Modeling Security Properties of Service Proxy Toolkits

Carolyn Talcott
Computer Science Laboratory, SRI International, Menlo Park, CA 94025 USA

July 26, 2004

Abstract

1 Introduction

Consider a distributed system in which servers wish to make services available and client applications need to find and access services (remotely located). The objective of a service proxy toolkit (SPTK) is to facilitate client - server interaction by providing

(1) mechanisms for service registration and lookup; and

(2) proxies that make the communication appear local to both the client and the server processes.

In addition to facilitating discovery and access, an SPTK should be able to transparently support a variety of security properties: protecting information communicated between client and server, assuring client and server of each others identity, and enforcing access policies.

The Secure Service Proxy Toolkit (SSPTK), based on Java RMI and Jini technologies, was developed by John Mitchell, Ninghui Li, and Derrick Tong as part of the Stanford-SRI Dynamic Coalitions project *Agile Management of Dynamic Collaboration*. The objective of the work reported herein was to formally model and analyze the SSPTK using the Maude tool (`http://maude.cs.uiuc.edu`) which supports development and analysis of executable specifications based on rewriting logic [3] An additional objective was to use this case study as an example of how to develop and organize models of such systems and to analyze them in terms of possible attacks. To this end we describe the model and analyses in some detail after first giving a high-level summary. The code and analyses are available at `www-formal.stanford.edu/clt/FTN/SPTK/index.html`.

2 SPTK architecture and SSPTK design

A basic requirement for an SPTK is to provide the desired level of security protection transparently. Thus it should provide fixed interfaces to client and server applications, only the internal interactions change in order to provide protection against given threat conditions.

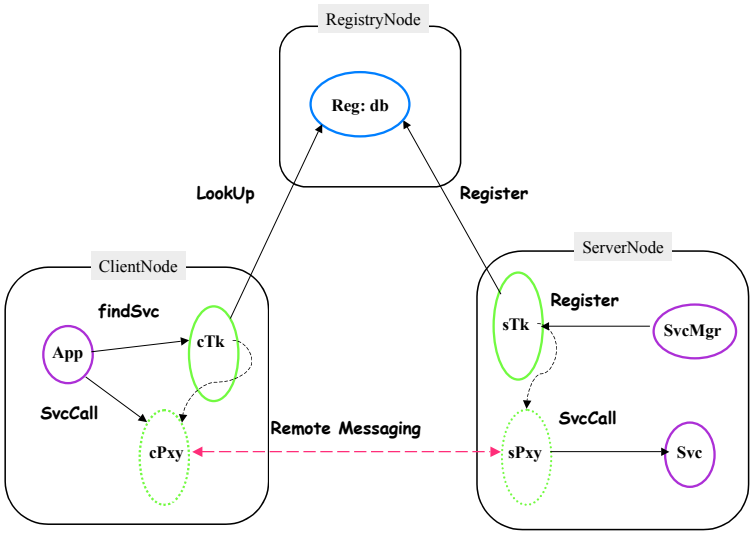

Figure 1: SPTK architecture

Figure 1 shows the client-server level architecture of an SPTK. The main components are: client applications (App), services (Svc), service managers (SvcMgr), client and server toolkit components (cTk, cPxy, sTk, sPxy), and a registry (Reg). The registry provides Register and Lookup interfaces used by the toolkit components sTk and cTk respectively. In turn these components provide Register and discovery (findSvc) interfaces to service managers (SvcMgr) and client applications (App) respectively. The interface provided by a service to a client is abstracted in the SvcCall interface which is also supported by the proxies on the client (cPxy) and server (sPxy) nodes. The dashed arrows indicate dynamic creation of proxies.

We assume agreements are in place that associate service behavior specifications with descriptions that can be used to register and lookup services. This is part of what it means for a client to trust a server. Then, intuitively, the desired behavior from the client point of view is the following. If a service Svc with description D is registered by server S (i.e. by its service manager, SvcMgr), and if client application App requests a service satisfying description D and receives a proxy handle cPxy, then interactions of App with the service using cPxy proceed according to the behavior specified by D. Furthermore, a clients private data should not be made available to other parties due to interactions with the service. Meeting this requirement has two aspects: firstly the server must implement the described service, and secondly the proxy received by the client should be a proxy for the requested service, that also provides security for the client and the server. It is the latter aspect that an SPTK is concerned with.

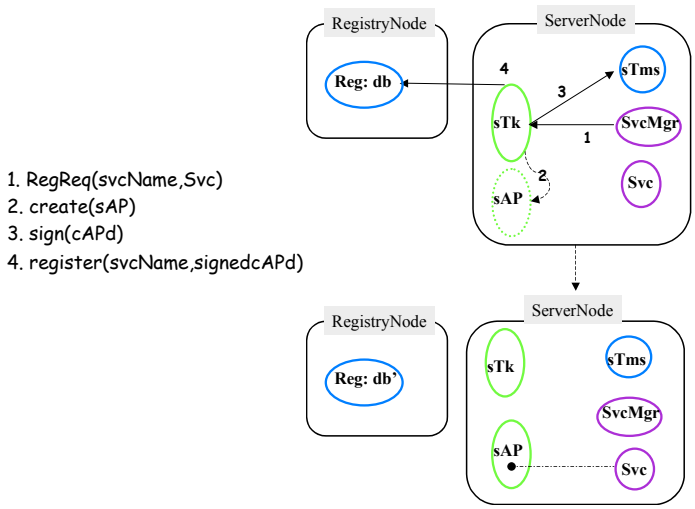

Figure 2: SSPTK Registration Scenario

The design of the SSPTK is motivated by the following security goals:

I Proxy access to client JVM and resources is controlled—this can be addressed by Java security mechanisms.

II Communications between proxies and services are secure—this can be addressed by proxies that use secure remote communication such as SSL, but can the client rely on the proxy it receives being such a proxy?

III Clients should be able to authenticate proxies—both code and data.

IV Some applications require services to authenticate and authorize client access, supporting

- authentication only once in a session (single sign on)
- multiple authentication mechanisms
- authorization based on arguments of calls

Figures 2, 3, 4, 5 depict the architecture of the SSPTK and its operation. It refines the Java RMI/Jini based remote service architecture in several ways. The service proxy is signed using the servers private key before being registered, and this signature is checked by the client before accepting the proxy. Also, there are two proxy stages: the first stage proxies carry out client authentication, and the second stage proxies implement a secure authenticated session that provides secure communication and access control.

Figure 2 shows a service registration scenario. To register a service, the server side toolkit creates client and server authentication proxies. The server proxy knows how to access (has

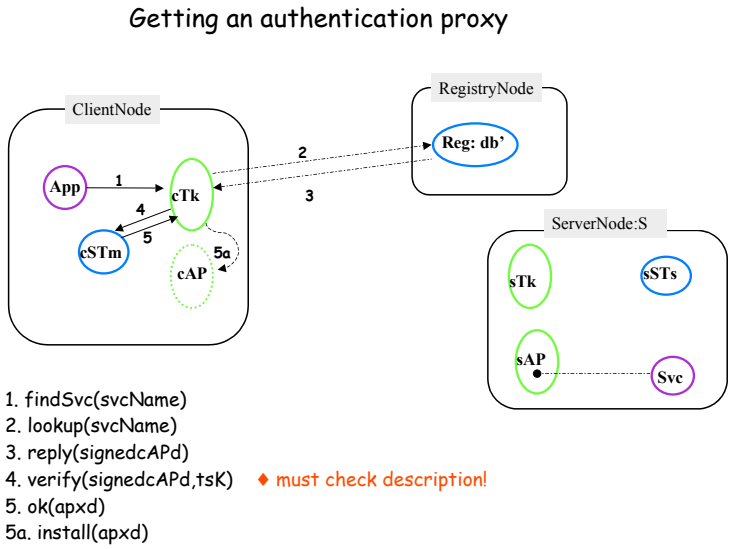

Figure 3: SSPTK Service Lookup and Verification

a reference to) the service and is waiting for authentication requests from installations of the client. The client authentication proxy has a handle for the server authentication proxy and also includes the service description. It is serialized, signed and registered.

Figure 3 shows the process of obtaining a registered service proxy and verifying that it is signed by a trusted server and registered for the desired service. Once verified the registered signed proxy—the client-side authentication proxy—is installed (loaded).

Note that without the check for the desired service an attacker could cause a proxy for the wrong service to be accepted, if a trusted server provides more than one service. The need for this additional check was discovered during the process of developing the formal models.

Figure 4 shows interaction of the authentication proxies to set up a secure authenticated session. First a secure connection is established, then the client proxy submits the clients credential to the server proxy. This is checked, the clients permissions are determined, and the server proxy installs a server-side secure session proxy to listen for service requests and provide the access control specified by the permissions. The server proxy then creates a client-side secure session proxy description with a handle to the server-side secure session proxy, encrypts this using the client credential and sends it to the client authentication proxy. The client proxy decrypts and installs the client-side secure session proxy. Control of the secure connection is passed from the authentication proxies to the session proxies and the requesting application is given a reference/handle to the client-side secure session proxy.

Figure 5 shows the application using the service it requested. From the application and service point of view this appears to be a direct communication in an ideal world where communication is secure, and the client only makes requests that are allowed.

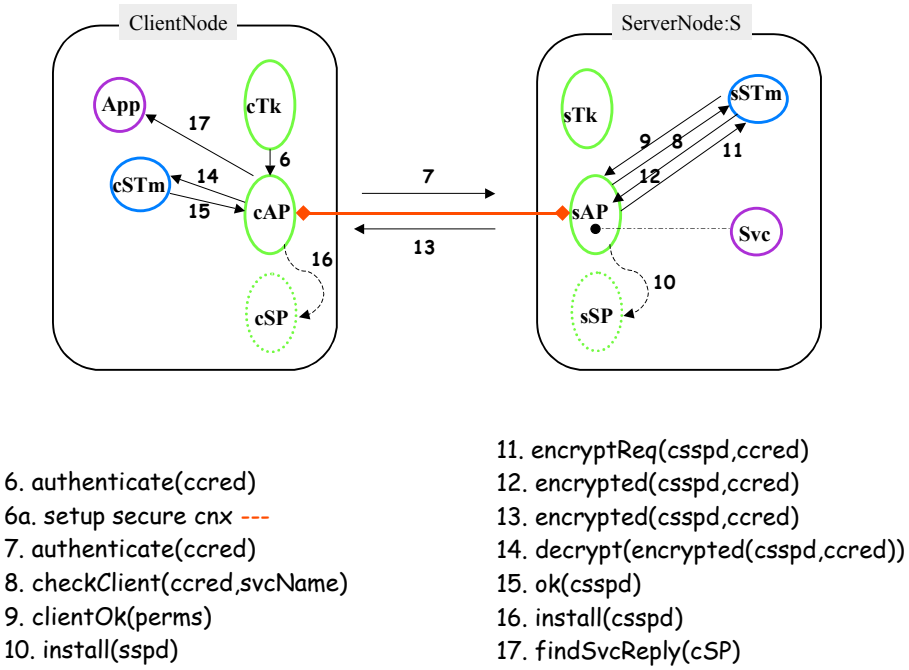

Figure 4: SSPTK Authentication and Secure Session Setup

3 Formal Models and Analysis: Overview

We have developed a family of formal executable specifications of SPTKs providing increasing levels of security (0-3), with level 3 corresponding to SSPTK. We also model attackers with different capabilities, both passive and active. The models are intended to specify correct use of basic security services such as SSL, and cryptographic signatures. These basic services are modeled abstractly in the form of a security and trust management service. Analysis is carried out on systems composed of clients and servers with toolkit components, a registry (also called lookup) node, and selected attackers. The specifications are written in the rewriting logic language Maude and analysis carried out using tools provided by the Maude system.

3.1 About Maude and the Maude approach to formal modeling

Maude [2, 1] is a specification language with supporting analysis tools based on rewriting logic. Rewriting logic is a very simple logic in which the *state space* of a distributed system is formally specified as an algebraic data type by giving a set of sorts (types), operations, and equations. The *dynamics* of such a distributed system is then specified by rewrite rules of the form

$$t \longrightarrow t'$$

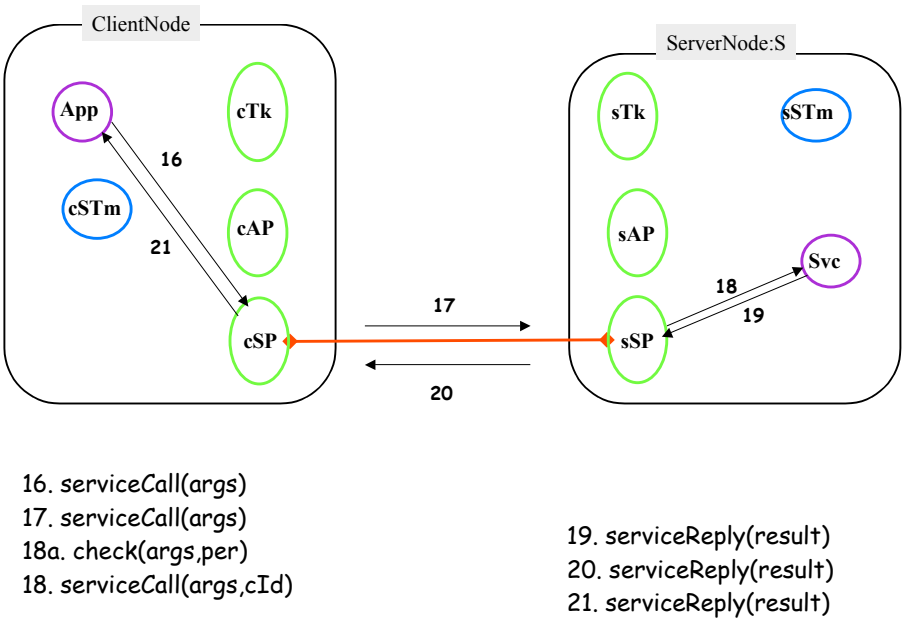

Figure 5: SSPTK Service Use Scenario

where t, t' are terms (patterns) that describe the *local, concurrent transitions* possible in the system. Specifically, when a part of the distributed state fits the pattern t, then it can change to a new local state fitting the pattern t'. Proofs in rewriting logic can also be thought of as computations of the system represented by the initial term. Maude provides a very efficient rewriting engine, supporting use of executable models as prototypes. Maude 2.0 also provides the capability to search the state space reachable from some initial state by application of the rewrite rules. This can be used to determine one or more reachable states satisfying a user defined property. There is also a model-checker for checking properties of a system expressed in linear temporal logic.

The formal methodology underlying our approach can be summarized by stating that *a small amount of formal methods can go a long way*. Approaches requiring full mathematical verification of a system can be too costly. But there are many important benefits that can be gained from "lighter" uses of formal methods, without necessarily requiring a full-blown proof effort. In this case study we used three levels of effort—formal specification, execution, and state-space search—in an iterative process.

The process of formal specification forces one to clarify ambiguities and hidden assumptions present in an informal specification. Some care was taken to organize the modules both to maximize reuse of common features and to clarify the differences between levels of protection. The specifications were first debugged and validated by defining prototypical initial

system configurations and using the Maude interpreter to execute the system and examine final states.

Since we are specifying distributed systems there are in general many possible executions starting with a particular initial configuration. The Maude interpreter picks a particular execution. Search and model-checking can be used to explore all possibilities in the case of finite state systems. Because of our use of abstractions, such as abstract services, where detail was not needed, our test configurations generated finite state systems. Properties related to the SSPTK security goals were expressed as search patterns and Maude's search tool was used to check these properties. Model-checking can also be used to for states satisfying simple search patterns as well as more complicated temporal properties.

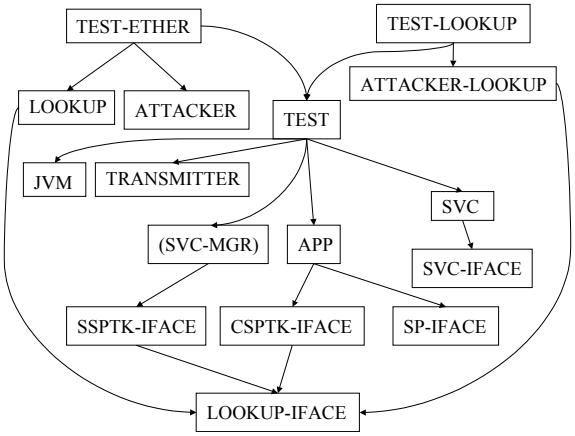

Figure 6: SPTK Architecture Modules

3.2 Model organization

Here we give a brief summary of the structure of the specification (and where the pieces can be found relative to the SPTK directory), and of the analyses carried out. The next section contains a more detailed discussion of how different aspects are represented and how the analyses are carried out.

Maude specifications are organized in modules which form an importation hierarchy. Figure 6 shows the module hierarchy (upper-levels) of the Maude SPTK specification. This corresponds to the architecture of Figure 1, elaborated with an explicit representation of the infrastructure and attack models. The module `TEST` models the SPTK architecture shown in Figure 1 and is common to (imported by) all levels. It assembles the component models and interfaces into a distributed system ready for TESTing (once the interfaces have been realized). The modules `JVM` and `TRANSMITTER` model the Java execution environment and the communication infrastructure provided by the environment, respectively. The module `APP` specifies a generic application that generates requests to find and use services. `SVC` specifies an abstract service—whose replies to requests are represented by unspecified functions of the current state,

and the request data. SVC-IFACE specifies the service component interface—the messages it can receive and send. The service manager (SVC-MGR) simply starts things off by registering some services, thus it is modeled by a set of registration requests in the initial system configuration. There is no need for an explicit module, hence the ()s. The modules SSPTK-IFACE and CSPTK-IFACE specify the interfaces to the service and client side toolkit components. LOOKUP-IFACE specifies the interface to the Lookup service, that models the Java registry.

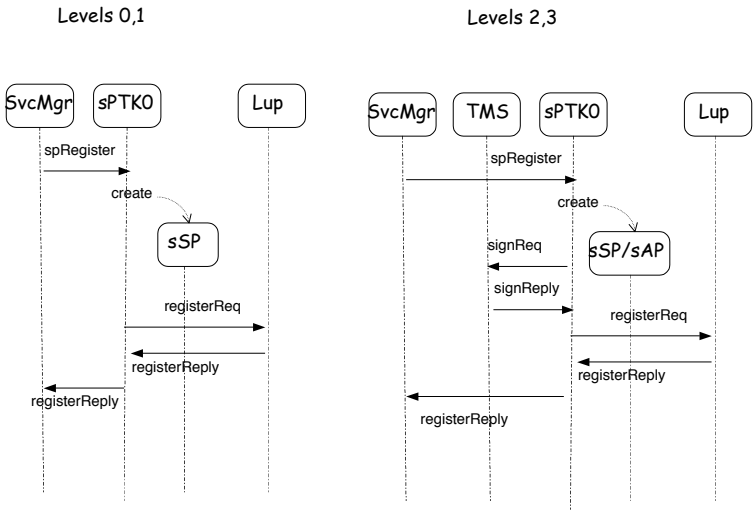

Figure 7: Levels 0-3 service registration

The shared modules also include the attacker models: one in which the attacker controls the network and the other in which the attacker controls the registry and lookup service. In both cases the attacker is also able to use its toolkit to register and access services. The module LOOKUP specifies an uncompromised Lookup service, while the module ATTACKER-LOOKUP specifies a lookup service that has been taken over by an attacker. Both implement the interface specified in LOOKUP-IFACE, but in the latter case, replies to find requests can be anything the attacker can construct. We have restricted the attacker to return some registered service (possibly one registered by the attacker) as non-service replies are just a form of denial of service, which we are not addressing. That is we assume that the client side toolkit at least checks that what it receives in response to a find request has the right form. The module ATTACKER models the situation in which the network is controlled an attacker. In this situation messages can be dropped, duplicated, generated, or modified. Again we restrict attention to modification, assuming that the toolkits ignore messages that are not expected, or not of the expected form. The module TEST-ETHER composes the ATTACKER and LOOKUP modules with the TEST module to form the shared core for configurations based on the *attacker in the network* assumption. Similarly, the module TEST-LOOKUP composes the ATTACKER-LOOKUP module with the TEST module to form the shared core for configurations based on the *attacker*

in registry assumption. These modules can all be found in the directory `STPK/All`. The file `SPTK/summary-all.txt` contains a complete listing of the modules defined in All and the imports of each module, along with brief comments. Included among the shared modules is the module STM, used by levels 2 and 3. This module specifies a security and trust management class that abstracts the details of key management, encryption/decryption, signing, and policy-based trust decisions—including signature checking.

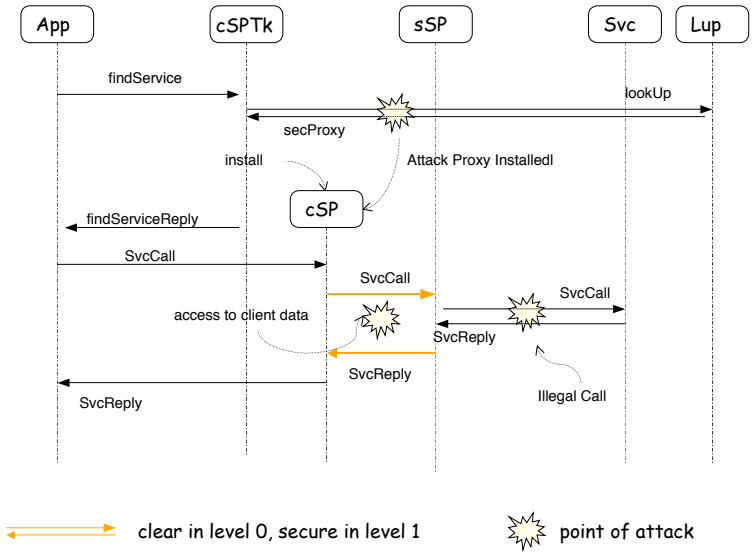

Figure 8: Levels 0-1 service lookup and use

For each security level the modules

```
TEST-ETHER<level> = TEST-ETHER + TK<level>
```

and

```
TEST-LOOKUP<level> = TEST-LOOKUP + TK<level>
```

compose the toolkit implementation for that level (`TK<level>`) with the corresponding core with attacker module to produce the full specification for that level.

Figure 7 shows registration scenarios for the four levels. Levels 0 and 1 have the same pattern as to levels 2 and 3. The difference in the latter case is the message content. Figure 8 shows service lookup and use scenarios for levels 0 and 1. Again the pattern is the same, in level 1 communication between proxies is encrypted. Figures 9 and 10 are service lookup and use scenarios for levels 2 and 3 respectively. In the lookup and use scenarios the "explosion" icons indicate potential points of attack. Slashed explosions indicate that the level protects against these attacks.

Level 0 is for use in situations with no attacker. The level 0 toolkit `TK0` is the composition `CSPTK0 + SSPTK0 + CSP0 + SSP0`. The job of the client and server side proxies (instances of the classes specified in `CSP0` and `SSP0`) is simply to make interaction appear local

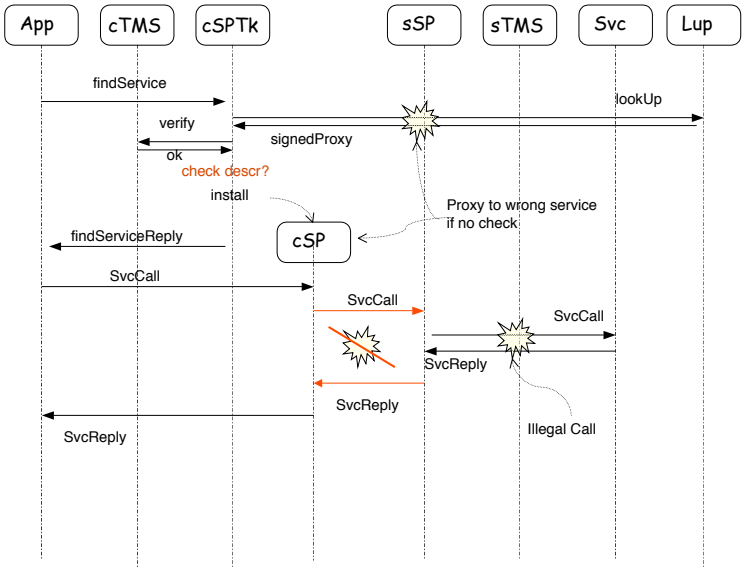

Figure 9: Level 2 service lookup and use

to the application and service, respectively. The server-side toolkit (`SSPTK0`) handles a register request by installing a level 0 server side proxy, and registering a description of a level 0 client side proxy with the lookup service. The client side toolkit (`CSPTK0`) handles a find request by installing the proxy described in the lookup service reply. This level achieves security goal I by relying on the underlying JVM to protect the host system.

Level 1 provides protection against an attacker that can observe and modify communications between the client and server. Such an attacker aims to obtain clients private information and might also modify service calls and replies. The level 1 toolkit (`TK1`) is the composition `CSPTK0 + SSPTK1 + CSP1 + SSP1`. The level 1 proxies (instances of the classes specified in `CSP1` and `SSP1`) communicate using secure connections (for example SSL). The toolkit components differ only from level 0 in that the server side component works with level 1 proxies rather than level 0. This level achieves security goal II.

Level 2 provides protection against an attacker that can observe and modify communication between the lookup service and client or server as well as client server communication. The level 2 toolkit (`TK2`) is the composition `CSPTK2 + SSPTK2 + CSP1 + SSP1 + STM`. The proxies of level 2 are the same as level 1. To foil the attacker, the level 2 server toolkit signs the level 1 proxy, using the servers security and trust component specified by the module `STM`, before registration, thus modification can be detected. The client side toolkit checks whether a proxy obtained by lookup was registered by a trusted server by checking the signature, using the clients security and trust component. At level 2a the registered level 1 proxy also contains the service description and the client side toolkit additionally confirms that the proxy received contains the description that was requested. This level achieves security goal III.

Level 3 adds client authentication to insure that the requests are from the claimed client

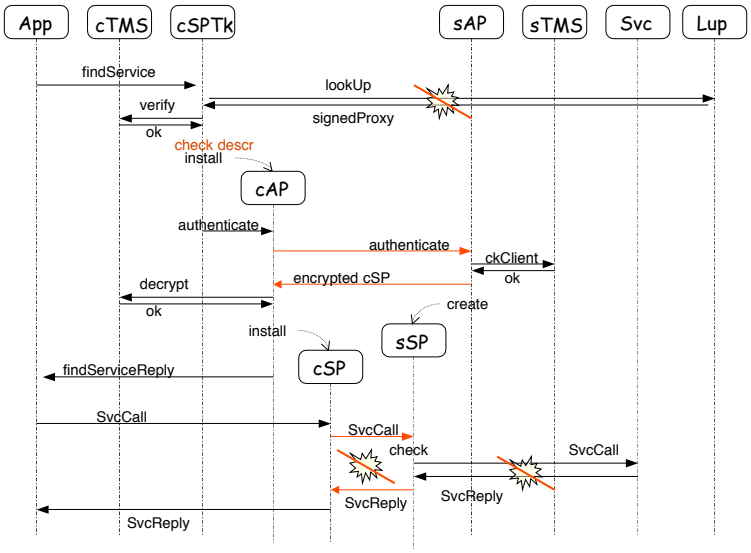

Figure 10: Level 3 service lookup and use

and that they are allowed for this client. This prevents an attacker from impersonating a client, thus possibly corrupting the servers data or obtaining client secrets that result from queries that contain only public data. This is achieved by using an additional pair of authentication proxies. The level 2 toolkit (TK3) is the composition

```
CSPTK3 + SSPTK3 + SAP + CAP + CSSP +  SSSP + STM.
```

The modules SAP and CAP specify the server and client side authentication proxies. To register a service, the server side toolkit installs a server side authentication proxy and registers a description of the client side authentication proxy. When the client side toolkit receives an authentication proxy in reply to a find request, it first checks the signature and service description. If the check succeeds, it installs the proxy and invokes its authenticate method, passing the clients credentials. The client authentication proxy sends these credentials to its server side peer (over a secure connection). The server authentication proxy checks the credentials, using the servers security and trust manager, and obtains a description of the requests that the client is permitted to make. The server authentication proxy then creates a server proxy, giving it the client id and permissions. It next creates a client side proxy giving it a handle to the server proxy. This proxy is serialized and encrypted using the client credentials, and sent to the client authentication proxy. The client authentication proxy decrypts this proxy and installs it. The encryption completes the client authentication by ensuring that only the intended client can decrypt and install a client proxy with access to the newly constructed service proxy that holds the clients permissions. The client and service proxy communication over a secure authenticated connection, and the service proxy checks permissions before forwarding a request to the service. This level achieves security goal IV.

Although we didn't organize the models this way, it would be possible to provide client and server authentication independently.

3.3 Summary of analysis results

Before going into detail about the formal models we summarize the analysis results for each level and attack model. In these summaries, the symbol + means the attacker succeeds while - means that the attacker fails. The attacks demonstrate the need for not only checking signatures, but also that the proxy represents the requested service.

	1.1	1.2	1.3	1.4
Level 0	+	+	+	+
Level 1	--	+	+	+
Level 2	--	+	+	+
Level 2a	--	--	+	+
Level 3	--	--	--	--

1.1 attacker can see/modify client data sent in service calls and replies

1.2 client accepts wrong proxy

1.3 unauthorized service call succeeds

1.4 imposter succeeds in forging client id

+ means yes, - means no

Figure 11: Analysis results for Attacker in the Network

Figure 11 summarizes analyses for the "attacker in the network" attack model, while Figure 12 summarizes analyses for the "compromised registry" attack model. In both cases, we see that all of the attacks succeed when the level 0 toolkit is used, while none succeed when the level 3 toolkit is used. For attacks on client-server communication (1.1) level 1 protection is sufficient. In the case of the compromised lookup service, we don't worry about client-server communication, but do a sanity check to see if it is possible for the client to find and use the requested service, and here the + means that the client can succeed in all cases. For attacks on communication with the registry (1.2) level 2 protection is sufficient if both the server signature and the service description are checked (level 2t). The distinction is made more precise in

the compromised registry model where level 2f is sufficient protection to keep the client from accepting a proxy to a service provided by an untrusted server (property 2.2), while level 2t is required to assure that a proxy to the correct service is accepted (2.3). Properties 1.3, 1.4, and 2.4 deal with situations in which the server is tricked into serving improper requests. These attacks require level 3 protection.

	2.1	2.2	2.3	2.4
Level 0	+	+	+	+
Level 1	+	+	+	+
Level 2	+	--	+	+
Level 2a	+	--	--	+
Level 3	+	--	--	--

2.1 client can obtain proxy for requested service (sanity check)

2.2 client accepts proxy to attacker service

2.3 client accepts wrong trusted server proxy

2.4 service integrity violated

+ means yes, - means no

Figure 12: Analysis results for Compromised Registry

4 Maude modeling details: shared core

We introduce the Maude syntax by way of some small examples that are part of the shared core of the SPTK modules. This should be sufficient to be able read and understand the specification. For full details, the Maude manual can be found at `maude.cs.uiuc.edu`, along with a primer for a first introduction to writing Maude specifications.

The Maude language provides syntax for expressing rewrite theories as Maude modules. There are two kinds of module: functional modules and system modules. Functional modules are used to specify data types (called *sorts*) and operations, while system modules also allow specification of behaviors using rewrite rules.

To represent byte streams (serialized objects, message payload, etc.) we introduce a sort `Data`. To abstract from the details of printed representation, data constructor functions are

used in place of operations to print to a byte stream, and pattern matching is used in place of parsing. This is specified in the the functional module, DATA, shown below.

```
fmod DATA is
  sorts Data DataX .
  subsort  Data < DataX .
  op badData : -> DataX .
endfm
```

As can be seen, functional modules are declared with the syntax

fmod <module-name> is ... endfm.

This module declares two sorts, Data and DataX (keyword sorts) and specifies that Data is a subsort of DataX (keyword subsort) . The subsort declaration says that the set denoted by Data is a subset of that denoted by DataX. The sort DataX can be thought of as an error supersort of Data. It contains elements that represent ill-formed or illegal data. In Maude constants are considered to be operations with no arguments and are declared using the syntax

op <constant-name> : -> <constant-sort> .

giving the constant's name and its sort. The third line of the DATA module declares a constant badData of sort DataX as a canonical non-data element of DataX.

For simplicity service descriptions are taken to be strings identifying the service. The state of a lookup service (registry) is represented as a multiset of service entries, called a ServiceTable, where a service entry is simply a pair consisting of a string identifying the service and the serialized service proxy description, considered to be uninterpreted data by the lookup service. The ServiceTable sort is specified in the functional module SERVICE-TABLE.

```
fmod SERVICE-TABLE is
  inc DATA .              *** defines Data sort
  inc STRING .            *** defines String sort
  sort ServiceTable .
  sort ServiceEntry .
  subsort ServiceEntry < ServiceTable .

  op _:=_ : String Data -> ServiceEntry .
  op noEntry : -> ServiceEntry .

  op svcName : ServiceEntry -> String .
  eq svcName(q:String := sp:Data) = q:String .

  op svcProxy : ServiceEntry -> Data .
  eq svcProxy(q:String := sp:Data) = sp:Data .

  op none : -> ServiceTable .
  op _,_ : ServiceTable  ServiceTable ->  ServiceTable
             [assoc comm id: none] .
```

```
    op lookup : ServiceTable String -> ServiceEntry .
    eq lookup(none,q:String) = noEntry .
    eq lookup((svcE:ServiceEntry,st:ServiceTable),q:String) =
        if svcName(svcE:ServiceEntry) == q:String
        then svcE:ServiceEntry
        else lookup(st:ServiceTable, q:String)
        fi  .
endfm
```

This module imports two submodules (keyword `include`): `DATA` and `STRING`, a built-in Maude module that axiomatizes a string data type. Importation means that all the sorts and operations declared in the imported modules are available in the importing module. Two sorts, `ServiceEntry` and `ServiceTable`, are declared. The 'pairing' operation for constructing service entries is the infix operator `_:=_`. A constant `noEntry` of sort `ServiceEntry` is declared, to be used to indicate lookup failure. Two selector operations, `svcName` and `svcProxy`, are declared and defined, by pattern matching equations, to select the components of a proper service entry. A service table is a multiset of service entries. This is expressed by declaring a constant `none` of sort `ServiceTable` (the empty table), an operator `_,_` (multiset union), and by declaring that `ServiceEntry` is a subsort of `ServiceTable`. The subsort declaration means that service entries can be thought of as singleton multisets. The attribute declaration `[assoc comm id: none]` for the operator `_,_` says that this operator is associative and commutative and has `none` as identity, exactly the properties needed for multiset union. More generally an operator declaration has the form:

```
op <opname> : <arg-sort-1> ... <arg-sort-n> -> <result-sort>
   [ <attributes> ] .
```

where the operator name `<opname>` can be a simple identifier (indicating prefix syntax) or an identifier containing underscores (indicating mixfix syntax) where each underscore corresponds to an argument position. In the `SERVICE-TABLE` module, `svcName` is a prefix operator while `_,_` and `_:=_` are mixfix operators. The sort list before the arrow specifies the arguments sorts and the sort after the arrow specifies the result sort. As discussed above, operators with no arguments (and hence no underscores), such as `none`, and `noEntry` are constants. The attribute part of an operator declaration is optional and can be used to specify additional properties including equations satisfied (as for `_,_`), formatting, evaluation strategies, an so on. Equational attributes are not only a convenient way of expressing common equational patterns, but more importantly rewriting takes place modulo these equational axioms, that is, rewrite rules operate on the resulting equivalence classes in an efficient manner. The operators `_,_` and `_:=_`, `none`, and `noEntry` are *constructors* used to give canonical representations of the denoted values. The meanings of the operators `svcName`, `svcProxy` and `lookup` are given by explicit equations, declared with the keyword `eq`. Semantically equation declarations have the usual meaning. Operationally such equations are used in the left-right direction to rewrite terms to a canonical form. We use 'inline' variable declarations. Such a declaration consists of the variable name and its sort, separated by a '`:`'. Thus in the statement

```
eq lookup(none,q:String) = noEntry .
```

the term `q:String` is a variable named `q` of sort `String`. This equation says that the result of lookup in an empty table (`none`) for any key is the constant `noEntry`.

In our SPTK models, a system consists of a soup (multiset) of JVM objects. together with an Ether object. A JVM object models a node running a Java virtual machine, and the Ether object abstracts the communication medium. Each JVM object contains a configuration of objects and messages (client applications, services, proxies, and so on). It manages object creation (name generation) and provides remote communication service.

We represent objects using the following syntax:

```
< o : C | a0(v0),  ..., an(vn) >
```

where `o` is the object identifier, `C` is the class identifier, and `a0(v0), ..., an(vn)` are the object attributes (fields) with `a0` being an attribute name and `v0` its value, and so on. This syntax is declared in the module `CONFIG` in the file `<SPTKpath>/All/jvm.m`.

For example a JVM object has the form

```
< J : JVM | oCtr(n),  ether(eee), { ...  } >
```

where `J` is the objects (unique) identifier, `JVM` is the class identifier, `oCtr(n)` is the object counter attribute, used to generate new object identifiers, `ether(eee)` is the ether attribute that keeps the identifier of the ether object that the JVM object uses to communicate with other JVMs, and `{ ... }` is the internal object configuration attribute.

The module `JCONF` (in the file `<SPTKpath>/All/jvm.m`) extends `CONFIG` with messaging primitives. Ordinary object communication is modeled using asynchronous messages of the form:

```
msg(to, from, msgbody)
```

where `to` is the (identifier of the) message target (intended receiver), `from` is the sender, and `msgbody` is the message body (contents), of sort `MsgBody`. A method call is represented by a (request,reply) pair of message body constructors or (request, OkReply, FailReply) triple to treat failures. We will see examples below. Secure communication (say via an SSL connection is modeled by an analogous message constructor

```
smsg(to, from, msgbody)
```

the difference being that only the sender and receiver can see the message body.

The abstract service specification is a simple system module that is typical of how we have organized modules that specify the behavior of a class of objects. The specification consists of three modules.

The module `SVC-IFACE` specifies the messages that a service object exchanges with other objects.

```
fmod SVC-IFACE is
  inc JCONF .
  inc DATA .
  inc STRING .
  op svcReq : Data String -> MsgBody .
  op svcReply : DataX -> MsgBody .
endfm
```

Service method calls are represented by a request-reply pair of message body constructors

```
svcReq(request-data,requestor-id)
svcReply(reply-data)
```

The module SVC-CLASS extends SVC-IFACE to specify a class identifier SVC and an attribute constructor, state.

```
fmod SVC-CLASS is
  inc SVC-IFACE .
  op state : Data -> Attribute .
  op SVC : -> Cid .
endfm
```

This module is sufficient to be able to create service objects. A service object in its initial state might look like

```
< someSvc : SVC | state(initData) >
```

where someSvc is the object identifier of the service, a constant of sort Oid, and initData is a constant standing for the services initial data.

The module SVC is a system module extending SVC-CLASS with rules specifying the behavior of service objects.

```
mod SVC is
  inc SVC-CLASS .
  op update : Data Data String -> Data .
  op reply : Data Data String -> Data .

  rl[svc.svcreq]:
  < svc:Oid : SVC | state(d:Data) >
  msg(svc:Oid, C:Oid, svcReq(dreq:Data,id:String))
  =>
  < svc:Oid : SVC | state(update(d:Data,dreq:Data,id:String)) >
  msg(C:Oid, svc:Oid, svcReply(reply(d:Data,dreq:Data,id:String))) .
endm
```

Two functions update and reply are declared, representing the fact that the next state and the reply are functions of the request history. For purposes of the current case study, we don't care what these functions are, services are distinguished simply by their initial data. There is one rule labeled svc.svcreq. The rule says that if a service object with state d:Data receives a message from a client C:Oid with body svcReq(dreq:Data,id:String) then it updates its state, and sends a reply to the client.

The security and trust management service is specified in the module STM. Here we just show the interface declarations, that specify the possible interactions with this service. Several new sorts are introduced.

```
  sorts Policy Key Cred Permissions .
```

The sort `Policy` abstracts details of security and trust management policies, using abstract credentials (sort `Cred`), and permissions (sort `Permissions`).

The services provided by the STM are: signing and signature verification

```
  op  signReq : Data -> MsgBody .
  op  signReply : Data -> MsgBody .
  op  signed : Data  Key -> Data .

  op  verifySigReq : Data Key -> MsgBody .
  op  verifySigOk : Data -> MsgBody .
  op  verifySigFail : -> MsgBody .
```

data encryption and decryption

```
  op  encryptReq : Data Cred -> MsgBody .
  op  encryptReply : Data -> MsgBody .
  op  encrypted : Data Key -> Data .

  op  decryptReq : Data -> MsgBody .
  op  decryptReply : DataX -> MsgBody .
```

and permission checking.

```
  op getId : Permissions -> String .
  op  checkClientReq : String Cred -> MsgBody .
  op  checkClientReply : Permissions -> MsgBody .
  op  clientPerm : String Cred Policy -> Permissions .
```

5 Maude modeling: attacker in the registry

The attacker in the registry model assumes that the registry service (called `LOOKUP` here) is compromised. We model this by an implementation of the `LOOKUP` interface by an object that collects registered proxies, forgetting the name, and replies to lookup requests with a non-deterministic choice from its proxy collection. In addition to the faulty lookup server, on the same node are a service manager that registers evil services, and an application that tries to compromise good services by making fake or illegal calls. To support the evil server and application the node also has both client and server side SPTKs.

The module `ATTACKER-LOOKUP-CLASS` includes the interface specifications and declares additional attributes.

```
fmod ATTACKER-LOOKUP-CLASS is
  inc LOOKUP-IFACE .
  inc ATTACKER-AUX .
  ops AttackerLookup : -> Cid .
  ops svcSet mySvcSet attackeeSet : DataSet -> Attribute .
  ops myCPTK mySPTK : Oid -> Attribute .
endfm
```

The attributes myCPTK and mySPTK store the identifiers of the local SPTKs. The attribute mySvcSet stores proxies registered by the local service manager, while the attribute svcSet stores proxies registered by others. Proxies registered by others are also placed in the attribute attackeeSet. This attribute is used to generate attacks on legitimate services, and once an attack has been launched on a service its proxy is removed from attackeeSet to ensure a finite number of attacks (assuming a finite number of registrations).

The module ATTACKER-LOOKUP includes ATTACKER-LOOKUP-CLASS and specifies rules for handling registry and lookup requests. The rule labeled lookup.register applies when a message of the form

```
msg(O:Oid, C:Oid, registerReq(sname:String, sp:Data))
```

is sent to an object of class AttackerLookup.

```
rl[lookup.register]:
  < O:Oid : AttackerLookup | myCPTK(T:Oid), mySPTK(ST:Oid),
                             mySvcSet(ds:DataSet),
                             svcSet(ss:DataSet),
                             attackeeSet(as:DataSet) >
  msg(O:Oid, C:Oid, registerReq(sname:String, sp:Data))
  =>
 (if (C:Oid == ST:Oid)
  then
   < O:Oid : AttackerLookup | myCPTK(T:Oid), mySPTK(ST:Oid),
                              mySvcSet(ds:DataSet sp:Data),
                              svcSet(ss:DataSet),
                              attackeeSet(as:DataSet) >
  else
   < O:Oid : AttackerLookup | myCPTK(T:Oid), mySPTK(ST:Oid),
                              mySvcSet(ds:DataSet),
                              svcSet(ss:DataSet  sp:Data),
                              attackeeSet(as:DataSet sp:Data) >
  fi)
  msg(C:Oid, O:Oid, registerReply) .
```

If the requester (named by C:Oid) is the local server toolkit (C:Oid == ST:Oid holds) then the data portion of the request (sp:Data) is stored in the attribute mySvcSet, otherwise it is stored in both svcSet and attackeeSet. In either case a message containing registerReply is sent to the requester. The rule labeled lookup.lookup applies when a message of the form

```
msg(O:Oid, C:Oid, lookupReq(sname:String))
```

is sent to an object of class AttackerLookup where the sender (C:Oid) is not the local client SPTK.

```
crl[lookup.lookup]:
  < O:Oid : AttackerLookup | myCPTK(T:Oid), mySPTK(ST:Oid),
```

```
                              mySvcSet(ds:DataSet),
                              svcSet(ss:DataSet),
                              attackeeSet(as:DataSet) >
   msg(O:Oid, C:Oid, lookupReq(sname:String))
   =>
   < O:Oid : AttackerLookup | myCPTK(T:Oid), mySPTK(ST:Oid),
                              mySvcSet(ds:DataSet),
                              svcSet(ss:DataSet),
                              attackeeSet(as:DataSet) >
    msg(C:Oid, O:Oid, lookupOk(sname:String, sp:Data))
 if C:Oid =/= T:Oid
 /\ choose(ss:DataSet ds:DataSet) => sp:Data  .
```

In this case an element of the union of the `svcSet` and `mySvcSet` attributes is selected non-deterministically and sent in a `lookupOK` reply. The choice is accomplished by the rewrite

```
choose(ss:DataSet ds:DataSet) => sp:Data
```

in the rule condition (following the `if`). The rule for choosing (labeled `choseD` is deceptively simple.

```
ops choose : DataSet -> DataX .
rl[choseD]: choose(d:Data ds:DataSet) => d:Data .
```

The selected element variable shown in the rule (`d:Data`) will match any element of the set `ss:DataSet ds:DataSet`. In fact in a search for all executions Maude will try all elements in turn.

6 Toolkit modules

To illustrate how toolkit functionality is modeled we will discuss some of the rules of the level 2 client-side SPTK class (with class identifier `CSPTK2`) specified in the module `CSPTK2`. In addition to messages specified in the client SPTK interface, objects of this class interact with a local security and trust management service (STMS) and create proxies from descriptions obtained from the registry. The attributes of such an object include

- `stms`–the identifier of the local STMS
- `lookup`–the identifier of a registry service
- `clientID`–a string identifying the client application.
- `ts`–the public key of a trusted server

In addition there is a state attribute which is either `idle` or of the form

```
waitfor(X:Oid,C:Oid,mb:MsgBody).
```

The latter is used to insure that a pending reply is handled before any additional requests are processed, and to remember what request is being processed. The first argument is the identifier of the object from which a reply is awaited, the second argument is the requester identifier, and the third argument is the request.

The rule labeled `csptk.findServiceReq` applies when a `findServiceReq` message arrives and the toolkit object is `idle`.

```
rl[csptk.findServiceReq]:
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                    clientId(cn:String), checksn(b:Bool),
                    ts(sk:Key),  idle >
 msg(O:Oid, A:Oid, findServiceReq(sn:String))
 =>
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                    clientId(cn:String), checksn(b:Bool), ts(sk:Key),
                waitfor(L:Oid, A:Oid, findServiceReq(sn:String)) >
 msg(L:Oid, O:Oid, lookupReq(sn:String))  .
```

In this case a `lookupReq` for the service identified by `sn:String` is sent to the lookup server whose identifier is stored in the `lookup` attribute. Also, the `idle` state attribute is replaced by a `waitfor` state. If a reply of the form `lookupOK(sn:String, signedPD:Data)` is received, the toolkit object sends a request to its STMS to verify the signature, using the trusted server key.

```
rl[csptk.lookupOk]:
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                clientId(cn:String), checksn(b:Bool), ts(sk:Key),
                waitfor(L:Oid, A:Oid, findServiceReq(sn:String)) >
 msg(O:Oid, L:Oid, lookupOk(sn:String, signedPD:Data))
 =>
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                 clientId(cn:String), checksn(b:Bool), ts(sk:Key),
                 waitfor(M:Oid, A:Oid, findServiceReq(sn:String)) >
 msg(M:Oid, O:Oid, verifySigReq(signedPD:Data,sk:Key) ) .
```

Now the `waitfor` attribute has the STMS identifier as its first argument. If the signature is verified, the reply body has the form `verifySigOK(pd:Data)` where `pd:Data` is the result of 'unsigning'. That is, it is the proxy description originally signed by the server.

```
rl[csptk.verifyOk]:
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                    clientId(cn:String), checksn(b:Bool), ts(sk:Key),
                 waitfor(M:Oid, A:Oid, findServiceReq(sn:String)) >
   msg(O:Oid, M:Oid, verifySigOk(pd:Data))
 =>
 < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                    pxd(pd:Data),
                    clientId(cn:String), checksn(b:Bool), ts(sk:Key),
                  waitfor(myJVM,A:Oid, findServiceReq(sn:String)) >
   msg(myJVM, O:Oid, newIdReq) .
```

The toolkit object saves this in an auxiliary attribute `pxd` and asks it containing JVM for an object identifier to use to create a proxy. The identifier `myJVM` is a special constant used to refer to an objects containing JVM. The reply to this request has the form `newIdReply(N:Oid)` where `N:Oid` can be used as the identifier of a newly created object, in this case the client-side service proxy.

```
rl[csptk.newidreply]:
   < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
         pxd(proxyD(cl:Cid, (sname(mysn:String), atts:AttributeSet))),
         clientId(cn:String), checksn(b:Bool), ts(sk:Key),
         waitfor(myJVM, A:Oid, findServiceReq(sn:String)) >
   msg(O:Oid, myJVM, newIdReply(N:Oid))
 =>
   < O:Oid : CSPTK2 | stms(M:Oid), lookup(L:Oid),
                     clientId(cn:String), checksn(b:Bool),
                     ts(sk:Key), idle >
   (if ((sn:String == mysn:String) or not(b:Bool))
    then
      < N:Oid : cl:Cid | sname(mysn:String), clientId(cn:String),
                         atts:AttributeSet,  idle >
      msg(A:Oid, O:Oid, findServiceOk(sn:String, N:Oid))
    else
      msg(A:Oid, O:Oid, findServiceFailed(sn:String))
   fi ) .
```

In general an object description has a class identifier and an attribute set. These are sufficient to create and object once an identifier is determined. In the case of a service proxy of level 2 or greater, there is a distinguished attribute, `sname`, that stores the string identifying the service. If this is not the same as that request by the application (and sent in the lookup request) a `findServiceFailed` message is sent to the requester. Otherwise a proxy is created (the new object in the rule conclusion) and a `findServiceOK` message containing the proxies identifier is sent to the requester.

There are two other situations in which a `findServiceFailed` message is sent to the requester. One is if the reply to the lookup request is a `lookupFailed` message, the other is if signature verification fails. Rules for these cases are included in the CSPTK2 specification.

7 Execution and Analyses

For execution and analysis we considered initial configurations with four objects: an object modeling the communication environment and three JVM objects one client, one server and one hosting the registry/lookup service. The client and server JVMs are independent of the attack model being considered, but parametric in the choice of toolkit objects, the choice depending on the security level. The choice of lookup and environment objects depends on the attack model. Recall that a JVM object has the form

```
< J : JVM | oCtr(n),  ether(eee), { ...  } >
```

where the ellipsis is to be filled by the object configuration running in the JVM. For our experiments, the server JVM is defined by

```
eq sjcf =
  < JS : JVM | oCtr(0),  ether(eee),
    { sptkobs
     < (JS .  GBPsvc) : SVC | state(GBPdata) >
     < (JS .  USsvc) : SVC | state(USdata) >
     msg(JS . ssptk, JS . fred,
         spRegisterReq("GBPquote", JS . GBPsvc))
     msg(JS . ssptk, JS . fred,
         spRegisterReq("USquote", JS . USsvc))
    } > .
```

where `sptkobs` stands for a server-side toolkit configuration to be defined for each level. This configuration must contain a server-side SPTK object with identifier `JS . ssptk` and possibly other objects. In addition to `sptkobs`, the configuration initially contained in the server JVM consists of two `SVC` objects (with different starting data) and two service registration requests.

The client JVM is defined by

```
eq cjcf =
      < JC : JVM | oCtr(0),  ether(eee),
    {  cptkobs
      < (JC . app) : APP | svcFound(none), mytk(JC . csptk),
                           todo(myTasks), done(none), idle >
     } >
```

where `cptkopbs` stands for a client-side toolkit object configuration containing at least an object with identifier `JC . csptk`. The application object (with identifier `JC . app`) has a set of tasks, `todo(myTasks)` that determine the services it will try to access. As an example the toolkit objects for Level 2 are defined by

```
eq sptkobs =
   ( < (JS . sstm) : STM | pkey(sPk), skey(sSk), policy(sP) >
     < (JS . ssptk) : SSPTK2 | stms(JS . sstm),
                               lookup(JL . lup),idle > ) .

eq cptkobs =
   ( < (JC . cstm) : STM | pkey(cPk), skey(cSk), policy(cP) >
     < (JC . csptk) : CSPTK2 | stms(JC . cstm), lookup(JL . lup),
                               checksn(true), clientId("sam"),
                               ts(sPk), idle > )  .
```

In addition to the level 2 client/server toolkit objects these configurations include a security and trust manager object (class `STM`).

For the case of attack in the network, the communication environment object (`eacf`) has class `Attacker` and the lookup JVM (`ljcf`) contains a normal lookup service (class `LOOKUP`).

```
eq eacf =
  < eee : Attacker | pxDs(none),
                     clientIds(none), clientDs(none),
                     inQ(none),  outQ(none) >  .
eq ljcf =
  < JL : JVM | oCtr(0), ether(eee),
               {< (JL .  lup) : LOOKUP | svctable(none) >} >  .
```

In the case of a compromised lookup service, the communication environment object has class `Transmitter` (it just forwards messages), and the lookup JVM contains a compromised lookup service (class `AttackerLookup`), as well as client and server toolkit objects (choice depending on security level) and an application that can be tricked into attacking a registered service.

The transmitter object (`eecf`) is defined by

```
eq eecf =  < eee : Transmitter | outQ(none), inQ(none)  > .
```

and the compromised lookup node (`aljcf`) is defined by

```
eq aljcf =
  < JL : JVM | oCtr(0),  ether(eee),
   {  acptkobs
     < (JL . app) : APP | svcFound(none), mytk(JL . csptk),
                          todo(task("GBPquote",fakeCall)),
                          done(none), idle >
     < (JL .  lup) : AttackerLookup |
             myCPTK(JL . csptk), mySPTK(JL . ssptk),
             mySvcSet(none), svcSet(none), attackeeSet(none) >
     < (JL . AttackSvc) : SVC | state(attackData) >
    } >  .
```

where for level 2 the parameter `aptkobs` is defined by

```
eq aptkobs =
   < (JA . cstm) : STM | pkey(aPk), skey(aSk), policy(aP) >
   < (JA .  csptk) : CSPTK2 | stms(JA . cstm), lookup(JL . lup),
                              clientId(fakeId), checksn(false),
                              ts(sPk), idle > .
```

To summarize, we considered two families of initial configurations:

- `eacf ljcf sjcf cjcf` for the attacker in the network case
- `eecf aljcf sjcf cjcf` for the attacker in the registry case

where different family members were obtained by using toolkits of different levels.

For each level, and each attack model one can simply rewrite the initial configuration using Maude's default rewrite strategy. The results of rewriting for level j can be found in the files

```
<SPTKpath>/L<j>/runs<j>-ether.txt
```

for the attacker in the network, and

```
<SPTKpath>/L<j>/runs<j>-lookup.txt
```

for the attacker in the registry. The attacker models lead to highly non-deterministic situations, so that a single execution is not very interesting. Instead we use the search command to look for situations of interest. For example the following command searches for a situation, in the attacker in the network model, in which the attacker has extracted client data from a service call message.

```
search [1] (eacf ljcf sjcf cjcf) =>+
   ( cf:Configuration
    < eee : Attacker | atts:AttributeSet,
                       clientDs(mycall cds:DataSet) > ) .
```

The `Attacker` subclass of the `Ether` class has an attribute `clientDS` where it stores data extracted from clear text messages. Data can not be extracted from secured (`smsg`) messages. In the search command, the `[1]` specifies that only one solution is wanted. The `+` after the arrow says that at least one rewrite must be done. The pattern after the arrow specifies the states of interest as any configuration matching the pattern. In this case we only care about the object with identifier `eee`, for it only the value of the `clientDs` attribute is constrained. As the table in Figure 11, property 1.1 indicates, the search succeeds at level 0 but not for higher levels since communication between proxies is secured at higher levels.

To search for a successful service find request in the attacker in the registry model we used the following command.

```
search [1]  (eecf aljcf sjcf cjcf) =>+
   ( cf:Configuration
    < JC : JVM | jcatts:AttributeSet,
      { cocf:Configuration
        < O:Oid : cl:Cid | svc(S:Oid), oatts:AttributeSet >
        < (JC . app) : APP | catts:AttributeSet,
                             todo(task(sn:String, cd:Data) ts:Tasks),
                             waitfor(sid:Oid)  >
          msg(JC . app, sid:Oid, findServiceOk(sn:String, O:Oid))
      } >
    < JS : JVM | jsatts:AttributeSet,
     {  socf:Configuration
       < S:Oid : spc:Cid |  sname(sn:String), satts:AttributeSet >
      } >
    )
   .
```

In this pattern, the application object (`JC . app`) is waiting for a reply to a request of the form `findService(sn:String)` (the first task in its `todo` list). The `waitfor` attribute indicates that the reply should come from `sid:Oid` and the expected message contains both the service

identification string and the identifier of the created client-side proxy (`O:Oid`). The client-side proxy object has an attribute `svc` whose value is the identifier of its server-side peer (`S:Oid`) and the object with identifier `S:Oid` in the server JVM contains its service identification string. Thus if a configuration matching this pattern is reached the application will be connected to the specified service. As the table in Figure 12, property 2.1 indicates, this search succeeds at all levels.

A slight modification of this pattern can be used to search for a situation in which the application is connected to a service other than the requested one. The service identification string in the server-side proxy is named by a different variable (`ssn:String`) and a condition is added to the search command, requiring that the values bound to `sn:String` and `sn:String` be different.

```
search [1] icf-aa =>+
   ( cf:Configuration
    < JC : JVM | jcatts:AttributeSet,
      { cocf:Configuration
        < O:Oid : cl:Cid | svc(S:Oid), oatts:AttributeSet >
        < (JC . app) : APP | catts:AttributeSet,
                             todo(task(sn:String, cd:Data) ts:Tasks),
                             waitfor(sid:Oid)  >
          msg(JC . app, sid:Oid, findServiceOk(sn:String, O:Oid))
      } >
    < JS : JVM | jsatts:AttributeSet,
     {  socf:Configuration
       < S:Oid : spc:Cid |  sname(ssn:String), satts:AttributeSet >
      } >
    )
   such that (ssn:String =/= sn:String) .
```

This pattern corresponds to property 2.3 in the table in Figure 12. The search succeeds in finding an attack in levels 0 and 1 and the weaker version of level 2. The search fails in the strengthened level 2 and level 3.

8 Conclusion

The objective of this case study was to formally model and analyze the Secure Service Proxy Toolkit developed by was developed by John Mitchell, Ninghui Li, and Derrick Tong as part of the Stanford-SRI Dynamic Coalitions project *Agile Management of Dynamic Collaboration*.

In the process of developing the model several ambiguities were clarified and one defect in the design was corrected. The formal model serves as a form of documentation of the design and can be used to study possible scenarios and alternative designs. In addition the security goals were formalized as patterns characterizing properties of states of interest.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. `http://maude.cs.uiuc.edu`.

[3] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.