The Design and Implementation of a JCA-Compliant Capture Protection Infrastructure
(Extended Abstract)

Michael K. Reiter* Asad Samar† Chenxi Wang†

Abstract

A capture protection server *protects a cryptographic key on a device that may be captured by authenticating the user of the device (e.g., by password) before permitting the key to be used.* Delegation *from one capture protection server to another enables the new server to perform this capture protection function for the device. Delegation, however, opens the system to new vulnerabilities, including difficulties in limiting online password-guessing attacks and in disabling a device that has been stolen by an attacker who knows the password. Here we propose a lightweight protocol for coordinating capture protection servers that eliminates these vulnerabilities. We also report on the implementation of our protocol in a JCA-compliant cryptographic service provider, and ramifications of the JCA interfaces for our approach.*

1 Introduction

Physical capture of a mobile device places at risk any cryptographic keys that the device holds, and any capabilities those keys engender. In this paper we consider a software approach to protect a key on a device that may be captured, in which the device interacts with a remote *capture protection server* when it must use its key. Since many uses of cryptographic keys require network connectivity anyway (e.g., in cryptographic protocols), this interaction seems acceptable in many cases, and will become more so as network connectivity continues to proliferate.

The challenges that motivate the present paper were brought about by a recent proposal in which a user may *authorize* a new capture protection server by *delegating* from an existing authorized server [15]. Her device can then interact with the newly authorized server in order to use its key. Delegating and revoking servers dynamically is appropriate as the device moves locations: e.g., if the user carries the device to a distant location, the device can avoid the latency of interacting with the original capture protection server and instead use a new server in its proximity. This is especially beneficial since possibly *every* cryptographic operation performed by the device requires interaction with an authorized server. As another example, the user could temporarily delegate to a smartcard, to permit the device to use its key with the smartcard inserted when it would not have network connectivity.

In this paper we present the design and implementation of a capture protection infrastructure that addresses the primary security and systems challenges that were left unaddressed in the original proposal. The security challenges are best understood by considering the specific function of a capture protection server, i.e., to confirm that the device remains in the possession of a legitimate user before permitting it to perform a cryptographic operation. Presuming a password is used to perform this confirmation (as in [15]), the server must limit the number of incorrect guesses against the device's password, lest it permit an attacker who has captured the device from progressing too far in an online dictionary attack. When servers are dynamically authorized, however, this may widen the window of vulnerability to such an attack: If the attacker captures the device, then it can mount an online dictionary attack against *each* currently authorized server, thereby gaining more password guesses than any one server allows. A second security challenge arises from the feature that a capture protection server can be *dis-*

*Department of Electrical & Computer Engineering and Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; reiter@cmu.edu

†Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; {asamar,chenxi}@ece.cmu.edu

This work was partially supported by the U.S. Defense Advanced Research Projects Agency and the U.S. National Science Foundation.

Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)
1060-9857/03 $17.00 © 2003 IEEE

abled for a device if the device is captured, even if the attacker has compromised the user's password. Delegation also poses challenges to disabling: If the device and password are compromised, and if there is some authorized server when this happens, then the attacker can delegate from this authorized server to *any* server permitted by the policy set forth when the device was initialized. Thus, to be sure that the device can never use its key again, every server in this "admissible" set must be disabled for the device.

Here we propose a lightweight mechanism for addressing these security issues. Our mechanism supports secure data sharing among capture protection servers in a way that reverses the negative effects of delegation. As a result, the number of password guesses permitted against a captured device is unaffected by the number of servers authorized for the device, and disabling the device at one authorized server has the effect of disabling the device at all servers. In addition, our approach exploits "locality of reference" by a mobile user, in two respects. First, our approach imposes communication overhead only when the device switches from using one capture protection server to using another; after one interaction to perform a cryptographic operation with the new server, there is no additional communication overhead for subsequent operations. Second, if delegation patterns follow a user's travels, the communication overhead of switching servers is typically incurred only between the new server and the previous one; there is no need to retrieve data from a distant "home location" or a designated server.

We describe the implementation of our technique within a *cryptographic provider* that is consistent with the Java Cryptography Architecture (JCA). An application would merely need to load our provider to transparently achieve capture protection benefits. We have also experimented with implementing *proactive updates* (c.f., [11]) for the device's key transparently via a protocol similar to delegation. We describe a tension between the JCA interfaces, which are designed around keys that are static, and proactive updates. This tension has far-reaching implications for our approach.

2 Related work

Online services that play a role similar (though not identical) to that of a capture protection server include the modified Kerberos server of Yaksha [10], a *semi-trusted mediator* [3], and a *security mediator* in server-aided signatures [9]. These servers are interposed in the critical path of a user performing cryptographic operations using her private key, and thereby can disable a private key that should no longer be used. To our knowledge, no prior effort other than that from which we build [15] has proposed a notion of delegation from one server to another, and consequently the issues that we attempt to address here have not been previously considered in these other efforts.

At the basis of our capture protection infrastructure is a novel protocol for achieving mutually exclusive access to a mobile object. Our protocol was inspired by prior algorithms for similar goals (e.g., [18, 17, 5, 7, 1, 19, 21, 4, 6]), but at the same time differs from them in significant ways. First, we assume a dynamic network topology determined by delegation patterns, whereas most prior work on distributed mutual exclusion for mobile objects (e.g., [18, 17, 7]) builds on static topologies. Second, we permit Byzantine node failures [14] within our attacker models, while most prior efforts in fault tolerant mutual exclusion for mobile objects deal with only benign node failures (e.g., [5, 1, 19, 21, 4, 6]).

Our implementation effort has uncovered challenges posed by the JCA interfaces. Though there are numerous implementations of standard cryptographic interfaces such as JCA [12] or GSS-API [13], we are not aware of prior work that uncovers similar issues arising from these interfaces. Key management implementations that change key state dynamically—notably toolkits such as [2] that support proactive management of private keys [11] (which we also achieve as a corollary of delegation)—have not strived to adhere to standardized interfaces and thereby have circumvented the difficulties that we uncover here.

3 Background in capture protection

A capture protection system consists of a device dvc and an arbitrary number of computers called *nodes*, each denoted nd. Each node can host (execute) multiple logical *servers*. A server is denoted by svr, typically with additional subscripts or other annotations. In our system, the device is used for generating digital signatures[1] (e.g., using RSA [20]), and does so by interacting with one of the servers over a public network. The signature operation is protected by a password π. The system is initialized with public data, secret data for the device, secret data for the user (i.e., π), and secret data for each node. The public and secret data associated with a node are simply a certified public/private key pair for the node, which are assumed to be established well before the device is initialized. We denote the public key of a node nd by pk_{nd}, and its private key by sk_{nd}. Each svr has a public key pk_{svr} that is simply the public key pk_{nd} of the node nd executing svr.

[1] The device can also be used for decrypting messages, however, for simplicity we only deal with signatures in this paper.

The device-server protocol allows a device operated by a user who knows π and enters it correctly to sign a message with respect to the public key of the device, after communicating with one of the servers. The device is initialized with one server available to it, denoted svr^* and executing on node nd^*, though the device can cause a new server to be created on another node via delegation. For dvc to deploy a new svr on a node, another existing server svr' must consent to delegating its authority to that node, after verifying that the creation of a server on that node is consistent with policy previously set forth by dvc (see below for details) and is being performed by dvc with the user's password. In this way, delegation is a protected operation just as signing is. The device can unilaterally *revoke* a server when it no longer intends to use that server. A node can be *disabled* (for a device) by being instructed to no longer respond to that device or, more precisely, to requests involving the device's key.

Here we will not specify a policy that determines the nodes to which dvc can delegate, here called the *admissible* nodes, though we do assume that the public key of such a node can be determined reliably. The policy that defines admissibility is user-tunable and must be set when dvc is initialized. An example policy might allow delegation to any node with a public key certified by a given certification authority. (Note this would also justify our assumption that the public key of an admissible node could be determined.) For such a policy, the admissible nodes are not known in advance and can change over time. Our approach is specifically designed to accommodate such flexibility.

Each attacker we consider controls the network; i.e., the attacker controls the inputs to the device and every node, and observes the outputs. Moreover, an attacker can permanently *compromise* certain resources. The resources that may be compromised by the attacker are any of the nodes, dvc, and π. Compromising reveals the entire contents of the resource to the attacker and permits the attacker to impersonate it. The one restriction on the attacker is that if he compromises dvc, then he does so after dvc initialization and while dvc is in an inactive state—i.e., dvc is not presently executing a protocol on user input—and the user does not subsequently provide input to the device. This decouples the capture of dvc and π, and is consistent with our motivation that dvc is captured while not in use by the user and, once captured, is unavailable to the user.

We formalize different aspects of the system described thus far as a collection of *operations*.

1. dvc.delegate(svr, nd): dvc performs a delegation with server svr, using the correct password π, to deploy a new server on nd.
2. dvc.revoke(svr): dvc revokes svr, indicating it will not be using svr in the future.
3. nd.disable: nd stops responding to any requests from dvc (signing or delegation).
4. dvc.comp: dvc is compromised.
5. nd.comp: nd is compromised.
6. π.comp: the password π is compromised.

We note that nd.comp compromises all servers ever hosted by nd. When convenient, we will use svr.comp to denote nd.comp where nd is the node hosting svr.

3.1 Overview of algorithms

Here we provide only the essentials of how the delegation and signature protocols of [15] work. The capture protection system requires a device initialization phase, for which the device dvc takes as input its private key sk_{dvc}, the password π, and the identity of node nd^* with public key pk_{nd^*}. The output of initialization is a *ticket* τ^* and an associated *authorization record* $\mathsf{authrec}_{\tau^*}$ containing secret information for the device. τ^* is a ciphertext encrypted under pk_{nd^*} by dvc, the plaintext for which is generated as a function of π, a secret stored in $\mathsf{authrec}_{\tau^*}$, sk_{dvc}, and an "inner ticket" ζ^* that is itself a ciphertext encrypted under pk_{nd^*}.

Cryptographic operations by dvc require that dvc use a ticket τ and authorization record $\mathsf{authrec}_\tau$ to induce the creation of a logical server at the node nd able to decrypt τ for processing requests bearing the ticket τ; we denote this server by svr_τ. (In particular, $\mathsf{svr}^* = \mathsf{svr}_{\tau^*}$.) nd initializes state for svr_τ including a counter $\mathsf{svr}_\tau.\mathsf{ctr} \leftarrow 0$ for counting requests bearing τ but reflecting incorrect password guesses.

dvc can then interact with svr_τ to either sign messages or delegate. To do so, dvc generates a request req as a function of π and the secret stored in $\mathsf{authrec}_\tau$, as well as the request parameters: The message m to be signed if a signature operation, or the identity and public key $pk_{\mathsf{nd}'}$ of nd' if delegating to nd'. dvc then invokes $\mathsf{svr}_\tau.\mathsf{doOperation}(\mathsf{req})$, which proceeds as in Figure 1. As shown, svr_τ first determines if: req and τ were created using different dvc secrets (line 2 of Figure 1); the password mistype counter $\mathsf{svr}_\tau.\mathsf{ctr}$ is already at its maximum, q_{svr_τ} (line 4); or req and τ were created using different passwords (line 6). If any of these conditions occur, the request is aborted (lines 3, 5, and 8). Otherwise the request is processed according to the req.opType field (which is a string constant, either `sign` or `delegate`) and a response is returned (lines 9–12).

If a signature operation, dvc completes the signature for m upon receiving a valid response from svr_τ. If a

```
1.  svr_τ.doOperation(req)                    /* can be invoked remotely (by some dvc) */
2.      if (¬fromSameDvc(req, τ))
3.          return ⊥                          /* return if τ and req are not from same device */
4.      if (svr_τ.ctr = q_svr_τ)              /* q_svr_τ is max # of allowed bad password guesses at svr_τ */
5.          return ⊥                          /* return if already at max # of bad password guesses */
6.      if (¬fromSameUser(req, τ))
7.          svr_τ.ctr ← svr_τ.ctr + 1         /* record a bad password guess */
8.          return ⊥                          /* return on bad password guess */
9.      if (req.opType = "sign")
10.         return svr_τ.handleSignReq(req)   /* process sign request and return result */
11.     else
12.         return svr_τ.handleDelReq(req)    /* process delegation request and return result */
```

Figure 1. svr_τ.doOperation **algorithm adapted from [15]**

delegation to nd′, the response enables dvc to generate a ticket τ' encrypted under $pk_{\mathsf{nd}'}$. In this case, the inner ticket ζ' is generated by svr_τ and sent to dvc for inclusion in τ'. ζ' is used to convey secret information from svr_τ to the yet-to-be-created $\mathsf{svr}_{\tau'}$.

3.2 Security

We say that svr_τ is *authorized at time t* if either (i) $\tau = \tau^*$ or (ii) at some $t' < t$ and before dvc.comp, dvc performs dvc.delegate(svr′, nd) with a svr′ authorized at time t' to obtain output $<\tau, \mathsf{authrec}_\tau>$, and no dvc.revoke($\mathsf{svr}_\tau$) occurs before t. In (ii), svr′ is the *consenting server*. In contrast to [15], svr* is always authorized by (i). We motivate this in Section 4.

We divide attackers into four nonoverlapping classes, based on what they compromise and when. We assume an attacker falls into one of these classes nonadaptively, i.e., it does not change its behavior relative to these classes depending on system execution.

- A1. An A1 attacker does not compromise dvc.
- A2. An A2 attacker compromises dvc, does not compromise π, and compromises no server authorized at the time of dvc.comp.
- A3. An A3 attacker compromises dvc, does not compromise π, and compromises some server authorized at the time of dvc.comp.
- A4. An A4 attacker compromises both dvc and π, but does not compromise any admissible node.

The security goals achieved in [15] against these attackers are as follows:

- G1. An A1 attacker is unable to forge signatures for dvc.
- G2. An A2 attacker can forge signatures for dvc with probability at most $\frac{q}{|\mathcal{D}|}$, where q is the total number of queries to authorized servers after dvc.comp, and $\mathcal{D}$ is the dictionary from which the password is drawn (assumed uniformly at random).
- G3. An A3 attacker can forge signatures only if it succeeds in an offline dictionary attack on the password.
- G4. An A4 attacker can forge signatures only until all admissible nodes are disabled for dvc.

These properties can be more intuitively stated as follows. If an attacker does not capture dvc (A1), then the attacker gains no ability to forge for the device (G1). On the other extreme, if an attacker captures both dvc and π (A4)—and thus is indistinguishable from the user—but does not compromise any admissible nodes, then it can forge only until all admissible nodes are disabled (G4). The "middle" cases are if the attacker compromises dvc and not π. If it compromises dvc and no then-authorized server is ever compromised (A2), then the attacker can do no better than an online dictionary attack against π (G2). If, on the other hand, when dvc is compromised some authorized server is eventually compromised (A3), then the attacker can do no better than an offline attack against π (G3).

4 Goals

Our high-level goal is to improve G2 and G4 from Section 3 (while keeping G1 and G3 unchanged). First

we motivate our improvements to G2. This property bounds the probability that an A2 attacker can forge signatures for the device, as a function of the total number q of password queries that the attacker can make to authorized servers after capturing the device. In a straightforward implementation, each server svr would individually limit the number of guesses to some number q_{svr}, and refuse to respond once svr has received q_{svr} queries from dvc with the wrong password. In this case, if A is the set of authorized servers when dvc is captured, then the number of queries that the attacker can make is $q = \sum_{\mathsf{svr} \in A} q_{\mathsf{svr}}$. Since servers are authorized dynamically, G2 provides little assurance without an additional mechanism to bound q, i.e., while q_{svr} is limited, q may not be. So, one goal is to regain the ability to limit q explicitly:

G2$^+$. An A2 attacker can forge signatures for dvc with probability at most $\frac{\hat{q}}{|\mathcal{D}|}$, where $\hat{q}$ is a prespecified constant and $\mathcal{D}$ is the dictionary from which the password is drawn.

Our second goal pertains to G4. As already noted, the number and identity of admissible nodes is not required to be fixed, and it seems most advantageous for it to be specified more fluidly (e.g., "all nodes certified by one of these three certification authorities"). Thus, disabling all admissible nodes, as required in G4, is a challenge. Even if the set of admissible nodes could be determined, disabling each of them may require interacting with potentially hundreds of far-flung nodes all over the world. Therefore, a second goal that we adopt here is to remedy this problem, by making one successful disable operation at nd* imply that dvc is disabled at *all* admissible nodes:

G4$^+$. An A4 attacker can forge signatures only until the time at which nd* is disabled for dvc.

5 Design

Our strategy for achieving G2$^+$ and G4$^+$ is to maintain a shared counter ctr for dvc that records the number of incorrect password guesses made globally against dvc. A server can access this counter using three operations: read, increment, and maximize; see Figure 2. Intuitively, read enables a server to read the current value of ctr, so that the server can refuse to interact with dvc if ctr $= \hat{q}$, thus enforcing G2$^+$. Upon an unsuccessful password guess from dvc a server will increment ctr. In addition, when nd* is disabled for dvc it invokes maximize to set ctr to $\hat{q}$, and then no server will respond to dvc, enforcing G4$^+$.

5.1 Mutually exclusive access

We strive to support concurrent requests for a counter from multiple servers to allow disabling a compromised device while the attacker is using it, and to permit maximum flexibility in legitimate uses of the device's private key (e.g., device cloning). To ensure the counter's consistency, our implementation enforces mutually exclusive access.

The protocol we propose for ensuring mutually exclusive access consists mainly of the svr_τ.initialize and svr_τ.retrieve functions shown in Figures 3 and 4. svr_τ.initialize is invoked by a node when τ is first submitted to it, and svr_τ.retrieve can be invoked either by svr_τ itself or remotely by another server. In a nutshell, each authorized capture protection server maintains a pointer—here called an *arrow* and denoted svr_τ.arrow—to the server from which it received the last *request* for access to the counter. That is, if svr_τ receives a request for the counter, then svr_τ requests it from svr′ ← svr_τ.arrow (line 4 in Figure 4) by invoking svr′.retrieve() (line 12 or 15). It also sets svr_τ.arrow to be the identity of the requester, denoted caller in Figure 4 (line 5). caller is authenticated by means that will be discussed in Section 5.2, so that if caller = svr″ and svr″ is not compromised, then svr″ performed this method invocation. Upon receiving the counter in response to the svr′.retrieve() request, svr returns the counter to caller (line 17). Figure 5 shows the effects of a retrieve request initiated by a server svr.

We emphasize that svr_τ.retrieve() may be invoked concurrently, e.g., by multiple remote servers. For simplicity, the psuedocode of Figure 4 and subsequent figures assumes that a thread of execution runs atomically (i.e., non-preemptively, without interference from other threads in svr_τ) until completion or until it blocks either on a semaphore[2] (line 7, 8 or 11) or due to invoking retrieve on another server (line 12 or 15). Once a running thread blocks, another can enter a retrieve operation. We denote global variables accessible to all threads using the "svr_τ." prefix, e.g., svr_τ.arrow. ("svr_τ", i.e., the identity of this server, is also global.) Variables without this prefix, specifically svr′ and caller, are local to this thread.

Use of two different semaphores requires some explanation. $\mathsf{svr}_\tau.\mathsf{sem}_1$ is used to ensure that once the counter is retrieved by svr_τ, requests to pull the counter away are blocked until svr_τ has executed its critical section (the lines marked "(∗)" in Figure 2). $\mathsf{svr}_\tau.\mathsf{sem}_2$ is

[2]To remind the reader, a semaphore s is a concurrency control primitive that represents a non-negative integer counter with two atomic operations: $V(s)$ increments s by one; $P(s)$ blocks the calling thread while $s = 0$ and then decrements s by one [8].

```
svr_τ.read()                    svr_τ.increment()               svr_τ.maximize()
    svr_τ.retrieve()                svr_τ.retrieve()                svr_τ.retrieve()
(*) c ← svr_τ.ctr           (*) svr_τ.ctr ← svr_τ.ctr + 1   (*) svr_τ.ctr ← q̂
    V(svr_τ.sem_1)                  V(svr_τ.sem_1)                  V(svr_τ.sem_1)
    return c
```

Figure 2. svr_τ.read, svr_τ.increment, **and** svr_τ.maximize **algorithms; can be invoked only locally**

```
1.  svr_τ.initialize()
2.      svr_τ.parent ← τ.getConsentingSvr()     /* extract consenting server from ticket */
3.      svr_τ.children ← ∅
4.      svr_τ.sem_2 ← 1                         /* so that it doesn't block to start with */
5.      if svr_τ.parent = 0                     /* τ = τ* */
6.          svr_τ.arrow ← svr_τ
7.          svr_τ.ctr ← 0                       /* initialize counter */
8.          svr_τ.sem_1 ← 1                     /* allow incoming retrieve requests */
9.      else
10.         svr_τ.arrow ← svr_τ.parent          /* initialize arrow to point to parent */
11.         svr_τ.sem_1 ← 0                     /* don't have the counter; block incoming retrieve requests */
```

Figure 3. svr_τ.initialize **algorithm; invoked locally by node hosting** svr_τ

used to block any retrieve requests made by svr_τ until svr_τ services the retrieve requests it received previously from others. Starvation is avoided if the retrieve requests blocked on each semaphore are serviced in a first-in-first out order per $V(svr_\tau.sem_i)$ invocation.

5.2 Limiting counter access

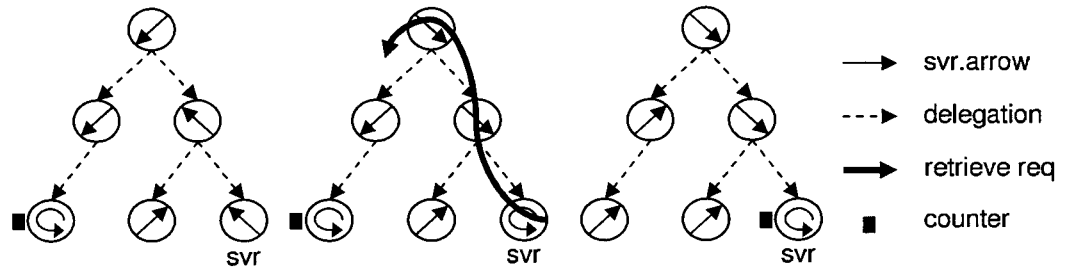
To achieve $G2^+$ and $G4^+$, it is necessary that only uncompromised servers can pass the counter once dvc is captured; otherwise a compromised server could manipulate the counter. For an A4 attacker, it would suffice to permit only admissible nodes to pass the counter, since admissible nodes are uncompromised by assumption. However, for an A2 attacker, where admissible nodes may be compromised, this simple rule does not suffice. Fortunately, since the servers authorized when an A2 attacker captures dvc are never compromised (by the definition of A2), it suffices to permit only authorized servers to pass or hold the counter. Because authorized servers are hosted only on admissible nodes, this is consistent with the A4 case.

Our protocol thus restricts counter passing to occur only between authorized servers in the A2 case. This is complicated by the fact that the set of authorized servers is dynamic, and there is no trustworthy record of this set. This problem can be partially alleviated by having a consenting server svr_τ record all the servers it has consented to authorize in a local set svr_τ.children (see line 2 of Figure 4). For simplicity, we portray svr_τ.children as a set of server names in our figures, though in reality a different representation is required. Specifically, because the ticket τ' resulting from a delegation to which svr_τ consented is not known to svr_τ, svr_τ cannot explicitly include τ' in svr_τ.children. However, if svr_τ includes a new cryptographic key k within both svr_τ.children and the inner ticket ζ' that it contributes as an input to the creation of τ', then svr_τ can use k to authenticate requests from $svr_{\tau'}$. For reasons described in Section 5.3, svr_τ also must send a preimage resistant and collision resistant hash of k, h_k, to dvc for storage in $authrec_{\tau'}$.

To ensure that the counter is passed only between authorized servers, it is also necessary for $svr_{\tau'} \in svr_\tau$.children to authenticate retrieve requests from svr_τ. Fortunately, $svr_{\tau'}$ can use the key k inserted into ζ' above (or another) to authenticate communication from svr_τ. To facilitate $svr_{\tau'}$ contacting svr_τ the first time, svr_τ's address is included within ζ' and τ'; $svr_{\tau'}$ assigns this address to $svr_{\tau'}$.parent (line 2 of Figure 3). (For τ_{svr^*}, a predetermined constant 0 is inserted in place of the consenting server address.)

```
1.  svr_τ.retrieve()
2.      if caller ∉ svr_τ.children ∪ {svr_τ.parent, svr_τ}   /* svr_τ.children, svr_τ.parent described in Sec. 5.2 */
3.          return ⊥                                          /* ignore requests from unknowns */
4.      svr' ← svr_τ.arrow                                    /* see who most recently requested the counter */
5.      svr_τ.arrow ← caller                                  /* record our caller as last requesting the counter */
6.      if (svr' = svr_τ)                                     /* if I most recently requested the counter, then ... */
7.          P(svr_τ.sem_2)                                    /* ... block request until I complete previous ones, */
8.          P(svr_τ.sem_1)                                    /* ... block caller until I'm done with the counter, */
9.          V(svr_τ.sem_2)                                    /* ... and permit requests to make progress again */
10.     else if (caller = svr_τ)                              /* if I am the one requesting, then ... */
11.         P(svr_τ.sem_2)                                    /* ... block request until I complete previous ones, */
12.         svr_τ.ctr ← svr'.retrieve()                       /* ... remote call to retrieve counter (blocks thread), */
13.         V(svr_τ.sem_2)                                    /* ... and permit requests to make progress again */
14.     else                                                  /* I am just a "transit server" for this request */
15.         svr_τ.ctr ← svr'.retrieve()
16.     if (caller ≠ svr_τ)
17.         return svr_τ.ctr                                  /* return counter to the remote caller */
```

Figure 4. svr_τ.retrieve **algorithm; can be invoked locally or remotely (by another server)**



(a) Before retrieve begins (b) Retrieve in transit (c) After retrieve returns

Figure 5. retrieve **initiated by** svr

5.3 Revocation

In Section 5.2, we described mechanisms by which servers keep track, in their children and parent variables, of other authorized servers (and how to authenticate them). Because of this, we must extend revocation to update these variables, so that servers do not work with outdated information about which servers are authorized. Whereas initially revocation was an operation local to dvc [15], here we extend it to include interaction with a server to update its children variable.

Specifically, before revoking a server $svr_{\tau'}$, dvc informs $svr_{\tau'}$.parent of this revocation. This notification indicates that dvc plans to revoke not just $svr_{\tau'}$ but also the servers in $svr_{\tau'}$.children, their children, and so forth. The purpose of dvc revoking the entire set of delegations derived from $svr_{\tau'}$ is to ensure that all still-authorized servers can continue to access the counter for dvc. Doing otherwise could partition the tree of delegations, and the counter may become inaccessible for some authorized servers. Note that svr* is never part of this revoked component. This is required since to achieve G4$^+$, svr* must be able to retrieve the counter.

During revocation of $svr_{\tau'}$, dvc informs svr_τ = $svr_{\tau'}$.parent by issuing a request req (with req.opType = `revoke`) to svr_τ. The revoked server $svr_{\tau'}$ is identified in req by h_k (hash of the key k) that svr_τ sent to dvc during the delegation protocol; see Section 5.2. This identifier for $svr_{\tau'}$ is extracted via req.getContents() (line 15 of Figure 6). This request induces a removal of $svr_{\tau'}$ (or rather k) from svr_τ.children. Also note that svr_τ retrieves the counter (see line 19 of Figure 6) thereby ensuring that the counter is not lost when $svr_{\tau'}$ is revoked. svr_τ retrieves the counter after removing

```
1.  svr_τ.doOperation(req)                          /* can be invoked remotely (by some dvc) */
2.      if (¬fromSameDvc(req, τ))
3.          return ⊥                                /* return if τ and req are not from same device */
4.      c ← svr_τ.read()
5.      if (c = q̂ ∨ c = ⊥)
6.          return ⊥                                /* return if counter at max or read failed */
7.      if (¬fromSameUser(req, τ))
8.          svr_τ.increment()                       /* record a bad password guess */
9.          return ⊥                                /* return on bad password guess */
10.     if (req.opType = "sign")
11.         return svr_τ.handleSignReq(req)         /* process sign request and return result */
12.     else if (req.opType = "delegate")
13.         return svr_τ.handleDelReq(req)          /* process delegation request and return result */
14.     else                                        /* req.opType = "revoke" */
15.         svr' ← req.getContents()                /* extract server to be revoked */
16.         if (svr' ∉ svr_τ.children)
17.             return ⊥                            /* return if svr' is not a child */
18.         svr_τ.children ← svr_τ.children \ {svr'}  /* remove svr' from the children set */
19.         svr_τ.retrieve()                        /* in case counter was pulled away after last retrieve */
20.         V(svr_τ.sem_1)
```

Figure 6. New svr_τ.doOperation algorithm

$svr_{\tau'}$ from svr_τ.children so that any subsequent requests from $svr_{\tau'}$ to retrieve the counter are rejected. After this svr_τ.doOperation call, dvc deletes $authrec_{\tau'}$, i.e., performs dvc.revoke($svr_{\tau'}$). Moreover, it must invoke dvc.revoke($svr_{\tau''}$) for each $svr_{\tau''} \in svr_{\tau'}$.children, their children, and so on.

5.4 Disabling

To achieve G4$^+$ we require that nd* can set the counter value to its maximum value $\hat{q}$ so as to disable dvc at all admissible nodes. The revocation mechanism presented in Section 5.3 ensures that svr* can always request the counter. Hence, upon receiving a disable request, nd* performs a svr*.maximize operation that causes servers to stop responding to dvc. The disable algorithm also uses a capability to authenticate the disable request; see [15].

Note that the nd.disable request can also be sent to another node nd hosting an authorized server for dvc. However, the ability of an A4 attacker to revoke and delegate makes it impractical to locate nodes besides nd* to disable after dvc has been compromised. Though the attacker can perform a dvc.revoke(svr*) operation, this will not restrict svr*'s access to the counter due to the measures described in Section 5.3. Hence, nd* is able to complete a disable request.

6 Implementation

We have developed the mechanisms described here within a cryptographic service provider implementing the standard service provider interfaces (SPIs) as specified by the Java Cryptography Architecture (JCA). Adhering to the SPIs allows our implementation to be plugged into a JCA-aware application at run time with minimal or no change to the application itself. These applications then gain the benefits of capture protection. Applications that are built with knowledge of capture protection can control delegation and revocation explicitly through additional interfaces.

6.1 Performance

Figure 7 shows the end-to-end latencies for dvc.sign, dvc.delegate, dvc.revoke (including the doOperation call at the parent of the server being revoked) and svr.disable operations. The horizontal axis represents the number of retrieves ("hops") through which the counter traveled to reach the server servicing dvc. These tests were performed on 1.4GHz, 256MB Linux machines running JVM 1.4.1 and connected by a 100Mbps local area network. In these tests, the device key and each server key was a 1024-bit RSA key [20]. Each data point is a mean of 20 recorded latencies.

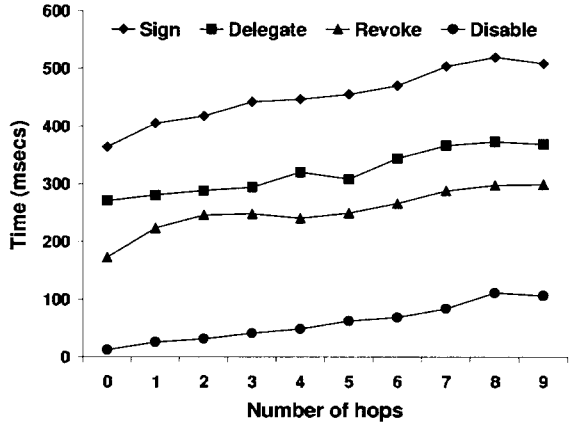


Figure 7. Capture protection performance

The costs of these operations at zero hops are inherited from the corresponding protocols in [15]. In the case of revoke and delegate operations, these are dominated by private key decryptions in fromSameDvc. The sign operation incurs the additional expense of computing the signature, which involves computation several times that of a standard RSA signature. The costs resulting from the mechanisms of the present paper are shown as the hops increase.

6.2 Transparency and proactive updates

Applications built without knowledge of capture protection do not get the benefits of delegation, as our present implementation delegates only by explicit application request, not automatically. However, our implementation can be configured to perform *proactive* key updates (c.f., [11]) for these applications. Proactivity is a mechanism by which cryptographic secrets are periodically updated so as to eliminate any advantage an adversary might have gained by the disclosure of a secret. In our setting, an attacker who learns sk_{nd^*} and who then captures the device constitutes an A3 attacker. A proactive update after disclosure of sk_{nd^*} but before dvc is captured, however, would limit this attacker to class A2. Such proactive updates can be achieved in a "delegation-like" protocol to replace server secrets including the public/private keypair. The proactive update results in a new sk_{nd^*} at nd^* and a new $<\tau^*, \mathsf{authrec}_{\tau^*}>$ at dvc, replacing the existing values holding old parameters for svr^* [15].

We have found that the existing JCA interfaces are an obstacle to implementing proactive updates transparently. To understand the challenges that JCA poses, we provide a brief description of two of the relevant JCA interfaces, specifically for the `KeyStore` class. A `KeyStore` object stores cryptographic keys—where in our case, each of these "keys" is a $<\tau^*, \mathsf{authrec}_{\tau^*}>$ pair for a private key, hereafter denoted a `CRKey`—and supplies interfaces to access these `CRKeys`. In addition, `KeyStore` supports interfaces for writing its contents to some (possibly remote) storage and for reading contents from storage to populate a `KeyStore` object. Specifically, the interface

```
KeyStore.store(OutputStream strm, char[] pwd)
```

stores the `KeyStore` state to `strm` and

```
KeyStore.load(InputStream strm, char[] pwd)
```

loads the state from `strm`. `pwd` is an optional argument that, if provided, is used as a key to compute (in `store`) or verify (in `load`) a message authentication code on the `KeyStore` contents. To aid G2$^+$, we omit this use of `pwd` in our capture-protected `KeyStore`, lest `pwd` bear a relation to a password π used with some `CRKey`.

The primary challenge to implementing proactivity for a `CRKey` stored in a `KeyStore` arises from an application's typical usage of the `KeyStore.store` and `KeyStore.load` interfaces. If an application makes no changes to the `KeyStore` after a `load` operation, then it is unlikely to save the `KeyStore` via a `store` operation. Unfortunately, in our efforts to implement proactive updates for a `CRKey`, our implementation necessarily changes the contained $<\tau^*, \mathsf{authrec}_{\tau^*}>$ pair without the application's knowledge. Without further measures, these changes will be lost when the application exits, rendering the `CRKey` thereafter unusable.

For this reason, ideally our `KeyStore` implementation would execute a `KeyStore.store` operation after each proactive update to any `CRKey` it contains, but herein lies a problem: Java provides no way to reliably convert the `InputStream` provided to `KeyStore.load` to an `OutputStream`, nor is there a way to determine the (possibly remote) location that this `InputStream` represents. Thus, there appears to be no means to implement proactivity transparently while faithfully storing `KeyStore` state to the `OutputStream` provided in the last `KeyStore.store` call.

Our present solution introduces a level of indirection in the storage of the `KeyStore` contents: `KeyStore` contents are not saved to the `OutputStream` provided to a `KeyStore.store` call, but rather somewhere else (we will elaborate below), and only a "pointer" to these contents is saved to the provided `OutputStream`. The `KeyStore` contents are encrypted and tagged with an authenticator before being stored, and the keys for decrypting and verifying the contents are saved to the

`OutputStream` along with the pointer. Naturally, during a `KeyStore.load` call, the provided `InputStream` is used to retrieve the pointer, which names the location from which the `KeyStore` contents (and the keys to decrypt and verify the contents) can be retrieved, and to which new `KeyStore` contents can be written as proactive updates occur.

The location for storing the `KeyStore` contents is a configurable parameter of our service provider. The best location is on dvc itself, but this may not be transparent to the application if `strm` is a connection to a remote machine (e.g., to enable a "clone" device to also reach the `KeyStore` contents, or because the remote machine is backed up but dvc is not). If `KeyStore` contents are stored off the device, then care must be taken to prevent their theft, since an attacker can nullify proactive updates if it can recover old `KeyStore` files for dvc. These older versions contain authorization records that should have been deleted within the proactive update protocol. So, if dvc is ever compromised and the key for decrypting `KeyStore` files is recovered, then old `KeyStore` files could be decrypted, yielding an A3 attacker if the attacker had also compromised the private key of nd* for the same time period.

7 Summary and future work

We presented the design and implementation of a capture protection infrastructure. Our innovation is a simple data-sharing protocol that strictly limits online dictionary attacks and achieves immediate disabling even with dynamically changing server populations. We also summarized our implementation of this infrastructure within a JCA service provider, and described challenges presented by the JCA interfaces for implementing proactive updates transparently. Future work includes availability of our capture protection services, which has not been the focus here.

References

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM T. Comput. Syst.* 9(1):1–20, Feb. 1991.

[2] B. Barak, A. Herzberg, D. Naor, E. Shai. The proactive security toolkit and applications. *Proc.* 6th *ACM Conf. Comput. & Commun. Secur.*, pages 18–27, 1999.

[3] D. Boneh, X. Ding, G. Tsudik, and M. Wong. A method for fast revocation of public key certificates and security capabilities. *Proc.* 10th *USENIX Secur. Symp.*, pages 297–308, Aug. 2001.

[4] R. Baldoni, A. Virgillito, R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. *Proc.* 7th *Int. Symp. Comput. & Commun.*, 2002.

[5] Y. Chang, M. Singhal and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. *Proc.* 9th *IEEE Symp. Reliable Distrib. Syst.*, pp. 146–154, 1990.

[6] Y. Chen and J. L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. *Proc.* 6th *Int. Workshop Disc. Algor. & Method. Mob. Comput. & Commun.*, Sep. 2002.

[7] M. J. Demmer and M. P. Herlihy. The Arrow distributed directory protocol. *Proc.* 12th *Int. Symp. Distrib. Comput.*, pp. 119–133, 1998.

[8] E. W. Dijkstra. Cooperating sequential processes. Mathematics Department, Technological University, Eindhoven, The Netherlands, 1965.

[9] X. Ding, D. Mazzocchi, and G. Tsudik. Experimenting with server-aided signatures. *Proc. 2002 Netw. & Distrib. Syst. Secur. Symp.*, Feb. 2002.

[10] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. *Proc. 1995 Netw. & Distrib. Syst. Secur. Symp.*, pp. 132–143, Feb. 1995.

[11] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. *Proc.* 4th *ACM Conf. Comput. & Commun. Secur.*, pp. 100–110, Apr. 1997.

[12] J. Knudsen. *Java Cryptography*, O'Reilly and Associates, May 1998.

[13] J. Linn. Generic security service application program interface. Internet RFC 2078, January 1997.

[14] L. Lamport, R. Shostak, M. Pease. The Byzantine generals problem. *ACM T. Progr. Lang. Sys.* 4(3):382–401, Jul. 1982.

[15] P. MacKenzie and M. K. Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distrib. Comput.*, 2003.

[16] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.

[17] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. *Proc.* 11th *Int. Conf. Distrib. Comput. Syst.*, pp. 354–360, 1991.

[18] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM T. Comput. Syst.* 7(1):61–77, Feb. 1989.

[19] R. L. N. Reddy, B. Gupta and P. K. Srimani. A new fault tolerant distributed mutual exclusion algorithm. *Proc. ACM Symp. Appl. Comput.*, pp. 831–839, 1992.

[20] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21(2):120–126, Feb. 1978.

[21] J. E. Walter, J. L. Welch and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Proc.* 2nd *Int. Work. Disc. Algor. & Method. Mob. Comput. & Commun.*, Oct. 1998.