define the denotation of a cell—each cell c is a structured collection of data including keys, certificates, etc. We don't define explicitly how cells evolve over time since that is not needed for the core security model definition.

Definition 2.2 *A cell* $\mathrm{c} \in \mathbb{C}$ *is defined as a tuple*

$$\mathrm{c} = \langle K, CertSTORE, SPT, NLT, BODY \rangle$$

where

- $K = \langle CID, CID^{-1} \rangle$ *are the cell's public and private keys.*
- $CertSTORE \in \mathbb{CERTSET}$ *is the set of certificates held by the cell for purposes of access and delegation.* $\mathbb{CERTSET}$ *is defined in Section 5.2 below.*
- $SPT \in \mathbb{SPT}$ *is the security policy table, defined in Section 5 below.*
- $NLT \in \mathbb{NLT}$ *is the naming lookup table, defined in Section 4 below.*
- $BODY$ *is the code body and internal state of the cell, including class definitions, service interfaces, connectors, and its objects. We treat this as abstract in this presentation.*

A cell environment $cenvt \in \mathbb{CENVT}$ is a snapshot of the state of all active cells in the universe: $\mathbb{CENVT} = \{C \subseteq \mathbb{C} \mid C \text{ finite and for any } c_1, c_2 \in C, \text{ their keys differ}\}$.

2.4 Cell References

Cells hold *cell references*, cr, to other cells they wish to interact with. Structurally, a cell reference corresponds to a SDSI/SPKI principal certificate, including the cell's *CID* and possible location information. This alignment of SDSI/SPKI principal certificates with cell references is an important and fundamental aspect of our use of the SDSI/SPKI architecture. In an implementation, cell references will likely also contain cached information for fast direct access to cells, but we ignore that aspect here.

Definition 2.3 *A cell reference* $cr \in \mathbb{CR}$ *is defined as a tuple*

$$cr = \langle CID_{cell}, CID_{host}, LOC_{host} \rangle$$

where CID_{cell} *is the CID of the referenced cell;* CID_{host} *is the CID of the CVM president cell where the referenced cell is located;* LOC_{host} *is the physical location of the CVM where the referenced cell is located. If* cr *refers to a CVM president cell,* $CID_{cell} = CID_{host}$.

All interactions with cells by outsiders are through cell references; the information in cr can serve as a universal resource locator for cells since it includes the physical location and CVM in which the cell is running.

Definition 2.4 $REF2CELL$ *is defined as follows: If* $cenvt \in \mathbb{CENVT}$ *and* $cr \in \mathbb{CR}$, $REF2CELL(cenvt, cr)$ *returns the cell* c *which* cr *refers to.*

3 An Example

In this section we informally introduce our security architecture with an example modeled on a B2B (Business-to-Business) system such as a bookstore that trades online with multiple business partners.

In our example, there is a "Chief Trader" business that communicates with its trading partners Business A, Business B, and Business M, all of which are encapsulated as cells. Also, Business A has a partner Business K which is not a partner of the Chief Trader. Fig. 2 shows the layout of the cells and the CVM president cells that hold them. *CID*'s are large numbers and for conciseness here we abbreviate them as e.g. $4\ldots9$, hiding the middle digits. The source code for the Chief Trader has the following JCells code, which is more or less straightforward.

```
cell ChiefTrader {
  service IQuery {
    double getQuote(int No, String cate);
    List search(String condition);
  }
  connector ITrade {
    plugouts{
      boolean makeTrans(TradeInfo ti);
    }
    plugins{
      EndorseClass getEndorse();
    }
  }
 /* ... cell body here ... */
}
```

Name	CID	CVM	LOC
PtnrB	3...1	8...1	bbb.biz
CVM1	7...5	7...5	aaa.biz

Table 1a

Name	Extended Name
PtnrA	[CVM1,A]

Table 1b

Name	Group Members
PtnrGrp	{CR(CID=3...1, CVM=8...1, LOC=bbb.biz), PtnrA, [CVM1,M]}

Table 1c

Table 1: Naming Lookup Table for Chief Trader Cell

Name	CID	CVM	LOC
PtnrK	4...1	9...7	kkk.biz
ChiefTrader	1...3	6...5	chieftrader.biz

Table 2: Naming Lookup Table for Business A Cell

The major functionalities of Chief Trader are shown above. In service interface `IQuery`, `getQuote` takes the merchandise number and the category (promotion/adult's/children's book) and returns the quote; `search` accepts an SQL statement and returns all merchandise satisfying the SQL query. The Chief Trader cell also has an `ITrade` connector, which fulfills transactions with its business partners. Inside it, `getEndorse` is a plugin operation that needs to be implemented by the invoking cell, which endorses transactions to ensure non-repudiation.

Our infrastructure has the following features.

Non-Universal Names Cells can be universally identified by their *CID*'s. However, other forms of name are necessary to facilitate name sharing, including sharing by linked name spaces, described below. Instead of assuming the existence of global name servers, each cell contains a lightweight *Naming Lookup Table* (*NLT*), interpreting local names it cares about, and only those it cares about; example *NLT*'s are given in Tables 1, 2, and 3. An *NLT* entry maps a local name into one of three forms of value:a cell reference, extended name, or group. So, an *NLT* is three sub-tables for each sort; the three sub-tables for the Chief Trader are given in Figures 1a, 1b, and 1c, respectively. In Figure 1a, a local name is mapped to a cell reference $cr = \langle CID_{cell}, CID_{host}, LOC_{host} \rangle$: the cell with *CID* `3...1` running on a CVM on `bbb.biz` with president cell CID `8...1` is named `PtnrB` in the Chief Trader's namespace. In Figure 1b, a map from local names to *extended names* is defined, serving as an abbreviation. In Figure 1c a group `PtnrGrp` is defined. We now describe the meaning of these extended name and group name entries.

Extended Names Although names are local, namespaces of multiple cells can be linked together as a pow-

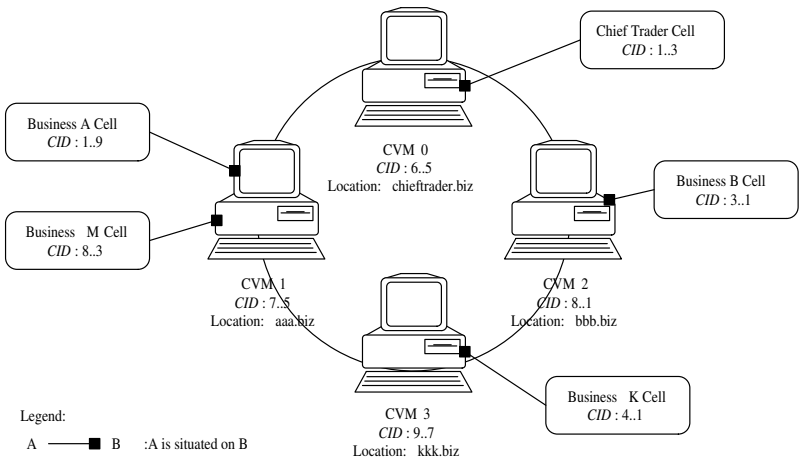

Figure 2: Example: A B2B System Across the Internet

Name	CID	CVM	LOC
A	1...9	thiscid	localhost
M	8...3	thiscid	localhost
Chief	1...3	6...5	chieftrader.biz

Table 3: Naming Lookup Table for CVM 1 President Cell

erful scheme to refer to other cells across cell boundaries; we just ran into an example, [CVM1, A] in Table 1c. This refers to a cell named A in the namespace of the cell named CVM1, from the perspective of the namespace of the Chief Trader cell itself. Extended names make it possible for a cell to interact with other cells whose *CID* and location information is not directly known: the Chief Trader cell does not directly keep track of the *CID* and location of Business A, but still has a name for it as PtnrA and the two can communicate. Extended names also help keep the size of naming information on any individual cell small.

Cell Groups Our system supports the definition of cell groups. The ability to define groups greatly eases the feasibility of defining security policies: policies can be group-based and need not be updated for every single cell. Table 1c defines a group PtnrGrp with three members. Group members can be direct cell references (CR(CID=3...1, CVM=8...1, LOC=bbb.biz)), local names (PtnrA), extended names ([CVM1,M]), or other sub-groups. This flexibility allows appropriate groups to be easily defined.

Access Control The security policy defines how resources should be protected. The units of cell protection include services, connectors and operations. Each cell contains a *security policy table* which specifies the precise access control policy. Table 4 gives the Chief Trader's security policy table. Here, operation search of its IQuery service can be invoked by any member in group PtnrGrp; the second entry in the table indicates that PtnrA can invoke any operation of its IQuery service; the third entry indicates PtnrA can also connect to its ITrade connector.

Hooks Security decisions can also be made contingent on the parameters passed in to an operation. This mechanism is called a *hook*, and provides access control at a finer level of granularity. The fourth entry of Table 4 has a hook h1 attached, which declares that the getQuote operation of IQuery service in the Chief Trader cell can be invoked by PtnrB only if the second parameter provided is equal to promotion, meaning only promotion merchandise can be quoted by PtnrB.

Delegation Delegation solves the problem of how two cells unaware of each other can interact effectively. For instance, Business K is not a partner of the Chief Trader, but since it is a partner of Business A (a partner of the Chief Trader), it is reasonable for Business K to conduct business with the Chief Trader. According to Table 4, Business A (named PtnrA in the name space of the Chief Trader cell) can have access to the IQuery interface of the Chief Trader, but business K can not. However, the Chief Trader has left the delegation bit (Delbit) of that entry to 1, which means any partner with this access right can delegate it to others. Table 5 shows Business A's security policy table; it defines a delegation policy whereby it grants PtnrK a certificate for the IQuery service of the Chief Trader cell. Thus cell K can invoke the IQuery service using this delegation certificate, even though the Chief Trader cell does not directly know Business K.

Subject	Resource	Accessright	Hook	Delbit
PtnrGrp	⟨thiscell, IQuery.search⟩	invoke	NULL	0
PtnrA	⟨thiscell, IQuery⟩	invoke	NULL	1
PtnrA	⟨thiscell, ITrade⟩	connect	NULL	0
PtnrB	⟨thiscell, IQuery.getQuote⟩	invoke	h1**	0

** h1(arg1, arg2) = {arg2 = "promotion"}

Table 4: Security Policy Table for Chief Trader

Subject	Resource	Accessright	Hook	Delbit
PtnrK	⟨ChiefTrader, IQuery⟩	invoke	NULL	0

Table 5: Security Policy Table for Business A

4 Name Services

Name services are needed to find cells for which a cell reference cr is lacking. The example of Section 3 gave several examples of name service in action.

4.1 Names and Groups

The Cell naming scheme uses the decentralized extended naming scheme of SPKI/SDSI [CEE+01]. In contrast with global naming schemes such as DNS and X.509 where global name servers are assumed, decentralized naming more reflects the nature of the Internet, for which Cells are designed. Each and every cell can serve as an independent, light-weight SDSI/SPKI naming server, so name service is pervasively distributed.

The set of local names are strings $n \in \mathbb{N}$. Local names map to cell references, cell extended names, and cell groups.

The extended name mechanism, illustrated in the example, enables a cell to refer to other cells through a chain of cross-cell name lookups.

Definition 4.1 *An extended name $n_{ext} (\in \mathbb{N}_{EXT})$ is a sequence of local names $[n_1, n_2, \ldots, n_s]$.*

Each n_{i+1} is a local name defined in the name space of the cell n_i.

SDSI/SPKI groups are represented as a name binding which maps the name of the group to the set of group members. Group members are themselves local names, which can in turn be mapped to cells or sub-groups. The owner of the group is the cell holding the local name.

Definition 4.2 *The set of groups $\mathbb{G}$ is defined as $\mathbb{G} = POWER(\mathbb{CR} \cup \mathbb{N}_{EXT})$.*

4.2 The Naming Lookup Table and Naming Interface

Local name bindings are stored in a *naming lookup table, NLT*. Every cell (including presidents) holds such a table, defining how local names relevant to it are mapped to cells and groups.

Definition 4.3 *Given a space $\mathbb{V} = \mathbb{CR} \cup \mathbb{G} \cup \mathbb{N}_{EXT}$ of values,*

1. *each naming lookup entry $NLE \in \mathbb{NLE}$ is a tuple $NLE = \langle n, v \rangle$ for $n \in \mathbb{N}$ and $v \in \mathbb{V}$;*
2. *a naming lookup table $NLT \in \mathbb{NLT}$ is a set of naming lookup entries: $NLT \subseteq \mathbb{NLE}$ such that for $\langle n_1, v_1 \rangle, \langle n_2, v_2 \rangle \in NLT$, $n_1 \neq n_2$.*

An example illuminating this definition can be found in Table 1 of Section 3. $GET\_VALUE(NLT, n)$ is a partial function which looks up the value of n in table *NLT*, and is undefined for names n with no mapping.

In the JCells language, the naming lookup table is maintained by an interface `INaming` present on all cells, which contains the operations for both looking up and modifying name lookup table information. Data in the naming lookup table can only be located or modified via `INaming`. The most important operation in `INaming` is `lookup`, which we now specify. Group membership is another important operation which we leave out of this short version.

4.3 Name Lookup

The `lookup`(*cenvt*, n_{ext}, cr_{srt}) operation, defined in Fig. 3, looks up an extended name n_{ext}, starting from the naming lookup table of the current cell which has reference cr_{srt}, and returns the final cell reference that n_{ext} refers to. Since arbitrary cell name servers could be involved in extended name lookup, the operation is parameterized by the global cell environment *cenvt*. A simple local name is a special case of an extended name with *LENGTH*(n_{ext}) = 1.

The first case in `lookup1` is when the name is not an extended name: the value v in the *NLT* is directly returned. The second case is for an extended name, and the next element of the extended name must be looked up using cell v's nameserver. The last case is where the value is an extended name itself.

The above algorithm does not define how the computation is distributed; to be clear, given an invocation `lookup` on a cell, the only parts involving distributed interaction are $REF2CELL$ and GET_VALUE, which in

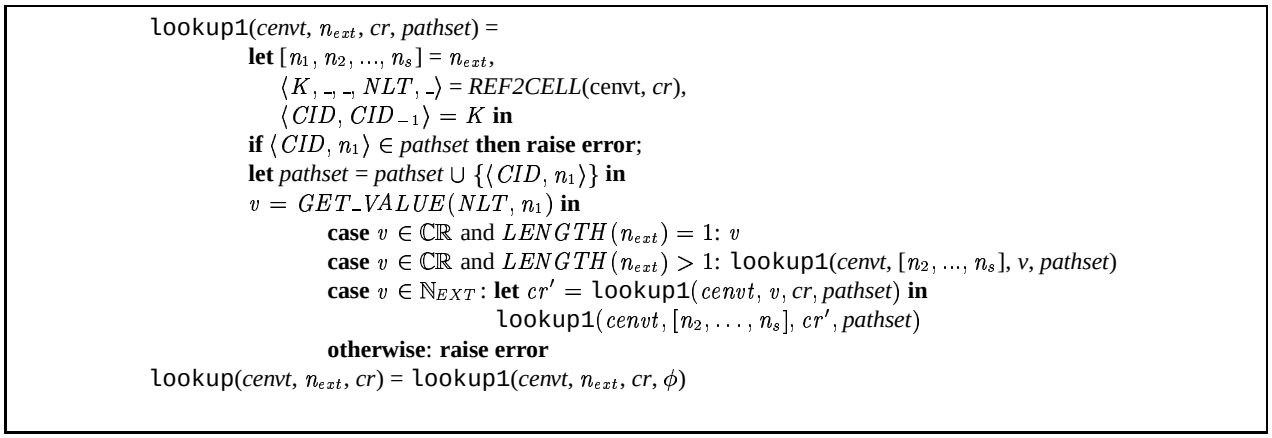

Figure 3: Definition of *lookup*

combination locate another cell, look up a name from it and immediately return the value, be it a cell reference or extended name, to the initial invoker; all recursive calls to `lookup1` are local.

Compared with SDSI/SPKI [CEE+01], our algorithm is more general since it allows arbitary expansion of local names (the third case in the algorithm). For example, "Sallie's Joe's Pete" could in the process of resolution return an extended name for "Joe" on Sally as "Sam's Joe", which then must be resolved before "Pete" can be looked up on it; [CEE+01] will not resolve such names. However, this more general form is expressive enough that cyclic names are possible (e.g. "Sallie's Joe" maps to the extended name "Sallie's Joe's Fred") and a simple cycle detection algorithm must be used to avoid infinite computation. The cycle detection algorithm used in Fig. 3 is as follows. The `lookup` function maintains a set *pathset* of each recursive `lookup` name this initial request induces; if the same naming lookup entry is requested twice, a cycle is flagged and the algorithm aborted. A cycle can only be induced in the third case (local name expansion), where the algorithm divides the lookup process into two subtrees. Since there are only finitely many expansion entries possible, the algorithm always terminates.

5 Access Control

Access control decisions in the cell security architecture are based on the SDSI/SPKI model, specialized to the particular resources provided by cells. Each cell has associated with it a *security policy table*, which declares what *subjects* have access to what *resources* of the cell. This is an "object-based" use of SDSI/SPKI—every cell is responsible for controlling access to its resources.

Definition 5.1 *1. The set of subjects $\mathbb{S} = \mathbb{N}_{EXT} \cup \{ALL\}$: Subjects are extended names for cells or groups, or ALL which denotes any subject.*

2. The set of resources $\mathbb{R} = \langle \mathbb{O}, \mathbb{U}\rangle$, where

- *$\mathbb{O} = \mathbb{N}_{EXT} \cup \{$`thiscell`$\}$ is the set of resource owners, with `thiscell` denoting the cell holding the security policy itself.*
- *$\mathbb{U} = \mathbb{SRV} \cup \mathbb{CNT} \cup \mathbb{OP}$ is the set of protection units, which can be a cell service interface, connector, or operation, respectively.*

A partial order $\preccurlyeq_u$ is defined on protection units: $u_1 \preccurlyeq_u u_2$ if and only if u_1 is subsumed by u_2; details are omitted from this short version.

Access rights are $\mathbb{A} = \{$`invoke`, `connect`$\}$, where if $u \in \mathbb{CNT}$, a will be `connect`, and if $u \in \mathbb{SRV} \cup \mathbb{OP}$, a will be `invoke`.

Definition 5.2 *A security policy entry $spe \in \mathbb{SPE}$ is a tuple $spe = \langle s, r, a, h, d\rangle$, meaning access right a to resource r can be granted to subject s, if it passes the security hook h. This access right can be further delegated if d is set to 1. Specifically, we require*

- *$s \in \mathbb{S}$, $r \in \mathbb{R}$, $a \in \mathbb{A}$.*
- *$h \in \mathbb{H}$ is an optional security hook, a predicate which may perform arbitrary checking before access rights are granted (details are in Section 5.1 below). It is ϵ unless $r = \langle o, u\rangle$, $o =$ `thiscell` and $u \in \mathbb{OP}$. The set of security hooks that is associated with operation $op \in \mathbb{OP}$ is denoted $\mathbb{H}_{op}$.*
- *$d \in \mathbb{D} = \{0, 1\}$ is the delegation bit as per the SDSI/SPKI architecture. It defines how the access rights can be further delegated, detailed in Section 5.2 below.*

The Security policy table is then a set of security policies held by a cell: $SPT \in \mathbb{SPT} = POWER(\mathbb{SPE})$.

5.1 Security Hooks

The access control model above restricts access rights to cell connectors and service interfaces based on requesting principals. However, more expressiveness is needed in some situations. One obvious example is local name entries: How can one protect a single local name entry, *i.e.*, one particular invocation of `lookup`? Currently, we have a naming interface defined (see Section 4.2), but completely protecting operation `lookup` is too coarse-grained. For these cases we need a parameterized security policy, in which the policy is based on the particular arguments passed to an operation such as `lookup`. Security hooks are designed to fill this need. Hooks can be implemented in practice either as JCells code or written in some simple specification language; here we abstractly view them as predicates. The set of security hooks $\mathbb{H}_{op}$ contains verifying predicates that are being checked when the associated operation *op* is triggered.

Definition 5.3 *Given security policy entry* $spe = \langle s, \langle o, op\rangle, a, h_{op}, d\rangle$*, a hook* $h_{op} \in \mathbb{H}_{op}$ *is a predicate*

$$h_{op}(v_1, v_2, ...v_m)$$

where $v_1, v_2, ...v_m$ *are operation op parameters, and each* $v_i \in \mathbb{VAL}$*, for* $\mathbb{VAL}$ *an abstract set of values which includes cell references cr along with integers and strings.*

Access control security hooks are checked right before invocation of the associated operation, and the invocation can happen only if the security hook returns true.

5.2 Delegation

We use the SPKI/SDSI delegation model to support delegation in cell access control. A subject wishing access to a cell's resource can present a collection of certificates authorizing access to that cell. And, revocation certificates can nullify authorization certificates.

Definition 5.4 *An authorization certificate* $AuthC \in \mathbb{AUTHC}$ *is a signed tuple*

$$AuthC = \langle CID_I, CID_D, CID_R, u, a, d\rangle$$

where CID_I *is the CID of certificate issuer cell;* CID_D *is the CID of the cell being delegated;* CID_R *is the CID of resource owner cell;* u *is the resource unit;* a *is the access right to be delegated;* d *is the delegation bit: if 1, the cell specified by* CID_D *can further delegate the authorization to other cells.*

Definition 5.5 *A revocation certificate* $RevoC \in \mathbb{REVOC}$ *is a signed tuple*

$$RevoC = \langle CID_I, CID_D, CID_R, u, a\rangle$$

In this definition, CID_I is the CID of revocation certificate issuer cell; CID_D is the cell which earlier received an authorization certificate from the issuer cell but whose certificate is now going to be revoked; CID_R is the CID of resource owner cell; u is the resource unit; and, a is the access right to be revoked. The set of certificates is defined as $\mathbb{CERTSET} = POWER(\mathbb{AUTHC} \cup \mathbb{REVOC})$.

Support for delegation is reflected in the definition of a security policy entry, SPE: a delegation bit d is included. This bit is set to permit the authorized requester cell to further delegate the access rights to other cells. Cells can define delegation security policies by composing security policy entries for resources they do not own. This kind of entry takes the form $\langle s, \langle o, u\rangle, a, \texttt{NULL}, d\rangle$, with $o \neq$ `thiscell`. This denotes that an $AuthC$ granting access a to u of o can be issued if the requester is s. Notice security hooks are meaningless in this case.

The delegation proceeds as follows: suppose that on the resource cell side there is a security policy entry $spe = \langle s, \langle \texttt{thiscell}, u\rangle, a, h, 1\rangle$; cell s will be granted an $AuthC$ if it requests access a to unit u on the resource cell. Cells holding $AuthC$ can define their own security policies on how to further delegate the access rights to a third party, issuing another $AuthC$, together with the certificate passed from its own delegating source. This propagation can iterate. Cells automatically accumulate "their" pile of authorization certificates from such requests, in their own $CertSTORE$. When finally access to the resource is attempted, the requestor presents a chain of $AuthC$ which the resource cell will check to determine if the access right should be granted. Existence of a certificate chain is defined by predicate $EXISTS\_CERTCHAIN$, see Fig. 4. In the definition, operator *COUNTERPART* maps authorization certificates to their corresponding revocation certificate, and vice-versa. $\mathbf{B}_3$ in Figure 4 checks revocation certificates: if some cell revokes the *AuthC* it issued earlier, it sends a corresponding *RevoC* to the resource owner. When any cell makes a request to the resource and presents a series of *AuthC*, the resource holder will also check if any *RevoC* matches the *AuthC*.

5.3 `isPermitted`: The Access Control Decision

Each cell in JCells has a built-in security interface `ISecurity`, which contains a series of security-sensitive operations. The most important is `isPermitted` (see Fig. 5), which checks if access right $a_{req}(\in \mathbb{A})$ to resource unit $u_{req}(\in \mathbb{U})$ can be granted to subject $cr_{req}(\in \mathbb{CR})$. If $u_{req} \in \mathbb{OP}$, a list of arguments are provided by $arglist_{req}(\in \mathbb{ARGLIST})$ for possible checking by a hook. A set of authorization certificates $CertSet_{req}(\in \mathbb{CERTSET})$ may also be provided. The cell performing the check is $cr_{chk}(\in \mathbb{CR})$. The environment for all active cells is $cenvt(\in \mathbb{CENVT})$.

$EXISTS\_CERTCHAIN(cenvt, cr_{chk}, cr_{req}, u_{req}, a_{req}, CertSet_{req})$=

let $(CID_{chk}, t_1, t_2) = cr_{chk}$ **in**
let $(CID_{req}, t_1, t_2) = cr_{req}$ **in**
let $\langle t_1, CertSTORE_{chk}, t_2, t_3, t_4\rangle = REF2CELL(cenvt, cr_{chk})$ **in**
case $\mathbf{B}_1$ and $\mathbf{B}_2$ and $\mathbf{B}_3$: $\langle CID_{k_1}, u_{k_1}, a_{req}\rangle$
otherwise ϵ

where
$\mathbf{B}_1 = \exists Auth_1, ...Auth_n \in CertSet_{req}$ with

$$\begin{aligned} Auth_1 &= \langle CID_{chk}, CID_{k_1}, \langle CID_{chk}, u_{k_1}\rangle, a_{req}, 1\rangle \\ Auth_2 &= \langle CID_{k_1}, CID_{k_2}, \langle CID_{chk}, u_{k_2}\rangle, a_{req}, 1\rangle \\ &\vdots \\ Auth_n &= \langle CID_{k_{n-1}}, CID_{req}, \langle CID_{chk}, u_{k_n}\rangle, a_{req}, d\rangle \qquad (d \in \{0,1\}) \end{aligned}$$

$\mathbf{B}_2 = u_{k_n} \preccurlyeq_u u_{k_{n-1}} \ldots \preccurlyeq_u u_{k_1}$
$\mathbf{B}_3 = \forall auth \in \{Auth_1, ..., Auth_n\}, COUNTERPART(auth) \notin CertSTORE_{chk}$

Figure 4: Definition of *EXISTS_CERTCHAIN*

`isPermitted`$(cenvt, cr_{req}, u_{req}, a_{req}, arglist_{req}, CertSet_{req}, cr_{chk}) =$
let $(CID_{req}, t_1, t_2) = cr_{req}$ **in**
let $\langle t_1, t_2, SPT_{chk}, t_3, t_4\rangle = REF2CELL(cenvt, cr_{chk})$ **in**
$(\exists\langle s, \langle o, u\rangle, a, h, d\rangle \in SPT_{chk}.$ $\mathbf{B}_1$ **and** $\mathbf{B}_2$ **and** $\mathbf{B}_3$ **and** $\mathbf{B}_4)$
or
let $t = EXISTS\_CERTCHAIN(cenvt, cr_{chk}, cr_{req}, u_{req}, a_{req}, CertSet_{req})$ **in**
let $\langle CID_{req}, u_{req}, a_{req}\rangle = t$ for $t \neq \epsilon$ **in**
$(\exists\langle s, \langle o, u\rangle, a, h, d\rangle \in SPT_{chk}.$ $\mathbf{B}_1$ **and** $\mathbf{B}_2$ **and** $\mathbf{B}_3$ **and** $\mathbf{B}_4)$

where
$\mathbf{B}_1 = (a = a_{req})$
$\mathbf{B}_2 = (o =$ `thiscell`$)$ **and** $(u_{req} \preccurlyeq_u u)$
$\mathbf{B}_3 =$
case $ISGROUP(cenvt, s)$: `isMember`$(cenvt, cr_{req}, s, cr_{chk})$
case $ISCELL(cenvt, s)$:
let $\langle CID_s, CID_{host_s}, LOC_{host_s}\rangle =$ `lookup`$(cenvt, s, cr_{chk})$ **in** $(CID_s = CID_{req})$
$\mathbf{B}_4 = h(arglist_{req})$ **where** $u_{req} \in \mathbb{OP}$

Figure 5: Definition of *isPermitted*

`isPermitted` grants access either if there is a direct entry in the security policy table granting access, or if proper authorization certificates are presented. The first **or** case checks if there is directly an entry in the security policy table granting access. If authorization certificates are provided together with the request (second **or** case), permission will be granted if these certificates form a valid delegation chain, and the first certificate of the chain can be verified to be in the security policy table. $\mathbf{B}_1$ matches the access right; $\mathbf{B}_2$ matches the resource; $\mathbf{B}_3$ matches the subjects, which is complicated by the case a subject is a group; and $\mathbf{B}_4$ checks the security hook if any.

6 Conclusions

In this paper we have shown how the SDSI/SPKI infrastructure can be elegantly grafted onto a component architecture to give a general component security architecture. Particularly satisfying is how components can serve as principals, and how SDSI/SPKI naming gives a secure component naming system. We believe this infrastructure represents a good compromise between simplicity and expressivity. Very simple architectures which have no explicit access control or naming structures built-in lack the ability to express policies directly and so applications would need to create their own policies. More complex architectures such as trust management systems [BFK99] are difficult for everyday programmers to understand and thus may lead to more security holes.

Beyond our idea to use SDSI/SPKI for a peer-to-peer component security infrastructure, this paper makes several other contributions. We define four principles of component security, including the principle that components themselves should be principals. An implementation aspect developed here is the *cell reference*: the public key

plus location information is the necessary and sufficient data to interact with a cell. This notion combines programming language implementation needs with security needs: the cell needs to be accessed, and information supposedly from it needs to be authenticated. Modelling each CVM with a president cell simplifies the definition of per-site security policies. It also separates low-level location information from security polices, a structure well-suited to mobile devices. We define a name lookup algorithm which is more complete than the one given in [CEE+01]—extended names can themselves contain extended names, and all names can thus be treated uniformly in our architecture. Our architecture for name service is more pervasive than the distributed architecture proposed in SDSI/SPKI—every cell has its own local names and can automatically serve names to others. So while we don't claim any particularly deep results in this paper, we believe the proposal represents a simple, elegant approach that will work well in practice.

Many features are left out of this brief description. SDSI/SPKI principals that are not cells should be able to interoperate with cell principals. Several features of SDSI/SPKI and of cells are not modeled. We have not given many details on how data sent across the network is signed and encrypted.

The case of migrating cells is difficult and largely skipped in the paper; currently cells migrate with their private key, and a malicious host can co-opt such a cell. There should never simultaneously be two cells with the same *CID*, but since the system is open and distributed it could arise in practice. By making *CID*'s significantly long and being careful in random number generation, the odds of accidentally generating the same *CID* approach zero; more problematic is when a *CID* is explicitly reused, either by accident or through malicious intent. In this case a protocol is needed to recognize and resolve this conflict, a subject of future work.

References

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Security Protocols—6th International Workshop*, volume 1550 of *Lecture Notes in Computer Science*, pages 59–66. Springer-Verlag, 1999.

[BV01] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 4:359–384, 2001.

[CEE+01] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, pages 285–322, 2001.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. Internet Engineering Task Force RFC2693, September 1999. `ftp://ftp.isi.edu/in-notes/rfc2693.txt`.

[GJ00] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997.

[HCC+98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. RFC 2459: Internet X.509 public key infrastructure certificate and CRL profile, January 1999. `ftp://ftp.internic.net/rfc/rfc2459.txt`.

[HK99] Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In *Proceedings of the First International Working Conference on Active Networks (IWAN '99)*, volume 1653, pages 307–314. Springer-Verlag, 1999.

[Lu02] Xiaoqi Lu. Report on the cell prototype project. (Internal Report), March 2002.

[Mil] Mark Miller. The E programming language. `http://www.erights.org`.

[OMG02] OMG. Corba security service specification, v1.8. Technical report, Object Management Group, March 2002. `http://www.omg.org/technology/documents/formal/security_service.htm`.

[RL96] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure, 1996. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[RS02] Ran Rinat and Scott Smith. Modular internet programming with cells. In *ECOOP 2002*, Lecture Notes in Computer Science. Springer Verlag, 2002. `http://www.cs.jhu.edu/hog/cells`.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[vDABW96] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.