Traitor Tracing Schemes for Protected Software Implementations

Marc Joye
Technicolor, Security & Content Protection Labs
Cesson-Sévigné, France
marc.joye@technicolor.com

Tancrède Lepoint
Technicolor, Security & Content Protection Labs
Cesson-Sévigné, France
tancrede.lepoint@technicolor.com

ABSTRACT
This paper considers the problem of converting an encryption scheme into a scheme in which there is one encryption process but several decryption processes. Each decryption process is made available as a protected software implementation (decoder). So, when some digital content is encrypted, a legitimate user can recover the content in clear using its own private software implementation. Moreover, it is possible to trace a decoder in a black-box fashion in case it is suspected to be an illegal copy. Our conversions assume software tamper-resistance.

Categories and Subject Descriptors
C.2.0 [Computer-Communication Networks]: General—Security and protection; E.3 [Data]: Data Encryption—Public key cryptosystems

General Terms
Security, Design

Keywords
Content protection, content distribution, traitor tracing, software decoders, tamper resistance, obfuscation.

1. INTRODUCTION
Premium content is usually encrypted so as to control its distribution. In content distribution systems, authorized users are given a hardware or software decoder containing a decryption key that allows them to get access to the content in clear. But users may collude and try to produce a pirate decoder; i.e., a non-registered decoder able to decrypt. Traitor tracing schemes [13, 14] enable an authority to recover the identity of at least one of the legitimate users who participated in the construction of a pirate decoder. Such a user is called a traitor. The aim of traitor tracing schemes is to deter users from building pirate decoders.

In the non-black-box setting, traitor tracing schemes assume that decoders are “open” and so that a legitimate user knows her decoder’s private decryption key. Traitor tracing schemes also assume that each decoder has a different private key: a decryption key uniquely identifies a decoder (and thus a user). In order to reduce the ciphertext expansion, when a same content has to be sent to a (large) set of users, traitor tracing schemes usually work in tandem with broadcast encryption techniques [21] (see also [4]). Ciphertext expansion is used as a metric to measure the “quality” of a traitor tracing scheme.

In that sense, an optimal traitor scheme could be obtained under the assumption of tamper resistance [27]. If the data stored in the decoder (including the cryptographic keys) are protected against unauthorized access, there is no need to add tracing capabilities. Tamper resistance rules out colluding attacks. So, further assuming that decoders cannot be cloned, the same decryption key could be used in all decoders. The size of the ciphertexts would therefore be constant, regardless of the number of users in the system. A possible realization is to rely on smart cards: they are tamper-resistant and unclonable. We note however that such a system also requires the decryption algorithm in the smart card to be properly implemented [1]. In particular, the implementation should not leak information about the decryption key. Examples of implementation attacks include side-channel attacks [26] and fault attacks [7].

Software-based solutions offer a number of advantages. They are cheaper and easier to distribute and to update (for example if a security flaw is identified). Software tamper-resistance [2] can be achieved through a combination of obfuscation techniques [16] and of cryptographic hashing [11, 25, 12]. Cryptographic hashing checks the integrity as the program is running while obfuscation makes it harder to realize intended changes in functionality (and in particular to bypass the integrity checks). Unclonability is difficult to achieve without resorting to hardware. We do not solve this problem in this paper. Instead, we suggest a method deterring users from distributing copies of their [software] decoder.

A possible approach is to embed copy-specific watermarks in the code [17]. This enables tracking illegal copies. A concrete implementation which nicely combines with integrity checking techniques is presented in [25] (see also [24]). The approach we propose is different, complementary. We require each decoder to have a different private key while being compatible with any encryption algorithm. Our idea is to add an extra, copy-specific entry to the decryption algo-
such that no information on $K$ is easily verified that the knowledge of at random.

Learning one of the components does not reveal half of the
ware implementations of discrete-log based cryptosystems.
methodology. We present traitor tracing schemes for soft-
Outline of the paper.
The rest of this paper is organized as follows. In the next
section, we review several key splitting techniques, with a
special focus on RSA. In Section 2, we present several traitor
tracing schemes for software implementations of RSA-based
cryptosystems. We also discuss the security of the schemes
we so obtain. Section 4 exemplifies the generality of our
methodology. We present traitor tracing schemes for soft-
ware implementations of discrete-log based cryptosystems.
Finally, we conclude in Section 5.

2. SECRET SPLITTING

Secret splitting or secret sharing [5, 31] is a cryptographic
techique to split a secret key into (at least) two components.
Learning one of the components does not reveal half of the
secret; it actually reveals no information at all. A simple way
to split a $k$-bit key $K$ in two shares is to choose uniformly at
random $K_1 \in \{0,1\}^k$ and to define $K_2 = K \oplus K_1$.
It is easily verified that the knowledge of $K_1$ (or $K_2$) yields
no information on $K$. The two shares are needed to the
reassemble secret key $K$.

For RSA [30], or more generally for most public-key cryp-
tosystems, the underlying algebraic structure can be ex-
loited to derive further key splitting schemes with differ-
tent properties. For concreteness, consider an RSA modulus
$N = pq$ where $p$ and $q$ are two large balanced primes. The
public primitive consists in raising some $x \in \mathbb{Z}_N^*$ to the $e$-
th power and the corresponding private primitive consists
in raising some $y \in \mathbb{Z}_N^*$ to the $d$-th power. The public ex-
ponent $e$ is coprime to $\lambda(N)$ and matches the private
exponent $d$ through the relation $ed \equiv 1 \pmod{\lambda(N)}$, where
$\lambda$ denotes Carmichael function [3]. By construction, if we
let $y = x^e \mod N$ then $x$ can be recovered as $y^d \mod N$:
\[
y^d \equiv (x^e)^d \equiv x^{ed} \mod \lambda(N) \equiv x \mod N.
\]

Multiplicative splitting.
The multiplicative splitting breaks down the private ex-
ponent $d$ into two components $(d_1, d_2)$ where
\[
\begin{align*}
&d_1 \text{ is a random element in } \mathbb{Z}_{\lambda(N)}^*; \\
&d_2 \text{ is computed as } d_2 = d - d_1 \mod \lambda(N).
\end{align*}
\]

The private exponentiation, $y^d \mod N$, can then be evaluated as
\[
(y^{d_1})^{d_2} \mod N.
\]
The multiplicative splitting was introduced by Boyd in [10]
(and analyzed in [23] as a means to produce digital multi-
signatures. Each party receives a share $d_i$ ($i \in \{1,2\}$), which
is then used to create a joint signature.

Additive splitting.
The additive splitting, also used in [10] as an alternative
to produce multisignatures, is a variant where the private
exponent $d$ is split additively into two shares $(d_1, d_2)$ where
\[
\begin{align*}
&d_1 \text{ is a random element in } \mathbb{Z}_{\lambda(N)} \setminus \{0\}; \\
&d_2 \text{ is computed as } d_2 = d - d_1 \mod \lambda(N).
\end{align*}
\]

The private exponentiation, $y^d \mod N$, can now be carried out as
\[
y^{d_1} \cdot y^{d_2} \mod N.
\]

With the multiplicative splitting, the two half exponen-
tiations are performed in a serial way. In contrast, the
additive splitting prescribes parallel operation. This was
used to design a variant of RSA, known as mediated RSA
(mRSA) [8], providing immediate revocation capabilities in
a PKI. mRSA involves an on-line semi-trusted entity, called
SEM, that issues message-specific tokens. The SEM is given
the key share $d_1$ while the user receives the second share, $d_2$.
If the user wants to sign or to decrypt a “message” $y$, she
must first obtain the token $y^{d_1} \mod N$. To revoke the user’s
ability to sign or decrypt messages, the SEM simply stops is-
scuing tokens. A multiplicative version of mRSA is presented
in [22]. Further generalizations are described in [33, 19].

Euclidean splitting.
Key splitting techniques also find applications in the de-
velopment of countermeasures against certain implementa-
tion attacks. It can be seen as a combination of the two
previous techniques (sequential and parallel splittings). The
Euclidean splitting [13] splits the private exponent $d$ into
two components $(d_1, d_2)$ where
\[
\begin{align*}
&d_1 \text{ is a random element in } \{0,1\}^*; \\
&d_2 = d_2, h || d_2, l \text{ is computed as } d_2, h = \lfloor d/d_1 \rfloor \text{ and } d_2, l = d \mod d_1.
\end{align*}
\]

Remarking that $d = d_1, d_2, h + d_2, l$, the private exponentiation
can be evaluated as
\[
(y^{d_1})^{d_2, h} \cdot y^{d_2, l} \mod N.
\]

The advantage of the Euclidean splitting is computational.
If parameter $\kappa$ is set to $d_{1/2} \approx \sqrt{N}$ (i.e., half the bit-
length of $N$), then the entity holding $d_2$ has to perform a
double exponentiation of the form $y^{d_{2, h}} \cdot y^{d_{2, l}} \mod N$
where the exponents $d_2, h$ and $d_2, l$ are half-sized. Since
the cost of a double exponentiation is only slightly more expen-
tive than a single exponentiation using Straus’ method [32]
(a.k.a. Shamir’s trick; see [20]), the overall computation is
roughly twice faster.

More generally, we notice that the Euclidean splitting can be
applied directly to $d$ or to an equivalent representation
thereof like $d + k\lambda(N)$ for some integer $k$. 

\textsuperscript{1}{Carmichael function of $N$ defines the exponent of the
multiplicative group $\mathbb{Z}_N^*$, that is, the smallest positive integer $t$
such that $a^t \equiv 1 \pmod{N}$ for every $a \in \mathbb{Z}_N^*$. For an RSA-
module $N = pq$, $\lambda(N)$ is given by $\text{lcm}(p-1, q-1)$.}
3. TRAITOR TRACING SCHEMES

FOR RSA

As mentioned in the introduction, our aim is to deter users from distributing the decryption software they receive from content providers. To this end, each user receives a personalized decryption software containing a unique decryption key. Yet a same encrypted content can be decrypted using any personalized decryption software. The key property of our schemes is that it is possible to identify a personalized decryption software. In other words, it is possible to trace a given implementation or a copy thereof.

Properly implemented decryption algorithms make use of state-of-the-art obfuscation and tamper-resistant techniques. The schemes we present in this section only rely on software and do not require secure hardware.

3.1 Basic Scheme

Imagine that a same digital content has to be sent to a large number of subscribers. Contents are encrypted using RSA with public key $(N, e)$ and private key $d$. Specifically, using the RSA cryptosystem, a binary string $m$ is encrypted as $c = \mu(m)^e \mod N$ for some (probabilistic) padding function $\mu$ (e.g., OAP [3]). Then, given ciphertext $c$, $m$ is recovered from $c^d \mod N$ using the private key $d$. A classical solution is to provide each legitimate user with a protected software implementation or a copy thereof.

Our approach is complementary. We propose to split the decryption key into two components. For each user, the first component, $\sigma_{ID}$, is derived from a unique identifier ID. The second component, $d_{ID}$, is defined so that the value of decryption key $d$ can be recovered from the pair $(\sigma_{ID}, d_{ID})$. We let $R$ denote the combining function that on input $\sigma_{ID}$ and $d_{ID}$ returns $d$. Component $d_{ID}$ is embedded in a protected software implementation while component $\sigma_{ID}$ serves as an additional input to the decryption software.

Our proposed implementation is different from existing ones. A classical solution is to provide each legitimate user with a protected software implementation while component $d_{ID}$ serves as an additional input to the decryption software.

![Figure 1: Classical scheme](image)

| $c$ | $c^d \mod N$ | $m$ or $\perp$
|---|---|---|

**Figure 1: Classical scheme**

In more detail, the scheme goes as follows. In the initialization phase, a content provider sets up an RSA modulus $N$ and a pair of encryption/decryption keys $(e, d)$. When a user wants to subscribe to the system, she has to produce a unique identifier ID (e.g., email address, bank account number, password, ...) for the system. The user then receives her personal string $\sigma_{ID}$ together with a protected software implementation (decoder) which embeds the matching secret value $d_{ID}$. When a registered user wishes to get access to an encrypted content $c$, she enters her string $\sigma_{ID}$ in her decoder which decrypts $c$ in two steps as:

- evaluate $d = R(\sigma_{ID}, d_{ID})$;
- compute $c^d \mod N$;

the last operation recovers and returns the plain content $m$, or returns $\perp$ in case the decryption failed. See Fig. 2.

Selecting $\sigma_{ID}$.

Without loss of generality, we view the unique identifier ID of a user as a binary string. There are several possibilities to derive $\sigma_{ID}$ from ID. Writing $\sigma_{ID} = f(ID)$, we require function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ to be collision-resistant. There are several possible choices for function $f$:

- $f$ can be the identity map;
- $f$ can be a cryptographic hash function (e.g., SHA-1, SHA-2 family, ...);
- $f$ can be a [deterministic] symmetric encryption function (e.g., AES, Serpent, ...);
- $f$ can be a [deterministic] authentication function (e.g., HMAC, RSA-FDH, ...); etc.

Defining $d_{ID}$.

The value of $d_{ID}$ is derived from the string $\sigma_{ID}$ and decryption key $d$. Again, there are plenty of choices. For example, viewing $d$ as a binary string, we can define $d_{ID} = \sigma_{ID} \oplus d$ (and thus $R(\sigma_{ID}, d_{ID}) = \sigma_{ID} \oplus d_{ID}$).

As will be shown in §3.2, the way $d_{ID}$ is constructed may simplify the implementation.

**Tracing traitors.**

Tracing a traitor is pretty easy. Suppose that a legitimate user (corresponding to identifier ID) is allegedly suspected to have made available illegal copies of her software decoder. When such a copy is found, it can be tested whether it corresponds to ID as follows:

- obtain a valid ciphertext $c$;
- compute $\sigma_{ID} = f(ID)$ (where ID is the identifier of the putative traitor) using derivation function $f$;
- input $c$ and $\sigma_{ID}$ to the pirated copy of the software decoder and check whether $c$ decrypts correctly or not.

If so, the user with identifier ID is identified as the source of leakage.

Interestingly, the tracing capability requires the knowledge of derivation function $f$. If the function $f$ is public, anyone can test if a given software decoder corresponds to some identifier ID. In contrast, if the function $f$ is private (for instance, if it is keyed), traitor tracing is solely possible for “authorized” entities (namely, those knowing $f$).

3.2 Enhanced Schemes

Although applicable to any cryptosystem, the basic scheme (Fig. 2) can be intricate to implement in a secure way. The proposed implementation is different from existing ones. Specifically, in addition to the usual decryption function...
(i.e., \(c^d \mod N\) in the case of RSA), our basic scheme first requires to reconstruct \(d\) from \(\sigma_{ID}\) and \(d_{ID}\). A possible fix to mitigate the damages would be to make private the combining function \(\mathcal{R}\).

A better approach would be to design a software implementation that solely performs operations similar to the regular decryption function. This allows one to get a protected implementation at minimum cost, augmented with our tracing method. More importantly, this allows one to base the security on proven techniques. While this may appear difficult in the general case, we will see that for RSA it is not. We can exploit the underlying algebraic structure. The "regular decryption function" for RSA is the modular exponentiation. The goal is, on input \((c, \sigma_{ID})\), to evaluate \(c^d \mod N\) using a routine that can securely evaluate operations of the form \(x^{\lambda_1} \mod N\) for a fixed value \(d_{ID}\). This is where the key splitting techniques introduced in Section 2 come into play.

An application of the different splitting techniques lead to the schemes depicted in the next figure. Given identifier ID and corresponding identifying value \(\sigma_{ID}\), the value of \(d_{ID}\) is respectively defined as:

- **multiplicative scheme**: \(d_{ID} \equiv d/\sigma_{ID} \mod (\lambda(N))\);
- **additive scheme**: \(d_{ID} \equiv d - \sigma_{ID} \mod (\lambda(N))\);
- **Euclidean scheme**: \(d_{ID} = (d_{ID}^{(1)}, d_{ID}^{(2)})\) with \(d_{ID}^{(1)} = \lfloor d/\sigma_{ID} \rfloor\) and \(d_{ID}^{(2)} = d \mod \sigma_{ID}\).

![Figure 3: Enhanced schemes for RSA decryption](image)

One may argue that the enhanced schemes involve modular exponentiations with an exponent other than \(d_{ID}\), namely for the computation of \(c_0\). We note that \(\sigma_{ID}\) is not a sensitive value. Therefore, the value of \(c_0\) (as well as its computation) does not reveal any sensitive information. For the Euclidean variant, we suppose that the double exponentiation in Step 2 is evaluated as an atomic operation based on the Straus-Shamir technique (otherwise there is no use to consider this variant).

### 3.3 Security Considerations

The traitor tracing schemes presented in this section require state-of-the-art obfuscation and tamper-resistant techniques. In particular, if the value of \(d_{ID}\) is recovered then the knowledge of \(\sigma_{ID}\) enables the reconstruction of the private decryption key \(d\) as \(\mathcal{R}(\sigma_{ID}, d_{ID})\). As aforementioned, keying \(\mathcal{R}\) in the basic scheme may help to mitigate the damages as the attacker should also recover the value of the key used by \(\mathcal{R}\).

For the enhanced schemes, the situation is better. The implementation inherits the same security guarantees as the regular implementation (i.e., without our added tracing capabilities). There is no security loss. In our RSA implementations (Fig. 3), the sensitive operation (Step 2) is a modular exponentiation with a private exponent, as is done in the regular implementation (the only difference is that \(d\) is replaced by \(d_{ID}\)). Step 3 leaks no sensitive information. An alternative to the multiplicative variant as depicted in Fig. 3(a) could be to use \(c_0\) as an input (instead of \(c\)) in the regular implementation, tailored with exponent \(d_{ID}\), compare Fig. 3 and Fig. 3(b). For the additive variant (Fig. 3(b)), the multiplication in Step 3 should be implemented with care. The recovery of \(c_1\) may give more power to the attacker; the expected security level is not necessarily preserved. For that reason, the multiplicative variant should be preferred.

We suppose that the attacker is a legitimate user. Her goal is to build an untraceable decoder, or at least a decoder that does not trace her identity.

#### 3.3.1 Key recovery attacks

A possible way to get an untraceable software decoder is to recover \(d_{ID}\), and then \(d\) from \(\sigma_{ID}\).

**Chosen-ciphertext security.**

Since the attacker possesses her own copy of the decryption software, she can use it to mount chosen-ciphertext attacks. This means that the underlying cryptosystem must at least meet the notion of unbreakability under chosen-ciphertext attacks. This notion is implied by the classical notion of indistinguishably under chosen-ciphertext attacks (IND-CCA). Although we are not aware of (black-box) key recovery attacks against plain RSA (a.k.a. textbook RSA or no-pad RSA) — which is obviously not chosen-ciphertext secure, we do not recommend its use. We rather recommend the use of RSA-OAEP or any other IND-CCA RSA-based cryptosystem.

**Size of \(d_{ID}\).**

There is no size recommendations for \(\sigma_{ID}\); the only requirement is that each \(\sigma_{ID}\) must be unique. Concerning \(d_{ID}\), Wiener’s attack tells us the a private RSA key cannot be chosen too small. A similar remark holds for \(d_{ID}\). In order to prevent Wiener’s attack and more sophisticated LLL-based attacks, we recommend that \(d_{ID}\) should be at least of the size of \(N^{1/2}\). [9]

#### 3.3.2 Re-obfuscation attacks

Re-obfuscation is another possible avenue for the attacker. The attacker might try to produce a program (from the software decoder she received) that does not take \(\sigma_{ID}\) on input so that tracing would not be possible. Worse, the attacker
could even try to produce a program with a chosen $\sigma_{ID'}$ to falsely accuse the user with identifier $ID'$ (impersonation). This is depicted in Fig. 4. Note that such an attack only makes sense for open environments; it is readily ruled out in semi-open environments (i.e., executing only signed code).

Of course the original decoder (i.e., the implementation the user received — in light gray on the picture) is needed because the value of $d_{ID}$ is unknown to the attacker. A pirate decoder may look like

1. call the original decoder with $(c, \sigma_{ID})$ as inputs and obtain the result, say $R$ (note that $R = m$ or $\bot$);
2. check whether the input $\sigma_{ID}$ corresponds to the user the pirate wants to impersonate; if so, return $R$, otherwise return $\bot$.

This will however not work if there is a global integrity check of the original decoder. The software decoder will then detect that it is part of another program and take appropriate actions. This can be done through self-checking means, in a static or dynamic way.

3.3.3 Collusion attacks

Collusion attacks are usually considered in the context of traitor tracing. A coalition of users tries to generate a decoder not related to them. This does not apply here: the knowledge of several $\sigma_{ID}$’s does not provide useful information to any coalition of users since $\sigma_{ID}$ is unrelated to private key $d$.

4. FURTHER SCHEMES

Most public-key cryptosystems are based on group theory. Provided the discrete logarithm problem is hard, cryptographic schemes can be devised. Examples include multiplicative groups of finite fields or elliptic curves over finite fields.

Consider a (multiplicatively written) cyclic group $G = \langle g \rangle$ of order $q$, generated by an element $g$. ElGamal cryptosystem \[20\] can easily be extended to this generalized setting. Let $H : G \to M$ be a cryptographic hash function that maps elements of $G$ to elements of message space $M$. The private key is a random element $d \in \mathbb{Z}_q$ and the corresponding public key is $y = g^d$. A message $m \in \mathbb{M}$ is encrypted as $c = (g^m, m \oplus H(g^m))$ where $k$ is uniformly chosen at random in $\mathbb{Z}_q$. Given ciphertext $c = (u, v)$, message $m$ is recovered as $v \oplus H(u^d)$. The correctness follows by observing that $u^d = (g^k)^d = y^k$.

4.3.3 Multiplicative splitting

Remarking that the main operation for the decryption process is an exponentiation in $G$ (i.e., $u^d$), the splitting techniques of Section 2 readily apply here as well. For example, defining $d_{ID} = d/\sigma_{ID} \text{ mod } q$, the multiplicative splitting yields the following ‘enhanced’ scheme

$$c = (u, v) \xrightarrow{\sigma_{ID}} \begin{cases} u_0 = u^{*\sigma_{ID}} \vspace{1ex} \\ v \oplus H(u_0^{*\sigma_{ID}}) \end{cases} \text{ or } \bot$$

4.4 Enhanced ElGamal decryption

The conversion from $u$ to $u_0$ is analogous to the proxy re-encryption technique used in \[6\] (see also \[28\]). Actually, our enhanced schemes can be seen as the re-encryption of a ciphertext for the user holding the private key $d_{ID}$, followed by the regular decryption with key $d_{ID}$.

ElGamal cryptosystem is semantically secure under the decision Diffie-Hellman assumption against chosen-plaintext attacks. We present below an implementation of a variant secure against chosen-ciphertext attacks, namely the Cramer-Shoup cryptosystem \[18\]. As an illustration, we use the multiplicative splitting.

The Cramer-Shoup cryptosystem makes use of a universal one-way hash function $H : \{0, 1\}^* \to \mathbb{Z}_q$ \[23\]. Let $g$ and $h$ be two random generators of $G$. The private key is \{d, s1, s2, t1, t2\} that are elements in $\mathbb{Z}_q$. The public key is \{y, y1, y2\} where $y = g^d$, $y1 = g^{s1}h^{t2}$, $y2 = g^{s2}h^{t2}$. The encryption of a message $m \in M$ is given by the tuple $c = (u, u', v, w)$ with $u = g^k$, $u' = h^k$, $v = m \oplus H(y^k)$, $\alpha = H(u, u', v)$ and $w = y1^k y2^k$ where $k$ is chosen uniformly at random in $\mathbb{Z}_q$. The decryption process of $c = (u, u', v, w)$ first checks whether $u^{s1+t1}\alpha(u')^{t2+s2} = w$ with $\alpha = H(u, u', v)$ and, if so, returns $m = v \oplus H(u^k)$. Again, letting $d_{ID} = d/\sigma_{ID} \text{ (mod } q)$, we get the following ‘enhanced’ scheme

$$c = (u, u', v, w) \xrightarrow{\sigma_{ID}} \begin{cases} \alpha = H(u, u', v) \vspace{1ex} \\ u^{s1+t1}\alpha(u')^{t2+s2} \equiv w \vspace{1ex} \\ u_0 = u^{*\sigma_{ID}} \vspace{1ex} \\ v \oplus H(u_0^{*\sigma_{ID}}) \end{cases} \text{ or } \bot$$

4.3 Enhanced Cramer-Shoup decryption

Further traitor tracing schemes can be obtained similarly using other splitting techniques or other cryptosystems.

5. CONCLUSION

This paper studied the problem of securely distributing content without resorting to secure hardware. We presented general traitor tracing schemes which nicely complement the state-of-the-art software implementations. Specific applications to RSA-based and discrete-log based cryptosystems were described.

6. REFERENCES


