

# Delegation of Cryptographic Servers for Capture-Resilient Devices

(Extended Abstract)

Philip MacKenzie  
Bell Labs, Lucent Technologies  
[philmac@research.bell-labs.com](mailto:philmac@research.bell-labs.com)

Michael K. Reiter  
Bell Labs, Lucent Technologies  
[reiter@research.bell-labs.com](mailto:reiter@research.bell-labs.com)

## ABSTRACT

A device that performs private key operations (signatures or decryptions), and whose private key operations are protected by a password, can be immunized against offline dictionary attacks in case of capture by forcing the device to confirm a password guess with a designated remote server in order to perform a private key operation. Recent proposals for achieving this allow untrusted servers and require no server initialization per device. In this paper we extend these proposals to enable dynamic delegation from one server to another; i.e., the device can subsequently use the second server to secure its private key operations. One application is to allow a user who is traveling to a foreign country to temporarily delegate to a server local to that country the ability to confirm password guesses and aid the user's device in performing private key operations, or in the limit, to temporarily delegate this ability to a token in the user's possession. Another application is proactive security for the device's private key, i.e., proactive updates to the device and servers to eliminate any threat of offline password guessing attacks due to previously compromised servers.

## 1. INTRODUCTION

A device that performs private key operations (signatures or decryptions) risks exposure of its private key if captured. While encrypting the private key with a password is common, this provides only marginal protection, since passwords are well-known to be susceptible to offline dictionary attacks (e.g., [14, 12]). Much recent research has explored better password protections for the private keys on a device that may be captured. These include techniques (i) to encrypt the private key under a password in a way that prevents the attacker from verifying a successful password guess (*cryptographic camouflage*) [11]; or (ii) to force the attacker to verify his password guesses at an online server, thereby turning on offline attack into an online one that can be detected and stopped (e.g., [8, 13]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'01, November 5-8, 2001, Philadelphia, Pennsylvania, USA.  
Copyright 2001 ACM 1-58113-385-5/01/0011 ...\$5.00.

We take as our starting point the latter approach, in which an attacker who captures that device must validate its password guesses at a remote server before the use of the private key is enabled. In particular, we focus on the proposals of [13], in which this server is untrusted—its compromise does not reduce the security of the device's private key unless the device is also captured—and need not have a prior relationship with the device. This approach offers certain advantages: e.g., it is compatible with existing infrastructure, whereas cryptographic camouflage requires that “public” keys be hidden from potential attackers. However, it also comes with the disadvantage that the device must interact with a designated remote server in order to perform a (and typically each) private key operation. This interaction may become a bottleneck if the designated remote server is geographically distant and the rate of private key operations is significant.

In this paper, we investigate a technique to alleviate this limitation, with which a device may temporarily delegate the password-checking function from its originally designated server to another server that is closer to it. For example, a business traveler in a foreign country may temporarily delegate the password-checking function for her laptop computer to a server in the country she is visiting. By doing so, her device's subsequent private key operations will require interaction only with this local server, presumably incurring far less latency than if the device were interacting with the original server. In the limit, the user could temporarily delegate to a hardware token in her possession, so that the device could produce signatures or decryptions in offline mode without network access at all.

Of course, delegating the password-checking function from one server to another has security implications. As originally developed, the techniques that serve as our starting point [13] have the useful property that the designated server, in isolation, gains no information that would enable it to forge signatures or decrypt ciphertexts on the device's behalf. However, if both it and the device were captured, then the attacker could mount an offline dictionary attack against the password, and then forge signatures or decrypt ciphertexts for the device if he succeeds. Naturally, in the case of delegation, this vulnerability should not extend to any server *ever* delegated by the device. Rather, our high-level security goal is to ensure that an individual server authorized for password-checking by delegation, and whose authority is then revoked, poses the same security threat as a server to which delegation never occurred in the first place.

Specifically, an attacker that captures the device after the device has revoked the authorization of a server (even if the server was previously compromised) must still conduct an *online* dictionary attack at an authorized server in order to attack the password.

Even with this goal achieved, delegation does impinge on security in at least two ways, however. First, 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. Second, a feature of the original protocols is that the password-checking server could be permanently *disabled* for the device even after the device *and* password were compromised; by doing so, the device can never sign or decrypt again. In a system supporting delegation, however, 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 will never sign or decrypt again, every server in this permissible set must be disabled for the device.

As a side effect of achieving our security goals, our techniques offer a means for realizing *proactive security* (e.g., [10]) in the context of [13]. Intuitively, proactive security encompasses techniques for periodically refreshing the cryptographic secrets held by various components of a system, thereby rendering any cryptographic secrets captured before the refresh useless to the attacker. We show how our delegation protocol can be used as a subroutine for proactively refreshing a password-checking server, so that if the server's secrets had been exposed, they are useless to the attacker after the refresh. In particular, if the attacker subsequently captured the device, any dictionary attack that the attacker could mount would be *online*, as opposed to offline.

In this extended abstract we specify security requirements for delegation in this context and then describe a delegation system for RSA signing [17]. (The ElGamal decryption system described in [13] can also be revised to support delegation, though we defer this to the full paper due to space limitations.) Supporting delegation for RSA signing not only requires devising a custom delegation protocol for RSA keys, but also modifying the original signing protocol [13] to accommodate delegation. For example, our revised RSA system utilizes three-way function sharing, versus the two-way function sharing used in the original system; this seems to be required to accomplish our objectives. And, whereas the original systems of [13] permitted the server to conduct an offline dictionary attack against the user's password (without placing the device's signing key at risk), here we must prevent a server from conducting such an attack. Our delegation protocol itself also contributes points of technical interest, as we will discuss later.

## 2. PRELIMINARIES

In this section we state the goals for our systems. We also introduce preliminary definitions and notation that will be necessary for the balance of the paper.

### 2.1 System model

Our system consists of a device *dvc* and an arbitrary, possibly unknown, number of servers. A server will be denoted by *svr*, possibly with subscripts or other annotations when

useful. The device communicates to a server over a public network. In our system, the device is used either for generating signatures or decrypting messages, and does so by interacting with one of the servers. The signature or decryption operation is password-protected, by a password  $\pi_0$ . The system is initialized with public data, secret data for the device, secret data for the user of the device (i.e.,  $\pi_0$ ), and secret data for each of the servers. The public and secret data associated with a server should simply be a certified public key and associated private key for the server, which most likely would be set up well before the device is initialized.

The device-server protocol allows a device operated by a legitimate user (i.e., one who knows  $\pi_0$ ) to sign or decrypt a message with respect to the public key of the device, after communicating with one of the servers. This server must be *authorized* to execute this protocol. (We define *authorized* precisely below.) The system is initialized with exactly one server authorized, denoted  $svr_0$ . Further servers may be authorized, but this authorization cannot be performed by *dvc* alone. Rather, for *dvc* to authorize *svr*, another already-authorized server  $svr'$  must also consent to the authorization of *svr* after verifying that the authorization of *svr* is consistent with policy previously set forth by *dvc* and is being performed by *dvc* with the user's password. In this way, authorization is a protected operation just as signing is. The device can unilaterally *revoke* the authorization of a server when it no longer intends to use that server. A server can be *disabled* (for a device) by being instructed to no longer respond to that device or, more precisely, to requests involving its key.

For the purposes of this paper, the aforementioned policy dictating which servers can be authorized is expressed as a set  $U$  of servers with well-known public keys. That is, an authorized server *svr* will consent to authorize another server  $svr'$  only if  $svr' \in U$ . Moreover, we assume that *svr* can reliably determine the unique public key  $pk_{svr'}$  of any  $svr' \in U$ . In practice, this policy would generally need to be expressed more flexibly; for example, a practical policy might allow any server with a public key certified by a given certification authority to be authorized. For such a policy, our delegation protocols would then need to be augmented with the appropriate certificates and certificate checks; for simplicity, we omit such details here.

To specify security for our system, we must consider the possible attackers that attack the system. Each attacker we consider in this paper is presumed to control the network; i.e., the attacker controls the inputs to the device and every server, and observes the outputs. Moreover, an attacker can permanently *compromise* certain resources. The possible resources that may be compromised by the attacker are any of the servers, *dvc*, and  $\pi_0$ . Compromising reveals the entire contents of the resource to the attacker. 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 with  $\pi_0$  as input—and that  $\pi_0$  is not subsequently input to the device by the user. This decouples the capture of *dvc* and  $\pi_0$ , 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 the aspects of the system described thus far as a collection of *events*.

1.  $\text{dvc.startDel}(svr, svr')$ : dvc begins a delegation protocol with server  $svr$  to authorize  $svr'$ .
2.  $\text{dvc.finishDel}(svr, svr')$ : dvc finishes a delegation protocol with server  $svr$  to authorize  $svr'$ . This can occur only after a  $\text{dvc.startDel}(svr, svr')$  with no intervening  $\text{dvc.finishDel}(svr, svr')$ ,  $\text{dvc.revoke}(svr)$  or  $\text{dvc.revoke}(svr')$ .
3.  $\text{dvc.revoke}(svr)$ : dvc revokes the authorization of  $svr$ .
4.  $svr.disable$ :  $svr$  stops responding to any requests of the device (signing, decryption, or delegation).
5.  $\text{dvc.comp}$ : dvc is compromised (and captured).
6.  $svr.comp$ :  $svr$  is compromised.
7.  $\pi_0.comp$ : the password  $\pi_0$  is compromised.

The time of any event  $x$  is given by  $T(x)$ . Now we define the following predicates for any time  $t$ :

- $\text{authorized}_t(svr)$  is true iff either (i)  $svr = svr_0$  and there is no  $\text{dvc.revoke}(svr_0)$  prior to time  $t$ , or (ii) there exist a  $svr'$  and event  $x = \text{dvc.finishDel}(svr', svr)$  where  $\text{authorized}_{T(x)}(svr')$  is true,  $T(x) < t$ , and no  $\text{dvc.revoke}(svr)$  occurs between  $T(x)$  and  $t$ . In case (ii), we call  $svr'$  the *consenting server*.
- $\text{nominated}_t(svr)$  is true iff there exist a  $svr'$  and event  $x = \text{dvc.startDel}(svr', svr)$  where  $\text{authorized}_{T(x)}(svr')$  is true,  $T(x) < t$ , and none of  $\text{dvc.finishDel}(svr', svr)$ ,  $\text{dvc.revoke}(svr)$ , or  $\text{dvc.revoke}(svr')$  occur between  $T(x)$  and  $t$ .
- $\text{tainted}_t(svr)$  is true iff  $\text{authorized}_t(svr)$  is true and the consenting server  $svr'$  was compromised before the most recent  $\text{dvc.finishDel}(svr', svr)$ .

For any event  $x$ , let

$$\begin{aligned} \text{Active}(x) &= \{ svr : \text{nominated}_{T(x)}(svr) \vee \\ &\quad \text{authorized}_{T(x)}(svr) \} \\ \text{Tainted}(x) &= \{ svr : \text{tainted}_{T(x)}(svr) \} \end{aligned}$$

Note that  $\text{Tainted}(x) \subseteq \text{Active}(x)$ .

## 2.2 Goals

It is convenient in specifying our security goals to partition attackers into four classes, depending on the resources they compromise and the state of executions when these attackers compromise certain resources. An attacker is assumed to fall into one of these classes independent of the execution, i.e., it does not change its behavior relative to these classes depending on the execution of the system. In particular, the resources an attacker compromises are assumed to be independent of the execution. In this sense, we consider static attackers only (in contrast to adaptive ones).

- A1. An attacker in class A1 does not compromise dvc or compromises dvc only if  $\text{Active}(\text{dvc.comp}) = \emptyset$ .
- A2. An attacker in class A2 is not in class A1, does not compromise  $\pi_0$ , and compromises dvc only if  $\text{Tainted}(\text{dvc.comp}) = \emptyset$  and  $svr.comp$  never occurs for any  $svr \in \text{Active}(\text{dvc.comp})$ .
- A3. An attacker in class A3 is not in class A1, does not compromise  $\pi_0$ , and compromises dvc only if  $\text{Tainted}(\text{dvc.comp}) \neq \emptyset$  or  $svr.comp$  occurs for some  $svr \in \text{Active}(\text{dvc.comp})$ .

- A4. An attacker in class A4 is in none of classes A1, A2, or A3, and does not compromise any  $svr \in U$ .

Now we state the security goals of our systems against these attackers as follows (disregarding negligible probabilities):

- G1. An A1 attacker is unable to forge signatures or decrypt messages for dvc.
- G2. An A2 attacker can forge signatures or decrypt messages for the device with probability at most  $\frac{q}{|D|}$ , where  $q$  is the total number of queries to servers in  $\text{Active}(\text{dvc.comp})$  after  $T(\text{dvc.comp})$ , and  $D$  is the dictionary from which the password is drawn (at random).
- G3. An A3 attacker can forge signatures or decrypt messages for the device only if it succeeds in an offline dictionary attack on the password.
- G4. An A4 attacker can forge signatures or decrypt messages only until  $\max_{svr \in U} \{T(svr.disable)\}$ .

These goals can be more intuitively stated as follows. First, if an attacker does not capture dvc, or does so only when no servers are authorized for dvc (A1), then the attacker gains no ability to forge or decrypt for the device (G1). On the other extreme, if an attacker captures both dvc and  $\pi_0$  (A4)—and thus is indistinguishable from the user—it can forge only until all servers are disabled (G4) or indefinitely if it also compromises a server. The “middle” cases are if the attacker compromises dvc and not  $\pi_0$ . If when it compromises dvc, no authorized server is tainted (i.e., authorized with the help of a compromised server) or ever compromised (A2), then the attacker can do no better than an online dictionary attack against  $\pi_0$  (G2). If, on the other hand, when dvc is compromised some authorized server is tainted or eventually compromised (A3), then the attacker can do no better than an offline attack against the password (G3). It is not difficult to verify that these goals are a strict generalization of the goals of [13]; i.e., these goals reduce to those of [13] in the case  $|U| = 1$ .

It is important to notice that though goal G3 permits attackers in class A3 to conduct an offline dictionary attack, this class of attacker can be constrained in practice in a large number of circumstances. That is, one way of using a system satisfying goals G1–G4 is to apply the delegation protocol twice successively: once to authorize a server  $svr$ , and once using  $svr$  to authorize *itself* a second time. If  $svr$  is not compromised, then it will not be tainted in its second delegation (from and to itself)—even if the attacker had previously compromised the server  $svr'$  that consented to authorize  $svr$  in the first place. So, if dvc revokes  $svr'$ , then by property G2, the attacker would be forced to conduct an *online* attack to forge or decrypt for the device, even if it captures the device.

This observation suggests an approach to proactively update dvc to render useless to an attacker any information it gained by compromising a server. That is, suppose that each physical computer running a logical server  $svr$  periodically instantiates a new logical server  $svr'$  having a new public and private key. If dvc first delegates from  $svr$  to  $svr'$  and then from  $svr'$  to  $svr'$ , and if then dvc revokes  $svr$ , any disclosure of information from  $svr$  (e.g., the private key of  $svr$ ) is then useless for the attacker in its efforts to forge or decrypt for dvc. Rather, if the attacker captures dvc, it must

compromise  $svr'$  in order to conduct an offline attack against  $dvc$ .

### 2.3 Tools

Our systems for meeting the goals outlined in Section 2.2 utilize a variety of cryptographic tools, which we define informally below.

**Security parameters** Let  $\kappa$  be the main cryptographic security parameter; a reasonable value today may be  $\kappa = 160$ . We will use  $\lambda > \kappa$  as a secondary security parameter for public keys. For instance, in an RSA public key scheme may we may set  $\lambda = 1024$  to indicate that we use 1024-bit moduli.

**Hash functions** We use  $h$ , with an additional subscript as needed, to denote a hash function. Unless otherwise stated, the range of a hash function is  $\{0, 1\}^\kappa$ . We do not specify here the exact security properties (e.g., one-wayness, collision resistance, or pseudorandomness) we will need for the hash functions (or keyed hash functions, below) that we use. To formally prove that our systems meet every goal outlined above, we generally require that these hash functions behave like random oracles [2]. (For heuristics on instantiating random oracles, see [2].) However, for certain subsets of goals, weaker properties may suffice; details will be given in the individual cases.

**Keyed hash functions** A keyed hash function family is a family of hash functions  $\{f_v\}$  parameterized by a secret value  $v$ . We will typically write  $f_v(m)$  as  $f(v, m)$ , as this will be convenient in our proofs. In this paper we employ various keyed hash functions with different ranges, which we will specify when not clear from context. We will also use a specific type of keyed hash function, a message authentication code (MAC). We denote a MAC family as  $\{\text{mac}_a\}$ . In this paper we do not require MACs to behave like random oracles.

**Encryption schemes** An *encryption scheme*  $\mathcal{E}$  is a triple  $(G_{enc}, E, D)$  of algorithms, the first two being probabilistic, and all running in expected polynomial time.  $G_{enc}$  takes as input  $1^\lambda$  and outputs a public key pair  $(pk, sk)$ , i.e.,  $(pk, sk) \leftarrow G_{enc}(1^\lambda)$ .  $E$  takes a public key  $pk$  and a message  $m$  as input and outputs an encryption  $c$  for  $m$ ; we denote this  $c \leftarrow E_{pk}(m)$ .  $D$  takes a ciphertext  $c$  and a secret key  $sk$  as input and returns either a message  $m$  such that  $c$  is a valid encryption of  $m$  under the corresponding public key, if such an  $m$  exists, and otherwise returns  $\perp$ . Our systems require an encryption scheme secure against adaptive chosen ciphertext attacks [16]. Practical examples can be found in [3, 6].

**Signature schemes** A *digital signature scheme*  $\mathcal{S}$  is a triple  $(G_{sig}, S, V)$  of algorithms, the first two being probabilistic, and all running in expected polynomial time.  $G_{sig}$  takes as input  $1^\lambda$  and outputs a public key pair  $(pk, sk)$ , i.e.,  $(pk, sk) \leftarrow G_{sig}(1^\lambda)$ .  $S$  takes a message  $m$  and a secret key  $sk$  as input and outputs a signature  $\sigma$  for  $m$ , i.e.,  $\sigma \leftarrow S_{sk}(m)$ .  $V$  takes a message  $m$ , a public key  $pk$ , and a candidate signature  $\sigma'$  for  $m$  as input and returns the bit  $b = 1$  if  $\sigma'$  is a valid signature for  $m$  for the corresponding private key, and otherwise returns the bit  $b = 0$ . That is,  $b \leftarrow V_{pk}(m, \sigma')$ . Naturally, if  $\sigma \leftarrow S_{sk}(m)$ , then  $V_{pk}(m, \sigma) = 1$ .

## 3. DELEGATION FOR S-RSA

The work on which this paper is based [13] described sev-

eral systems by which  $dvc$  could involve a server for performing the password-checking function and assisting in its cryptographic operations, and thereby gain immunity to offline dictionary attacks if captured. The first of these systems, denoted **GENERIC**, did not support the disabling property (the instantiation of G4 for a single server and no delegation), but worked for any type of public key algorithm that  $dvc$  used. As part of the signing/decryption protocol in this system,  $dvc$  recovered the private key corresponding to its public key. This, in turn, renders delegation in this system straightforward, being roughly equivalent to a re-initialization of the device using the same private key, but for a different server. The few minor technical changes needed to accommodate delegation are also reflected in the RSA system we detail here, and so we omit further discussion of **GENERIC** due to space limitations.

The system described in [13] by which  $dvc$  performs RSA signatures is called **S-RSA**. At a high level, **S-RSA** uses 2-out-of-2 function sharing to distribute the ability to generate a signature for the device's public key between the device and the server. The server, however, would cooperate with the device to sign a message only after being presented with evidence that the device was in possession of the user's correct password.

In this section we describe a new system for RSA signatures, called **S-RSA-DEL**, that supports delegation in addition to signatures. In order to accommodate delegation in this context, the system is changed so that the device's signature function is shared using a 3-out-of-3 function sharing, where one of the three shares is generated from the password itself. In this way, the user share (i.e., the password) may remain the same while the device share is changed for delegation purposes. Other changes are needed as well; e.g., whereas the server in the **S-RSA** system could mount an offline dictionary attack against the user's password (without risk to the device's signature operations), here we must prevent the server from mounting such an attack. While introducing these changes to the signing protocol, and introducing the new delegation protocol, we strive to maintain the general protocol structure of **S-RSA**.

### 3.1 Preliminaries

We suppose the device creates signatures using a standard encode-then-sign RSA signature algorithm (e.g., “hash-and-sign” [7]). The public and secret keys of the device are  $pk_{dvc} = \langle e, N \rangle$  and  $sk_{dvc} = \langle d, N, \phi(N) \rangle$ , respectively, where  $ed \equiv_{\phi(N)} 1$ ,  $N$  is the product of two large prime numbers and  $\phi$  is the Euler totient function. (The notation  $\equiv_{\phi(N)}$  means equivalence modulo  $\phi(N)$ .) The device's signature on a message  $m$  is defined as follows, where  $\text{encode}$  is the encoding function associated with  $S$ , and  $\kappa_{sig}$  denotes the number of random bits used in the encoding function (e.g.,  $\kappa_{sig} = 0$  for a deterministic encoding function):

$$S_{\langle d, N, \phi(N) \rangle}(m): \begin{array}{l} r \leftarrow_R \{0, 1\}^{\kappa_{sig}} \\ \sigma \leftarrow (\text{encode}(m, r))^d \bmod N \\ \text{return } \langle \sigma, r \rangle \end{array}$$

Note that it may not be necessary to return  $r$  if it can be determined from  $m$  and  $\sigma$ . We remark that “hash-and-sign” is an example of this type of signature in which the encoding function is simply a (deterministic) hash of  $m$ , and that PSS [4] is another example of this type of signature with a probabilistic encoding. Both of these types of sig-

natures were proven secure against adaptive chosen message attacks in the random oracle model [2, 4]. Naturally any signature of this form can be verified by checking that  $\sigma^e \equiv_N \text{encode}(m, r)$ . In the function sharing primitive used in our system,  $d$  is broken into shares  $d_0$ ,  $d_1$  and  $d_2$  such that  $d_0 + d_1 + d_2 \equiv_{\phi(N)} d$  [5].

### 3.2 Device initialization

The inputs to device initialization are the identity of  $\text{svr}_0$  and its public encryption key  $pk_{\text{svr}_0}$ , the user's password  $\pi_0$ , the device's public key  $pk_{\text{dvc}} = \langle e, N \rangle$ , and the corresponding private key  $sk_{\text{dvc}} = \langle d, N, \phi(N) \rangle$ . The initialization algorithm proceeds as follows:

```

 $d_0 \leftarrow h(\pi_0)$ 
 $t \leftarrow_R \{0, 1\}^\kappa$ 
 $u \leftarrow h_{\text{dsbl}}(t)$ 
 $v \leftarrow_R \{0, 1\}^\kappa$ 
 $a \leftarrow_R \{0, 1\}^\kappa$ 
 $b \leftarrow f(v, \pi_0)$ 
 $d_1 \leftarrow_R \{0, 1\}^{\lambda+\kappa}$ 
 $d_2 \leftarrow d - d_1 - d_0 \bmod \phi(N)$ 
 $\tau \leftarrow E_{pk_{\text{svr}_0}}(\langle a, b, u, d_2, N \rangle)$ 

```

$h$  is assumed to output a  $(\lambda + \kappa)$ -bit value. The value  $pk_{\text{dvc}}$  and *authorization record*  $\langle \text{svr}_0, pk_{\text{svr}_0}, \tau, t, v, d_1, a \rangle$  are saved on stable storage in the device. All other values, including  $d$ ,  $\phi(N)$ ,  $\pi_0$ ,  $b$ ,  $u$ ,  $d_0$ , and  $d_2$ , are deleted from the device. The value  $t$  should be backed up offline for use in disabling if the need arises. The  $\tau$  value is the device's "ticket" that it uses to access  $\text{svr}_0$ . The  $u$  value is the "ticket identifier".

The ticket  $\tau$  will be sent to  $\text{svr}$  within the context of the S-RSA-DEL signing and delegation protocols (see Sections 3.3 and 3.4), and the server will inspect the contents of the ticket to extract its share  $d_2$  of the device's private signing key. In anticipation of its own compromise,  $\text{dvc}$  might include a policy statement within  $\tau$  to instruct  $\text{svr}_0$  as to what it should or should not do with requests bearing this ticket. This policy could include an intended expiration time for  $\tau$ , instructions to cooperate in signing messages only of a certain form, or instructions to cooperate in delegating only to certain servers. As discussed in Section 2.1, here we assume a default policy that restricts delegation to only servers in  $U$ . For simplicity, we omit this policy and its inspection from device initialization and subsequent protocols, but a practical implementation must support it.

### 3.3 Signature protocol

Here we present the protocol by which the device signs a message  $m$ . The input provided to the device for this protocol is the input password  $\pi$ , the message  $m$ , and the identity  $\text{svr}$  of the server to be used, such that  $\text{dvc}$  holds an authorization record  $\langle \text{svr}, pk_{\text{svr}}, \tau, t, v, d_1, a \rangle$ , generated either in the initialization procedure of Section 3.2, or in the delegation protocol of Section 3.4. Recall that  $\text{dvc}$  also stores  $pk_{\text{dvc}} = \langle e, N \rangle$ . In this protocol, and all following protocols, we do not explicitly check that message parameters are of the correct form and fall within the appropriate bounds, but any implementation must do this. The protocol is described in Figure 1.

The means by which this protocol generates a signature for  $m$  is to construct  $\text{encode}(m, r)^{d_0+d_1+d_2}$ , where  $d_0$  is derived from the user's password,  $d_1$  is stored on  $\text{dvc}$ , and  $d_2$  is stored in  $\tau$ .  $\nu = \text{encode}(m, r)^{d_2} \bmod N$  is computed

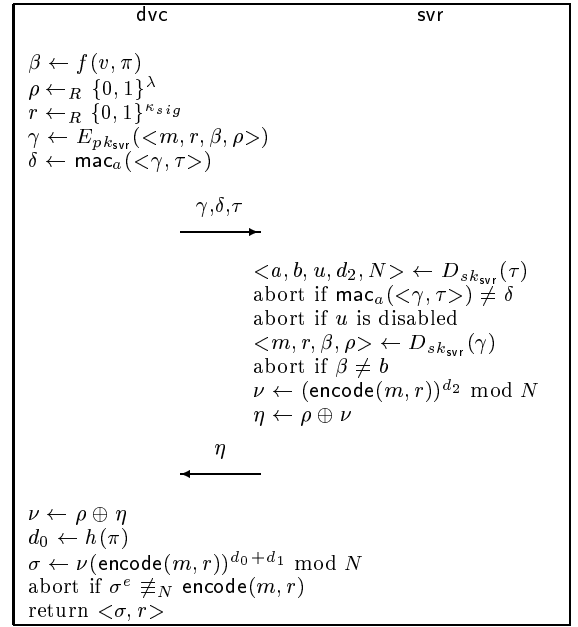


Figure 1: S-RSA-DEL signature protocol

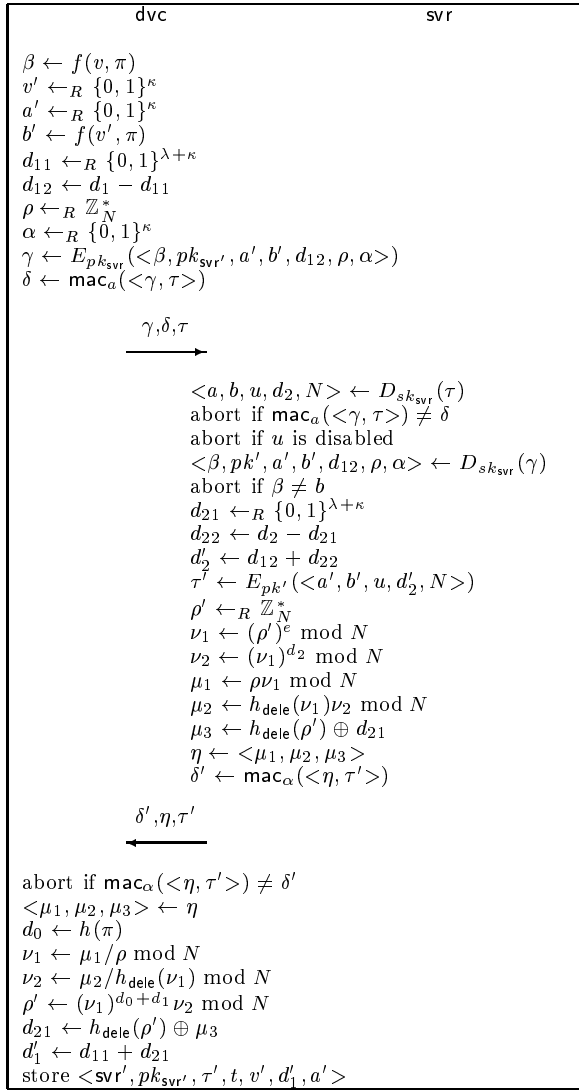
at  $\text{svr}$  after  $\text{svr}$  has confirmed that  $\beta$  is valid evidence that  $\text{dvc}$  holds the user's password. The device multiplies  $\nu$  by  $\text{encode}(m, r)^{d_0+d_1} \bmod N$  to get the desired result. It is important that the device delete  $\beta$ ,  $d_0$  and  $\rho$  (used to encrypt  $\nu$ ) when the protocol completes, and that it never store them on stable storage.

$\delta$  is a message authentication code computed using  $a$ , to show the server that this request originated from the device.  $\delta$  enables  $\text{svr}$  to distinguish an incorrect password guess by someone holding the device from a request created by someone not holding the device. Since  $\text{svr}$  should respond to only a limited number of the former (lest it allow an online dictionary attack to progress too far),  $\delta$  is important in preventing denial-of-service attacks against the device by an attacker who has not compromised the device.

### 3.4 Delegation protocol

Here we present the protocol by which the device delegates the capability to help it perform cryptographic operations to a new server (or simply generates new data for the same server). The inputs provided to the device are the identity  $\text{svr}$  of the server to be used, such that  $\text{dvc}$  holds an authorization record  $\langle \text{svr}, pk_{\text{svr}}, \tau, t, v, d_1, a \rangle$ , a public key  $pk_{\text{svr}'}$  for another server  $\text{svr}' \in U$ , and the input password  $\pi$ . (As described in Section 3.2, one could also input additional policy information here.) Recall that  $\text{dvc}$  also stores  $pk_{\text{dvc}} = \langle e, N \rangle$ . The protocol is described in Figure 2. In this figure,  $h_{\text{dele}}$  is assumed to output a  $(\lambda + \kappa)$ -bit value.

The overall goal of the protocol in Figure 2 is to generate a new share  $d'_2$  for server  $\text{svr}'$ , and new share  $d'_1$  and new ticket  $\tau'$  for the device to use with  $\text{svr}'$ . The device's new share  $d'_1$  is created as the sum of  $d_{11}$  and  $d_{21}$ , selected randomly by  $\text{dvc}$  and  $\text{svr}$ , respectively. The new share  $d'_2$  for  $\text{svr}'$  is constructed as  $d'_2 = d_{12} + d_{22} = (d_1 - d_{11}) + (d_2 - d_{21})$ , with the first and second terms being computed by  $\text{dvc}$  and  $\text{svr}$ , respectively. As a result,  $d'_1 + d'_2 = d_1 + d_2$ . Note that  $\text{svr}$



**Figure 2:** S-RSA-DEL delegation protocol

learns  $d'_2$  and in fact creates  $\tau'$  with it. It is for this reason that we define  $svr'$  to be tainted if  $svr$  was compromised before this protocol is executed (see Section 2.2).

In addition to the manipulation of these shares of  $d$ , this protocol borrows many components from the signature protocol of Figure 1. For example,  $\beta$ ,  $\gamma$  and  $\delta$  all play similar roles in the protocol as they did in Figure 1. And deletion is once again important:  $dvc$  must delete  $\beta$ ,  $b'$ ,  $d_{11}$ ,  $d_{21}$ ,  $\rho$  and all other intermediate computations at the completion of this protocol. Similarly,  $svr$  should delete  $\beta$ ,  $b$ ,  $b'$ ,  $d_{12}$ ,  $d_{21}$ ,  $d_{22}$ ,  $d'_2$ ,  $\rho$  and all other intermediate results when it completes.

A point of interest in the protocol of Figure 2 is the construction of  $\eta = \langle \mu_1, \mu_2, \mu_3 \rangle$ , which is sent back to  $dvc$ .  $\eta$  is an encryption of the value  $d_{21}$  to transport it securely to  $dvc$ . This encryption is public-key-like, in that an attacker who subsequently compromises  $svr$  will be unable to determine  $d_{21}$  from the message  $\delta', \eta, \tau'$ . If, in contrast,  $d_{21}$  were transported back to  $dvc$  encrypted only symmetrically (e.g., using  $\alpha$ ), then the compromise of  $svr$  would reveal  $\alpha$  and

then  $d_{21}$ . It is then not difficult to verify that G2 could be violated.

To relate this protocol to the system model of Section 2.1, and for our proofs in Section 4, we define the execution of the code before the first message in Figure 2 to constitute a  $dvc.\text{startDel}(svr, svr')$  event. Likewise, we define the execution of the code after the second message in Figure 2 to constitute a  $dvc.\text{finishDel}(svr, svr')$  event. The event  $dvc.\text{revoke}(svr)$ , though not pictured in Figure 2, can simply be defined as  $dvc$  deleting any authorization record  $\langle svr, pk_{svr}, \tau, t, v, d_1, a \rangle$  and halting any ongoing delegation protocols to authorize  $svr$ .

### 3.5 Key disabling

As in [13], the S-RSA-DEL system supports the ability to *disable* the device's key at servers, as would be appropriate to do if the device were stolen. Provided that the user backed up  $t$  before the device was stolen, the user can send  $t$  to a server. The server can then store  $u = h_{\text{dsbl}}(t)$  on a list of disabled ticket identifiers. Subsequently, the server should refuse to respond to any request containing a ticket  $\tau$  with a ticket identifier  $u$ . Rather than storing  $u$  forever, the server can discard  $u$  once there is no danger that  $pk_{dvc}$  will be used subsequently (e.g., once the public key has been revoked). Note that for security against denial-of-service attacks (an attacker attempting to disable  $u$  without  $t$ ), we do not need  $h_{\text{dsbl}}$  to be a random oracle, but simply a one-way hash function.

In relation to the model of Section 2.1,  $svr.\text{disable}$  denotes the event in which  $svr$  receives  $t$  and marks  $u = h_{\text{dsbl}}(t)$  as disabled. For convenience, we say that a ticket  $\tau$  is disabled at  $svr$  if  $\tau$  contains  $u$  as its ticket identifier and  $u$  is marked as disabled at  $svr$ .

## 4. SECURITY FOR S-RSA-DEL

In this section we provide a formal proof of security for the S-RSA-DEL system in the random oracle model. We begin, however, with some intuition for the goals G1–G4 in light of the protocols of Figures 1 and 2.

- An A1 attacker never obtains an authorization record  $\langle svr, pk_{svr}, \tau, t, v, d_1, a \rangle$  from  $dvc$ , either because it never compromises  $dvc$  or because  $dvc$  has deleted all such records by the time it is compromised. Without any  $d_1$  used with any  $svr$ , the attacker has no ability to forge a signature for  $dvc$  (even if it knows  $\pi_0$  and thus  $d_0$ ); this is property G1.
- An A2 attacker *can* obtain  $\langle svr, pk_{svr}, \tau, t, v, d_1, a \rangle$  for some  $svr$ , but only for a  $svr$  that is not tainted and never compromised. Thus, the attacker has no information about the  $d_2$  in  $\tau$  and can forge only by succeeding in an online dictionary attack with  $svr$  (goal G2).
- Now consider an A3 attacker. If  $\text{Tainted}(dvc.\text{comp})$  contains some  $svr$ , then the attacker knows the  $d_2$  stored in the ticket  $\tau$  of a  $dvc$ 's record  $\langle svr, pk_{svr}, \tau, t, v, d_1, a \rangle$ , since it had corrupted the consenting server in the delegation protocol for  $svr$ . Similarly, if some  $svr \in \text{Active}(dvc.\text{comp})$  is ever compromised, then the attacker can obtain  $d_2$  by simply decrypting  $\tau$ . In either case, the A3 attacker can then conduct an offline dictionary attack on  $\pi_0$  using  $d_1$  and  $d_2$ , and so goal G3 is the best that can be achieved in this case.

- An attacker in class A4 compromises both  $\pi_0$  (i.e.,  $d_0$ ) and  $dvc$  when there is at least one active  $svr$  (and so it learns  $d_1$  for  $svr$ ). Moreover, the attacker can delegate from  $svr$  to any other  $svr' \in U$ , and obviously will learn the  $d_1'$  for that  $svr'$ . Thus, to achieve disabling, it is necessary that the attacker never corrupts any  $svr$  (and so never learns any  $d_2$  for any  $svr$ ). If this is the case, then goal G4 says that disabling all servers will prevent further forgeries.

We now proceed to a formal proof of goals G1–G4.

## 4.1 Definitions

To prove security of our system, we must first state requirements for the security of a mac scheme, of an encryption scheme, of a signature scheme, and of S-RSA-DEL.

**Security for mac schemes** We specify chosen-plaintext security for a mac schemes. We assume that a mac oracle is initialized with a random key  $a$ , and this oracle takes a message  $m$  as input and outputs  $\text{mac}_a(m)$ . An attacker  $A$  is allowed to query this mac oracle on arbitrary messages, and then  $A$  outputs a pair  $(x, y)$ .  $A$  succeeds if  $y = \text{mac}_a(x)$  and  $A$  did not previously query  $\text{mac}_a(x)$ . We say an attacker  $A$   $(q, \epsilon)$ -breaks the mac scheme if the attacker makes  $q$  queries to the mac oracle and succeeds with probability  $\epsilon$ .

**Security for encryption schemes** We specify adaptive chosen-ciphertext security [16] for an encryption scheme  $\mathcal{E} = (G_{enc}, E, D)$ . (For more detail, see [1, Property IND-CCA2].) An attacker  $A$  is given  $pk$ , where  $(pk, sk) \leftarrow G_{enc}(1^\lambda)$ .  $A$  is allowed to query a decryption oracle that takes a ciphertext as input and returns the decryption of that ciphertext (or  $\perp$  if the input is not a valid ciphertext). At some point  $A$  generates two equal length strings  $X_0$  and  $X_1$  and sends these to a test oracle, which chooses  $b \leftarrow_R \{0, 1\}$ , and returns  $Y = E_{pk}(X_b)$ . Then  $A$  continues as before, with the one restriction that it cannot query the decryption oracle on  $Y$ . Finally  $A$  outputs  $b'$ , and succeeds if  $b' = b$ . We say an attacker  $A$   $(q, \epsilon)$ -breaks a scheme if the attacker makes  $q$  queries to the decryption oracle, and  $2 \cdot \Pr(A \text{ succeeds}) - 1 \geq \epsilon$ .

**Security for signature schemes** We specify existential unforgeability versus chosen message attacks [9] for a signature scheme  $\mathcal{S} = (G_{sig}, S, V)$ . A forger is given  $pk$ , where  $(pk, sk) \leftarrow G_{sig}(1^\lambda)$ , and tries to forge signatures with respect to  $pk$ . It is allowed to query a signature oracle (with respect to  $sk$ ) on messages of its choice. It succeeds if after this it can output a valid forgery  $(m, \sigma)$ , where  $V_{pk}(m, \sigma) = 1$ , but  $m$  was not one of the messages signed by the signature oracle. We say a forger  $(q, \epsilon)$ -breaks a scheme if the forger makes  $q$  queries to the signature oracle, and succeeds with probability at least  $\epsilon$ .

**Security for S-RSA-DEL** Let  $\text{S-RSA-DEL}[\mathcal{E}, \mathcal{D}]$  denote an S-RSA-DEL system based on an encryption scheme  $\mathcal{E}$  and dictionary  $\mathcal{D}$ . A forger is given  $\langle e, N \rangle$  where  $(\langle e, N \rangle, \langle d, N, \phi(N) \rangle) \leftarrow G_{RSA}(1^\lambda)$ , and the public data generated by the initialization procedure for the system. The initialization procedure specifies  $svr_0$ . The goal of the forger is to forge RSA signatures with respect to  $\langle e, N \rangle$ . The forger is allowed to query a  $dvc$  oracle, a  $disable$  oracle,  $svr$  oracles, a password oracle, and (possibly) random oracles. A random oracle takes an input and returns a random hash of that input, in the defined range. A  $disable$  oracle query returns a value  $t$  that can be sent to the server to disable it for the device. A password oracle may be queried with  $comp$ , and returns  $\pi_0$ .

A  $svr$  oracle may be queried with  $handleSign$ ,  $handleDel$ ,  $disable$ , and  $comp$ . On a  $handleSign(\gamma, \delta, \tau)$  query, which represents the receipt of a message in the S-RSA-DEL signature protocol ostensibly from the device, it returns an output message  $\eta$  (with respect to the secret server data generated by the initialization procedure). On a  $handleDel(\gamma, \delta, \tau)$  query, which represents the receipt of a message in the S-RSA-DEL delegation protocol ostensibly from the device, it returns an output message  $\delta', \eta, \tau'$ . On a  $disable(t)$  query the  $svr$  oracle rejects all future queries with tickets containing ticket identifiers equal to  $h_{dsbl}(t)$  (see Section 3.5). On a  $comp$  query, the  $svr$  oracle returns  $sk_{svr}$ .

The  $dvc$  oracle may be queried with  $startSign$ ,  $finishSign$ ,  $startDel$ ,  $finishDel$ ,  $revoke$ , and  $comp$ . We assume there is an implicit notion of sessions so that the  $dvc$  oracle can determine the  $startSign$  query corresponding to a  $finishSign$  query and the  $startDel$  query corresponding to a  $finishDel$  query. On a  $startSign(m, svr)$  query, which represents a request to initiate the S-RSA-DEL signature protocol, if  $svr$  is authorized, the  $dvc$  oracle returns an output message  $\gamma, \delta, \tau$ , and sets some internal state (with respect to the secret device data and the password generated by the initialization procedure). On the corresponding  $finishSign(\eta)$  query, which represents the device's receipt of a response ostensibly from  $svr$ , the  $dvc$  oracle either aborts or returns a valid signature for the message  $m$  given as input to the previous  $startSign$  query. On a  $startDel(sv, svr')$  query, which represents a request to initiate the S-RSA-DEL delegation protocol, if  $svr$  is authorized, the  $dvc$  oracle returns an output message  $\gamma, \delta, \tau$ , and sets some internal state. On the corresponding  $finishDel(\delta', \eta, \tau')$  query, which represents the device's receipt of a response ostensibly from  $svr$ , the  $dvc$  oracle either aborts or authorizes  $svr'$ , i.e., it creates a new authorization record for  $svr'$ . On a  $revoke(sv)$  query, the  $dvc$  oracle erases the authorization record for  $svr$ , thus revoking the authorization of  $svr$ . On a  $comp$  query, the  $dvc$  oracle returns all stored authorization records.

A class A1, A2, or A3 forger *succeeds* if after attacking the system it can output a pair  $(m, \langle \sigma, r \rangle)$  where  $\sigma^e \equiv_N \text{encode}(m, r)$  and there was no  $startSign(m, svr)$  query. A class A4 forger *succeeds* if after attacking the system it can output a pair  $(m, \langle \sigma, r \rangle)$  where  $\sigma^e \equiv_N \text{encode}(m, r)$  and there was no  $handleSign(\gamma, \delta, \tau)$  query, where  $D_{sk_{svr}}(\gamma) = \langle m, *, *, * \rangle$ , before all servers received  $disable(t)$  queries, where  $h_{dsbl}(t)$  is the ticket identifier generated in initialization.

Let  $q_{dvc}$  be the number of  $startSign$  and  $startDel$  queries to the device. Let  $q_{svr}$  be the number of  $handleSign$  and  $handleDel$  queries to the servers. For Theorem 4.2, where we model  $h$  and  $f$  as random oracles, let  $q_h$  and  $q_f$  be the number of queries to the respective random oracles. Let  $q_o$  be the number of other oracle queries not counted above. Let  $\bar{q} = (q_{dvc}, q_{svr}, q_o, q_h, q_f)$ . In a slight abuse of notation, let  $|\bar{q}| = q_{dvc} + q_{svr} + q_o + q_h + q_f$ , i.e., the total number of oracle queries. We say a forger  $(\bar{q}, \epsilon)$ -breaks S-RSA-DEL if it makes  $|\bar{q}|$  oracle queries (of the respective type and to the respective oracles) and succeeds with probability at least  $\epsilon$ .

## 4.2 Theorems

Here we prove that if a forger breaks the S-RSA-DEL system with probability non-negligibly more than what is inherently possible in a system of this kind then either the underlying RSA signature scheme, the underlying mac scheme, or

the underlying encryption scheme used in S-RSA-DEL can be broken with non-negligible probability. This implies that if the underlying RSA signature scheme, the underlying mac scheme, and the underlying encryption scheme are secure, our system will be as secure as inherently possible.

We prove security separately for the different classes of attackers from Section 2.2. The idea behind each proof is a simulation argument. We assume that a forger  $F$  can break the S-RSA-DEL system, and then depending on how  $F$  attacks the system, we show that we can use it to either break the underlying mac scheme, break the underlying encryption scheme, or break the underlying RSA signature scheme.

For security against all classes of forgers, we must assume  $h$ ,  $f$ , and  $h_{\text{dele}}$  are random oracles. However, for certain types of forgers, weaker hash function properties suffice. For proving security against a forger in class A2, we make no requirement on  $h$ , and we only require  $f_v$  (for random  $v$ ) to have a negligible probability of collisions over the dictionary  $\mathcal{D}$ . For proving security against a class A1 or class A4 forger we make no requirement on  $h$  or  $f$ . For proving security against a class A4 forger, we also make no requirement on  $h_{\text{dele}}$ .

In the theorems below, we use “ $\approx$ ” to indicate equality to within negligible factors. Moreover, in our simulations, the forger  $F$  is run at most once, and so the times of our simulations are straightforward and omitted from our theorem statements. Due to space limitations, here we provide only proof sketches.

**THEOREM 4.1.** *Let  $h_{\text{dele}}$  be a random oracle. If a class A1 forger  $(\bar{q}, \epsilon)$ -breaks the S-RSA-DEL $[\mathcal{E}, \mathcal{D}]$  system, then there is a forger that  $(q_{\text{dvc}}, \epsilon')$ -breaks the RSA signature scheme with  $\epsilon' \approx \epsilon$ .*

**PROOF SKETCH.** Assume a class A1 forger  $F$  forges in the S-RSA-DEL system with probability  $\epsilon$ . Then we show how to break the underlying RSA signature scheme with probability  $\epsilon' \approx \epsilon$ . Say we are given an RSA public key  $\langle e, N \rangle$  and a corresponding signature oracle. We construct a simulation of the S-RSA-DEL system that behaves like the real system except we

1. use  $\langle e, N \rangle$  for the device’s RSA public key,
2. compute the user’s share of the private key  $(d_0)$  as normal, but choose  $\text{svr}_0$ ’s share  $d_2 \leftarrow_R \mathbb{Z}_N$  and the device’s share  $d_1 \leftarrow_R \{0, 1\}^{\lambda+\kappa}$ ,
3. use the knowledge of  $d_2$ , and the knowledge of queries made to the random oracle  $h_{\text{dele}}$ , to simulate the delegation protocol on the device and store the  $d_2$  value for the newly authorized server, and
4. use the signature oracle and the knowledge of  $d_2$  associated with the appropriate authorized server to simulate the signature protocol on the device.

We show that this simulation is statistically indistinguishable from the real system (from  $F$ ’s viewpoint), so since  $F$  forges with  $\epsilon$  probability in the real system, it also forges with roughly that probability in the simulation. Then to break the RSA signature scheme with probability  $\epsilon' \approx \epsilon$ , we simply run  $F$  in Sim and output any forgery produced by  $F$ .  $\square$

**THEOREM 4.2.** *Let  $h$ ,  $f$ , and  $h_{\text{dele}}$  be random oracles. If a class A3 forger  $(\bar{q}, \epsilon)$ -breaks the S-RSA-DEL $[\mathcal{E}, \mathcal{D}]$  system with  $\epsilon = \frac{q_h + q_f}{|\mathcal{D}|} + \psi$ , then there is a forger  $F^*$  that  $(q_{\text{dvc}}, \epsilon')$ -breaks the RSA signature scheme with  $\epsilon' \approx \psi$ .*

**PROOF SKETCH.** Assume a class A3 forger  $F$  forges in the S-RSA-DEL system with probability  $\frac{q_h + q_f}{|\mathcal{D}|} + \psi$ . Then we show how to break the underlying RSA signature scheme with probability  $\epsilon' \approx \psi$ . Say we are given an RSA public key  $\langle e, N \rangle$  and a corresponding signature oracle. We construct a simulation of the S-RSA-DEL system as in the proof of Theorem 4.1, except that the simulation aborts if the  $h$  oracle or  $f$  oracle is queried with  $\pi_0$ . We show that this simulation is statistically indistinguishable from the real system (from  $F$ ’s viewpoint) unless the simulation aborts, the probability of which is exactly that of an offline dictionary attack (i.e.,  $\frac{q_h + q_f}{|\mathcal{D}|}$ ). So since  $F$  forges with probability  $\frac{q_h + q_f}{|\mathcal{D}|} + \psi$  in the real system, it forges with probability  $\epsilon' \approx \psi$  in the simulation. Then to break the RSA signature scheme, we simply run  $F$  in Sim and output any forgery by  $F$ .  $\square$

**THEOREM 4.3.** *Suppose  $f_v$  (for random  $v$ ) has a negligible probability of collision over  $\mathcal{D}$  and  $h_{\text{dele}}$  is a random oracle. If a class A2 forger  $(\bar{q}, \epsilon)$ -breaks the S-RSA-DEL $[\mathcal{E}, \mathcal{D}]$  system where  $\epsilon = \frac{q_{\text{svr}}}{|\mathcal{D}|} + \psi$ , then either (1) there is an attacker  $A$  that  $(q_{\text{svr}}, \frac{\psi}{4|U|q_{\text{dvc}}})$ -breaks the mac, (2) there is an attacker  $A^*$  that  $(2q_{\text{svr}}, \frac{\psi}{4|U|(q_{\text{dvc}} + q_{\text{svr}} + 1)})$ -breaks  $\mathcal{E}$ , or (3) there is a forger  $F^*$  that  $(q_{\text{dvc}}, \epsilon'')$ -breaks the RSA signature scheme with  $\epsilon'' \approx \frac{\psi}{4}$ .*

**PROOF SKETCH.** Assume a class A2 forger  $F$  forges in the S-RSA-DEL system with probability  $\frac{q_{\text{svr}}}{|\mathcal{D}|} + \psi$ . Say a good server is one that never gets compromised. Then consider a simulation Sim of the S-RSA-DEL system that behaves like the real system except it (1) replaces all real ciphertexts generated for good servers with encryptions of zero strings (storing the real ciphertexts), and (2) modifies the protocols for good servers to use the (stored) real ciphertexts instead of the encryptions of zero strings. In other words, the only way Sim and the real system differ (from  $F$ ’s viewpoint) is with respect to the good server ciphertexts. Now we say  $F$  wins if it either forges a signature, forges the mac sent to the dvc oracle in the last message of a delegation protocol session with an uncorrupted server, or decrypts the ciphertext  $\eta$  generated by a server in the last message of a delegation protocol session with an uncorrupted server (detected by  $F$  querying the  $h_{\text{dele}}$  oracle with a certain value). By our assumption,  $F$  wins in the real system with probability  $\frac{q_{\text{svr}}}{|\mathcal{D}|} + \psi$ . Then if  $F$  wins in Sim with probability only  $\frac{q_{\text{svr}}}{|\mathcal{D}|} + \frac{3\psi}{4}$ , we can use a standard hybrid argument to break the encryption scheme used by the servers with probability  $\frac{\psi}{4|U|(q_{\text{dvc}} + q_{\text{svr}} + 1)}$ .

Now we consider the case that  $F$  wins in Sim with probability more than  $\frac{q_{\text{svr}}}{|\mathcal{D}|} + \frac{3\psi}{4}$ . If  $F$  forges a mac (sent to the dvc oracle in the last message of a delegation protocol session with an uncorrupted server) with probability more than  $\frac{\psi}{2}$  in Sim, then consider a simulation Sim’ that behaves like Sim except it guesses the server svr involved in the mac forgery, considers svr good (if it was not already), and aborts when and if svr is compromised. Now the only way Sim’ and

Sim differ (from  $F$ 's viewpoint) is with respect to the ciphertexts for  $svr$ , if  $svr$  were not good already. Then if  $F$  forges a mac supposedly from  $svr$  in  $\text{Sim}'$  with probability only  $\frac{\psi}{4|U|}$  (where the probability is taken over the random choice of  $svr$ ), we can use a standard hybrid argument to break the encryption scheme used by  $svr$  with probability  $\frac{\psi}{4|U|(q_{dvc}+q_{svr}+1)}$ . Otherwise we can break the underlying mac scheme with probability  $\frac{\psi}{4|U|q_{dvc}}$ , as follows. Say we are given a mac oracle initialized with a random key. We run  $\text{Sim}'$ , except we (1) guess the delegation protocol session (with  $svr$ ) for which  $F$  will forge a mac, and (2) use the mac oracle to generate any mac values returned by  $svr$  in that session. Finally, we output the mac value that  $F$  sends to the device in the last message of that session. Note that this modified version of  $\text{Sim}'$  is perfectly indistinguishable from  $\text{Sim}'$  (from  $F$ 's viewpoint), since any ciphertext that would have contained the real mac key  $\alpha$  was replaced by an encryption of a zero string.

The last case is that  $F$  wins in  $\text{Sim}$  with probability more than  $\frac{q_{svr}}{|D|} + \frac{3\psi}{4}$ , but forges a mac with probability at most  $\frac{\psi}{2}$  in  $\text{Sim}$ . Then  $F$  must forge a signature, or decrypt a ciphertext  $\eta$  generated by the server in a delegation protocol session, with probability more than  $\frac{\psi}{4}$  in  $\text{Sim}$ , and we show how to break the underlying RSA signature scheme with probability roughly  $\frac{\psi}{4}$ . Say we are given an RSA public key  $\langle e, N \rangle$  and a corresponding signature oracle. We run  $\text{Sim}$ , except we

1. use  $\langle e, N \rangle$  for the device's RSA public key,
2. compute the user's and device's shares of the private key ( $d_0$  and  $d_1$ , respectively) as normal during initialization, but choose  $svr_0$ 's share  $d_2 \leftarrow_R \mathbb{Z}_N$ ,
3. generate a message  $m$  and let  $w = \text{encode}(m, 0^{\kappa_{sig}})$ ,
4. use the signature oracle and knowledge of  $d_0$  and  $d_1$  associated with any given authorized server to simulate that server,
5. modify the server delegation protocol to multiply  $w$  into  $\nu_1$  (the raw-RSA-encrypted random value), and
6. use the knowledge of the value  $d_{21}$  from the simulation of the delegation protocol on the server to simulate the delegation protocol on the device (so as to avoid needing to perform the raw RSA decryption).

Finally we either output any forgery produced by  $F$ , or if  $F$  queries  $h_{\text{delete}}$  with the raw RSA decryption of one of the  $\nu_1$  values, use this decryption to compute the raw RSA decryption of  $w$ , and thus compute the signature of the message  $m$ . This modified version of  $\text{Sim}$  is statistically indistinguishable from  $\text{Sim}$  (from  $F$ 's viewpoint), since any ciphertext that would have contained the real  $d_2$  or  $\rho$  values (the latter used to blind the  $\nu_1$  values) was replaced by an encryption of zeros, and without knowing the  $d_{21}$  value used when delegating to a server,  $F$  has no information about the  $d_2$  value for that server, assuming the server is not compromised or tainted.  $\square$

**THEOREM 4.4.** *Suppose the RSA signature scheme is deterministic (i.e.,  $\kappa_{sig} = 0$ ). If a class A4 forger  $(\bar{q}, \epsilon)$ -breaks the S-RSA-DEL $[\mathcal{E}, \mathcal{D}]$  system, then there is either an attacker  $A$  that  $(q_{svr}, \frac{\epsilon}{3q_{dvc}})$ -breaks the mac function, an attacker  $A^*$  that  $(2q_{svr}, \frac{\epsilon}{3|U|(q_{dvc}+q_{svr}+1)})$ -breaks  $\mathcal{E}$ , or a forger  $F^*$  that  $(q_{svr}, \frac{\epsilon}{3})$ -breaks the RSA signature scheme.*

**PROOF SKETCH.** Assume a class A4 forger  $F$  forges in the S-RSA-DEL system with probability  $\epsilon$ . Then consider the simulation  $\text{Sim}$  from the proof of Theorem 4.3. The only way  $\text{Sim}$  and the real system differ (from  $F$ 's viewpoint) is with respect to the server ciphertexts. Now we say  $F$  wins if it either forges a signature or forges the mac sent to the device in the last message of the delegation protocol. By our assumption,  $F$  wins in the real system with probability  $\epsilon$ . Then if  $F$  wins in  $\text{Sim}$  with probability only  $\frac{2\epsilon}{3}$ , we can use a standard hybrid argument to break the encryption scheme used by the servers with probability  $\frac{\epsilon}{3|U|(q_{dvc}+q_{svr}+1)}$ .

Now we consider the case that  $F$  wins with probability greater than  $\frac{2\epsilon}{3}$  in  $\text{Sim}$ . If  $F$  wins by forging the mac with probability greater than  $\frac{\epsilon}{3}$  in  $\text{Sim}$ , then we can break the underlying mac scheme with probability  $\frac{\epsilon}{3q_{dvc}}$ , as follows. Say we are given a mac oracle initialized with a random key. We run  $\text{Sim}$ , except we (1) guess the delegation protocol session for which  $F$  will forge a mac, and (2) use the mac oracle to generate any mac values returned by the server in that session. Finally, we output the mac value that  $F$  sends to the device in the last message of that session. Note that this modified version of  $\text{Sim}$  is perfectly indistinguishable from  $\text{Sim}$  (from  $F$ 's viewpoint), since any ciphertext that would have contained the real mac key  $\alpha$  was replaced by an encryption of a zero string.

The final case is that  $F$  wins with probability greater than  $\frac{2\epsilon}{3}$  in  $\text{Sim}$ , but forges the mac with probability at most  $\frac{\epsilon}{3}$  in  $\text{Sim}$ . Then  $F$  must win by forging a signature with probability at least  $\frac{\epsilon}{3}$  in  $\text{Sim}$ , and we show how to break the underlying RSA signature scheme with probability  $\frac{\epsilon}{3}$ . Say we are given an RSA public key  $\langle e, N \rangle$  and a corresponding signature oracle. We run  $\text{Sim}$ , except we

1. use  $\langle e, N \rangle$  for the device's RSA public key,
2. compute the user's and device's shares of the private key ( $d_0$  and  $d_1$ , respectively) as normal, but (arbitrarily) set  $svr_0$ 's share ( $d_2$ ) to zero, and
3. use the signature oracle and knowledge of  $d_0$  and  $d_1$  associated with any given authorized server to simulate that server.

Finally we output any forgery produced by  $F$ . This modified version of  $\text{Sim}$  is perfectly indistinguishable from  $\text{Sim}$  (from  $F$ 's viewpoint), since any ciphertext that would have contained the real  $d_2$  value was replaced by an encryption of a zero string.  $\square$

## 5. REFERENCES

- [1] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO '98* (LNCS 1462), pp. 26–45, 1998.
- [2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1<sup>st</sup> ACM Conf. on Comp. and Comm. Security*, pp. 62–73, 1993.
- [3] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *EUROCRYPT '94* (LNCS 950), pp. 92–111, 1995.
- [4] M. Bellare and P. Rogaway. The exact security of digital signatures—How to sign with RSA and Rabin. In *EUROCRYPT '96* (LNCS 1070), pp. 399–416, 1996.
- [5] C. Boyd. Digital multisignatures. In H. J. Beker and F. C. Piper, editors, *Cryptography and Coding*, pp. 241–246. Clarendon Press, 1989.
- [6] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO '98* (LNCS 1462), pp. 13–25, 1998.
- [7] D. E. Denning. Digital signatures with RSA and other public-key cryptosystems. *Comm. of the ACM* 27(4):388–392, Apr. 1984.
- [8] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. In *Proceedings of the 1995 ISOC Network and Distributed System Security Symposium*, February 1995.
- [9] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. on Computing* 17(2):281–308, Apr. 1988.
- [10] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *4<sup>th</sup> ACM Conf. on Comp. and Comm. Security*, pp. 100–110, 1997.
- [11] D. N. Hoover and B. N. Kausik. Software smart cards via cryptographic camouflage. In *1999 IEEE Symp. on Security and Privacy*, pp. 208–215, 1999.
- [12] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *2<sup>nd</sup> USENIX Security Workshop*, Aug. 1990.
- [13] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. DIMACS Technical Report 2001-19, May 2001. Extended abstract in *2001 IEEE Symp. on Security and Privacy*, May 2001.
- [14] R. Morris and K. Thompson. Password security: A case history. *Comm. of the ACM*, 22(11):594–597, Nov. 1979.
- [15] U. Maurer and S. Wolf. The Diffie-Hellman protocol. *Designs, Codes, and Cryptography* 19:147–171, Kluwer Academic Publishers, 2000.
- [16] C. Rackoff and D. Simon. Noninteractive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO '91*, (LNCS 576), pp. 433–444, 1991.
- [17] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM* 21(2):120–126, Feb. 1978.
- [18] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In *EUROCRYPT '98*, pp. 1–16, 1998.