The RMX Transform and Digital Signatures[*]

Shai Halevi[†] Hugo Krawczyk[‡]

IBM T.J. Watson Research Center,
Hawthorne, New York 10532.

August 22, 2006

Abstract

This document describes RMX, a simple message randomization scheme that when used as a front end to existing hash-then-sign signature schemes, such as RSA and DSS, frees these signatures from their current vulnerability to off-line collision attacks on the underlying hash function. We demonstrate the practicality of the approach, which requires no change to hash functions or signature algorithms, by describing how to accomodate RMX in the context of actual applications (e.g., certificate signing, XML signatures) and existing implementations (e.g., `openssl`). In many cases, the required changes are only slightly more complex than accomodating a new (deterministic) hash function.

[*]See `http://www.ee.technion.ac.il/~hugo/rhash/` for updates.
[†]Email: shaih@alum.mit.edu.
[‡]Email: hugo@ee.technion.ac.il

1 Introduction

The recent collision attacks against popular hash functions have a profound effect on the security of some applications of these functions, most notably digital signatures. In this work we propose a *randomized mode of operation* for hash functions that when used in conjunction with standardized hash-then-sign signature schemes (such as RSA or DSS) frees these schemes from their current essential dependency on full collision resistance. This mode of operation can work with any (iterated) hash function and requires no change to the underlying signature algorithms. It consists of a *simple message randozmization transform, called RMX*, which is fully specified in this document and used as a front-end to existing hash-then-sign signature schemes. It has been proven [3] that breaking a signature scheme that uses RMX requires to solve a cryptanalytical problem which is related to finding second pre-images in the underlying compression function and hence significantly harder than just finding (off-line) collisions as in current signature schemes.

The full specification of RMX is presented in Section 2. In a nutshell, RMX prepends to the message a random string ("salt") of one block, and then XOR the same random string into every block of the message itself (where a "block" is the blocksize used by the underlying hash function). That is, if $m = (m_1, ..., m_n)$ then $RMX(m, r) = (r, m_1 \oplus r, ..., m_n \oplus r)$. We note that the detailed description of RMX includes a simple padding rule for the last message block; also, to save bandwidth and randomness, the scheme accomodates salt strings shorter than a full message block. RMX can be implemented either as a simple front-end interface to the iterated hash function (leaving the hash implementation unchanged), or it can be integrated with typical implementations of digest functions that read the message block by block and feed these successive blocks into the compression function.

RMX can be used with any hash-then-sign scheme by replacing the digest $H(m)$ in the original signature scheme with $H(RMX(r, m))$. In this case, the salt r is generated for each signature by the signer and transmitted to the verifier together with the message and signature[1]. The verifier uses the regular verification procedure where the original digest function is applied to m' rather than m. Note that only the signer needs to generate randomness, the verifier receives it with the message/signature. As said, off-line collision search is useless against a signature scheme that uses RMX. Rather, to break the signatures the attacker needs to solve a cryptanalytical problem close to finding second preimages (which is a much much harder task than finding collisions). Importantly, to gain this security advantage the value of r must be unpredictable by the attacker before receiving the signature on a given message, and therefore we recommend lengths between 128 bits to a full-block size (with 160 or 256 bits being reasonable default values).

[1]In contrast to a previous proposal by the same authors, the salt r does *not* need to be included under the signature.

The use of RMX and its application to signatures do not depend on the way the salt r is transmitted; therefore, different applications may choose different ways to transport r. This is analogous to the use of the IV in CBC encryption: the definition of CBC telss you how to use a block cipher to encrypt any-lenth message given the value of a random IV but it does not limit you to a particular way to transmit the IV. In Sections 4.2 we discuss some options for the transport of the salt in applications using RMX. In particular, we suggest a mechanism that can be shared by applications that use algorithm identifiers (as per X.509), namely, transporting the salt as a parameter of the algorithm identifier. We illustrate this approach via an implementation in the context of certificate signing and verification. As an additional case study we also describe an implementation of RMX and its use for XML signatures.

It is important to note that the use of RMX in the context of digital signatures does *not* require changes in signature standards such as PKCS#1 (RSA) or FIPS 186 (DSS). Furthermore, an important conclusion of this work is that the complexity of implementing and deploying RMX in the context of digital signatures is comparable to the effort needed to upgrade existing systems to use a new deterministic hash function, say SHA256. Moreover, once the mechanisms are in place to deal with such upgrade [1], supporting RMX becomes a relatively simple matter. See Section 4.

We stress that our proposal is not intended as an alternative to the search for new, stronger hash functions to replace SHA-1 and MD5, but it is rather intended to complement this effort by providing a "safety net" for digital signature in case a hash function in use is later found to be weaker than believed initially. Given our limited understanding of the best ways to build collision resistant hash functions, prudent engineering principles call for building cryptographic primitives that rely as little as possible on the strength of hash functions. Our work addresses this priciple in the context of digital signatures and as such it resembles the effect of the HMAC design in the message authentication area.

We refer the reader to the companion paper by these authors [3] for a more extensive description and rationale of the design of RMX, as well as an analysis of the cryptographic strength of the scheme (in that paper the scheme specified here, namely, applying RMX to the message before inputting it to the hash function, is referred to as the "eTCR construction $\{\tilde{H}_r\}$").

Organization. In Section 2 we specify the the RMX message randomization transform. In Section 3 we describe the use of RMX with digital signatures. In Section 4 we discuss the integration of RMX with the `openssl` library, and in Section 5 we describe integrating RMX with XML signatures.

2 The Message Randomization Scheme RMX

Given a message m we apply to it a randomization scheme called *RMX* (pronounced *remix*) that takes as inputs the message m and a random string r and produces an output message m'. Informally, a message m' produced by *RMX*(r, m) is defined as the concatenation of the string r (say, of the length of a hashing block) followed by the result of XOR-ing each block of the message m with the string r followed by a padding rule for the last block of m' defined specifically for the *RMX* and described below (this *RMX* padding rule is designed such that the last block of m' is of length a full block less the minimal number of bits required by the padding rule of the underlying hash function H). Following is a precise definition of *RMX*.

The *RMX* message randomization scheme for hash function H. Assume H to be a Merkle-Damgard hash function[2] with block size b (in bits), such that H adds at least $c + 1$ bits of padding and length-encoding to each message. For example, all the currently used iterated hash functions pad the input message to a multiple of b bits by appending a single '1' bit followed by as many zeros as needed (possibly none) and then followed by c bits that encode the bit-length of the input message. Typical parameters are $b = 512$, $c = 64$ for SHA1 and SHA256, and $b = 1024$, $c = 128$ for SHA512. For the specification of RMX below we assume that $b < 2^{16}$.

The function *RMX* accepts as inputs a message m of bit-length at most $2^c - b$ and a string r of length between 128 and b bits and produces an output string m' as follows:

1. It computes three strings r_0, r_1, r_2 from r as follows:
   - r_0 is a b-bit string that is obtained from r by padding it with as many zero-bits as needed.
   - r_1 is a b-bit string that is obtained as r concatenated with itself as many times as needed to cover b bits (with the last repetition of r possibly truncated).[3]
   - r_2 is set to the first $b - c - 8$ bits of r_1.

   (Roughly, r_0 will be prepended to the message, r_1 will be XORed to all the blocks except the last, and r_2 will be XORed to the last block.)

2. Parse the message m into $L - 1$ full b-bit blocks $m_1, \ldots, m_{L-1}$ and a last block m_L of length b', $1 \le b' \le b$.

3. Set $m'_0 = r_0$.

[2]The same approach can be adapted to other iterative constructions.

[3]For example, if $b = 512$ and $|r| = 160$ then r_1 consists of the concatenation of three times r and then the first 32 bits of r.

4. For $i = 1, \ldots, L-1$ set $m'_i = m_i \oplus r_1$.

5. Let ℓ be a two-octet (16-bit) string, representing the bit-length b' of m_L in big-endian notation. Namely, if ℓ_0, ℓ_1 are the first and second bytes of ℓ, respectively, and each of these bytes represents a number between 0 and 255, then $b' = \ell_1 \cdot 256 + \ell_0$.

   (a) If $b' \leq b - c - 24$ then set m^*_L as a string of $b - c - 8$ bits, obtained by concatenating m_L with as many zero-bits as needed and the 16-bit string ℓ. Namely, in this case $m^*_L = m_L | 0^k | \ell$ where $k = b - b' - 16 - c$.
   Set $m'_L = m^*_L \oplus r_2$.

   (b) If $b' > b - c - 24$ then set m^*_L as the concatenation of m_L and as many zero-bits as needed to get a full b-bit block, and set m^*_{L+1} as a string of $b-c-8$ bits obtained by concatenating as many zero-bits as needed and the 16-bit string ℓ. Namely, in this case $m^*_L = m_L | 0^{b-b'}$ and $m^*_{L+1} = 0^{b-c-24} | \ell$.
   Set $m'_L = m^*_L \oplus r_1$ and $m'_{L+1} = m^*_{L+1} \oplus r_2$.

6. Output the string m' as the concatentation of $m'_0, m'_1, \ldots, m'_L$ in case 5(a), and $m'_0, m'_1, \ldots, m'_L, m'_{L+1}$ in case 5(b).

Implementation. We discuss implementation issues in Sections 4 and 5. In particular, note that the definition of RMX allows for an implementation that acts as a simple front-end interface to the iterated hash function, or it can be integrated with typical implementations of digest functions that read the message block by block and feed these successive blocks into the compression function.

3 Building Signatures using *RMX*

The main purpose of our randomized hashing algorithm, and specifically the RMX transform, is for use with digital signatures where RMX may preserve the security of the signatures even in the presence of off-line collision attacks.

To compute a signature on a message m using the RMX transform with hash function H (e.g., SHA1, SHA2) and signature algorithm SIG (e.g., RSA or DSA) one proceeds as follows:

1. Choose a random value r as the salt for the *RMX* transform.

2. Using r and m compute a new message m' following the *RMX* message randomization scheme defined in Section 2.

3. Apply H to m'

4. Sign using algorithm SIG the value $H(m')$ to obtain a signature s.

5. Transmit the salt r, message m and signature s to the receiving side.

Note: Steps 2 and 3 are block-wise computations and can be interleaved (or pipelined). That is, there is no need to wait for the full message m' to be computed out of r and m before starting the H computation. In a typical implementation one feeds each block of m into the *RMX* computation and then feeds the resultant block of m' into the hash function H.

The verification procedure is defined similarly to the above: it receives the three elements r, m, s, computes $m' = RMX(r, m)$ and provides the (randomized) message m' and signature s to the verification procedure (as before the *RMX* and hash computations can be pipelined).

Note that the above procedure can be used with *any* signature scheme that follows the hash-then-sign paradigm including the two major signature standards: RSA (both deterministic and PSS encoding) [5] and DSS [2].

To support RMX-enabled signatures as above, an application needs to satisfy two requirements: (1) the ability of the signer (not the verifier) to generate the random (unpredictable) salt r; and (2) the ability of the application to accomodate the transmission of r. We believe that most applications meet these requirements, and even more so given the increasing capabilities of computing devices. In particular, most cryptographic applications already require the ability to generate (pseudo) random bits for key generation, IV's, nonces, or probabilistic signatures such as DSS. As for (2), a great majority of applications can afford the sending of a few extra bytes of salt in addition to a message and signature. While different applications can accomodate the sending of the salt in different ways, we discuss a general mechanism that may work for many different application in Section 4.2. A few more comments are in order here:

1. Observe that the receiver of the signature can only start to hash the message after it knows the salt. Hence, in applications where buffering the entire incoming message is impractical, it is preferable to send the salt *before* the message. In particular, in such applications one probably cannot make the salt be a component of the signature itself (since typically a signature is transmitted after the message).

Also, we stress that an application using RMX must ensure that an attacker cannot choose the message to be signed (or part of its contents) after seeing the salt. Hence the salt, even if sent before the message, will be sent to the verifier only after the message has been fully determined.

2. For extremely bandwidth-limited applications, one can sometime save on bandwidth by including the salt in the signature (even if it means sending the salt after the message). For example, with DSS signatures one can re-use the random component $r = g^k$ that already exists in the signature also as the hashing salt, thus preserving the original data size. It should be noted, however, that this means that the quantity $r = g^k$ must be computed before computing the message digest.

In the case of RSA-PSS [5] an approach similar to DSS can be used to save bandwidth (here the randomness used internally by the signature can be recovered by the recipient of the signature via the RSA-verification operation). The situation is more problematic with the deterministic RSA encoding of PKCS#1 v1.5 [5]; here the only way to preserve bandwidth is to include the salt r under the signature itself. That is, instead of applying the RSA operation solely to the result of the randomized hash operation one applies it to the concatenation of this result and the salt r. In this way, the recipeient of the signature can recover the salt via the RSA-verification procedure. This, however, requires a change in the message encoding of PKCS#1 v1.5, and hence less desirable as a general solution.

3. We end by noting that using an independent salt value has the additional advantage that it allows for the pre-computation of the randomized hash value (i.e., one can choose r, compute $d = H(RMX(r, m))$ and store the triple (r, d, m), such that upon a request for a signature on m one computes the signature directly on the pre-computed d). Also, it supports multi-level hashing which is essential in some cases, for example the XML signatures treated in Section 5.

4 Integrating RMX into `openssl`

Here we report on our experience with integrating RMX-enabled signatures into the `openssl` library. The intergation was suprisingly easy, involving writing only a few hundred lines of code. Moreover, most of that work is not specific to RMX and is needed whenever one introduces a new hash function into `openssl` (even a deterministic one). Indeed, *the RMX-specific work involved changing only four files and less than 100 lines of code!*

The integration consisted of three parts: (a) implementing the RMX transform itself (and adding the appropriate hooks for it in the library), (b) adding a "thin transform layer" to the message-digest layer of `openssl` to support a transformation before the actual hashing, and (c) modifying the library routines for signature and verification of certificates to support RMX-enabled signatures on certificates. The bulk of the work was invested in part (a), whereas only only parts (b) and (c) are unique to RMX. On a high level, parts (b) and (c) consisted of the following changes:

- In the digest layer (that mutiplexes between the different low-level hashing algorithms) we added a new interface that allows the calling routine to pass a parameter to the hashing algorithm. (This is needed to specify the salt for RMX.)

  We further modified the digest layer, making it check whether a transform is needed, and if so calling this transform (RMX in our case) before calling the low-level hashing functions. These changes involved ony two files and less than 40 lines of code.

- The `openssl` library contain code to handle signing and verifying certificates. As we said before, we think that certificate handling is one of the applications that would benefit the most from RMX-enabled signatures, and so we modified the certificate-handling code of `openssl` to support this.

  We added code to choose (or retrieve) the salt for hashing and then call the new digest-layer interface that we described above. We also added the salt as a parameter to the `AlgorithmIdentifier` in the certificate, thus enabling the signer to specify the salt and the verifier to read it off the certificate. These changes involved ony two files (one for signing and one for verifying) and about 40 lines of code in total.

The rest of this section assumes familiarity with `openssl` (and can be safely skipped by readers not interested in the specifics of the implementation). From a code-design perspective, the most important decision was how to add RMX support to the digest layer of `openssl`. Once we made that decision, the other implementation details followed quite naturally. Below we therefore begin by describing the changes that we made to the digest layer in Section 4.1, followed by the changes to the certificate handling (Section 4.2) and the RMX implementation itself (Section 4.3).

4.1 The `openssl` digest layer

The `openssl` library has a message-digest layer, providing high-level interfaces to hash functions. This layer is implemented by the source file `crypto/evp/digest.c` and the header file `crypto/evp/evp.h`. The three main interfaces of that layer are:

```
int EVP_DigestInit_ex(EVP_MD_CTX *ctx,const EVP_MD *type,ENGINE *impl);
int EVP_DigestUpdate(EVP_MD_CTX *ctx,const void *d,size_t cnt);
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx,unsigned char *md,unsigned int *s);
```

In these interfaces, the `EVP_MD_CTX` structure contains all the data that is associated with this particular instance of the hash function, and the `EVP_MD` structure contains pointers to the `openssl` functions that implement the hashing algorithm as well as some other information about that algorithm (e.g., its name, block-length, etc.).[4]

[4]See discussion of the `ENGINE` parameter later in this subsection.

The openssl library maintains a table of hashing algorithms, where each supported hashing algorithm has an entry of type EVP_MD in that table. The library also provides helper routines to search through this table (e.g., based on the algorithm name). A typical application would have a piece of code more or less as follows:

```
// hashing a message buffer ‘msg’ of length ‘len’ into the
// digest buffer ‘dgst’ using the hashing algorithm SHA1
[...]
EVP_MD_CTX ctx;  // an empty (uninitialized) context
int dgst_len;    // will be used to hold the digest size

EVP_MD *type = EVP_get_digestbynid(NID_sha1); // get algorithm details

EVP_DigestInit_ex(&ctx, type, NULL);
EVP_DigestUpdate(&ctx, msg, len);
EVP_DigestFinal_ex(&ctx, dgst, &dgst_len);
[...]
```

The call to EVP_DigestInit_ex initializes the EVP_MD_CTX structure (and in particular saves in it the pointer type), and also calls the initialization function of the low-level hashing algorithm (which would be SHA1 in this example). The later calls to Update and Final call the corresponding functions of the low-level hashing algorithm.

4.1.1 Specifying the salt

To use randomized hashing we must introduce the salt somewhere in the flow. Note that the salt typically cannot be completely transparent to the calling application, since this application may need to send it (if it uses randomized hashing for signing) or receive it (if it uses randomized hashing to verify signatures). We thus let the application explicitly provide the salt as part of the initialization. Specifically, we introduce a new interface

```
int EVP_DigestInit_ex2(EVP_MD_CTX *ctx,
                       const EVP_MD *type, ENGINE *impl, void *params);
```

An RMX-enabled application would then call this new interface (rather than calling EVP_DigestInit_ex) and pass the salt inside the params argument. If the calling routine uses RMX for signing then it needs to choose the salt by itself, and if it uses RMX to verify signatures then it needs to receive the salt from the signer.

In addition, we add a params field to the EVP_MD_CTX structure to holds the extra parameters, so that after this parameter is passed to the Init function it can be used by the Update and Final functions of the RMX transform.

4.1.2 The calling sequence

To insert the RMX processing to the flow of control in the digest layer, we add the following flag to the `flags` field of the `EVP_MD` structure.

```
#define EVP_MD_FLAG_TRANSFORM      0x0100
#define EVP_NEEDS_TRANSFORM(md)    ((md)->flags & EVP_MD_FLAG_TRANSFORM)
```

(Currently this flag identifies only the RMX transform.) Then, for each supported hashing algorithm we add another entry to the table of algorithms, pointing to the same low-level functions that implement the hashing algorithm but having the `TRANSFORM` flag set. (Also, the new entry will have a different name, e.g. `NID_rmxWithSHA1` instead of `NID_sha1`.)

The implementation of the `Init/Update/Final` functions in `crypto/evp/digest.c` looks for the `TRANSFORM` flag and if the flag is set then it calls the transform functions instead of calling directly the low-level hashing algorithm. For example, the `Update` function has the following code:

```
if (EVP_NEEDS_TRANSFORM(ctx->digest))
    return TRANS_Update(ctx, data, count);
else
    return ctx->digest->update(ctx, data, count);
```

The `TRANS_Update` does the RMX transform and then calls `ctx->digest->update`. We note that in our implementation `TRANS_Init/Update/Final` are just macros for `RMX_Init/Update/Final`, but in principle one can use this mechanism to implement a "full blown transform layer" by having the `TRANS` functions multiplex between different transformations based on information in the `EVP_MD` structure.

Alternative approaches. We remark that an alternative approach to what we did is to implement wrapper functions RMX-with-XYZ for every supported hashing algorithm XYZ. These wrapper functions would first call the RMX functions and then the low-level functions for hashing algorithm XYZ. The entry in the algorithms table corresponding to RMX-with-XYZ will point to these wrapper functions rather than to the underlying implementation of the hashing algorithm XYZ. Implementing things this way would require only minimal changes to the digest layer itself (essentially only adding the `Init_ex2` interface), but would require writing these wrapper functions for every supported algorithm.

Yet another alternative would be to replace the `TRANSFORM` flag in the `EVP_MD` structure by a flag that is provided as a parameter to `EVP_DigestInit_ex2` by the calling routine (and stored in the `EVP_MD_CTX` structure).

The signature layer. The `openssl` library has a signature layer above the digest layer, but *our implementation leaves this layer almost unchanged.* The interfaces for signing are `EVP_SignInit_ex` and `EVP_SignUpdate` that are just macros for `EVP_DigestInit_ex` and `EVP_DigestUpdate`, and the function:

```
int EVP_SignFinal(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s,
                  EVP_PKEY *pkey);
```

that calls `EVP_DigestFinal_ex` and then signs the digest. For verification we have `EVP_VerifyInit_ex` and `EVP_VerifyUpdate` that are macros for `EVP_DigestInit_ex` and `EVP_DigestUpdate`, and the function:

```
int EVP_VerifyFinal(EVP_MD_CTX *ctx, const unsigned char *sigbuf,
                    unsigned int siglen, EVP_PKEY *pkey);
```

that calls `EVP_DigestFinal_ex` and then verifies the digest against the signature. The only changes that we need to make to this layer is to define `EVP_SignInit_ex2` and `EVP_VerifyInit_ex2` as macros for `EVP_DigestInit_ex2` (in `crypto/evp/evp.h`).

Interaction with `ENGINE`s. The `openssl` library provides the ENGINEs interface to enable dynamically overriding the default implementations of the various cryptographic primitives (e.g. in order to use hardware accelerators when available). The way it works roughly is that the routine that calls `EVP_DigestInit_ex` provides via the parameter `impl` an alternative `EVP_MD` structure that points to the alternative implementation of the hash function. The `EVP_DigestInit_ex` function then stores in the current context the pointers to this alternative implementation, and subsequent calls activate the alternative implementation via `ctx->digest->init/update/final`.

With the specific way that we chose to implement RMX in the digest layer, it follows that an ENGINE cannot override the implementation of the RMX transform itself, since the code in `EVP_DigestInit/Update/Final` directly calls the functions `TRANS_Init/Update/Final` (which are just macros for `RMX_`...) with no place to overwrite these functions. The ENGINE can still implement the underlying hash function since the `RMX` functions call the underlying hash-function implementation via `ctx->digest->`$\cdots$.

One way to circumvent this (for an ENGINE that implements also the RMX transform itself) would be to supply an `EVP_MD` structure with the `TRANSFORM` flag turned off. This practice is a likely cause of bugs, however, so it is probably a good idea to disallow it (say, by having the function `EVP_DigestInit_ex2` verify that the `TRANSFORM` flag in the ENGINE is the same as the flag in the original `EVP_MD` structure and abort otherwise).

Perhaps a better way of allowing ENGINES to offer an alternative implementation of RMX would be to replace the current "thin transform layer" (that only calls the built-in RMX functions) with a real layer that calls the transform functions via pointers. Another way would be to switch to the alternative implementation via "wrapper functions" that was described above.

4.2 Handling Certificates

As we said above, we modified the certificate-handling code of `openssl` to demonstrate how an application might work with RMX-enabled signatures. Specifically, we needed to change the two functions

```
int ASN1_item_sign(const ASN1_ITEM *it, X509_ALGOR *algor1, X509_ALGOR *algor2,
                   ASN1_BIT_STRING *signature, void *asn, EVP_PKEY *pkey,
                   const EVP_MD *type);
int ASN1_item_verify(const ASN1_ITEM *it, X509_ALGOR *a,
                   ASN1_BIT_STRING *signature, void *asn, EVP_PKEY *pkey);
```

in `crypto/asn1/a_sign.c` and `crypto/asn1/a_verify.c` respectively.

Given the changes to the digest layer that we described above, modifying the certificate handling code was fairly straightforward. Specifically, the signing code needs to check if we use RMX, and if so choose a random salt and pass it to the new `SignInit_ex2` interface, and then include the salt in the certificate so that the verification routine can find it. The verification routine needs to retrieve the salt from the certificate and pass it to the new `VerifyInit_ex2` interface. The only decision to make is where to put the salt in the certificate, and we considered the following two options:

The salt as part of the signature. The simplest solution would be to include the salt as part of the signature string. This means that after calling `SignFinal`, the signing code will concatenate the signature that it gets with the salt, and includes the entire result as the signature. Notice that as opposed to the cases that were discussed in Section 3, here the entire certificate is presented for verification at once, so there is no real disadvantage to having the salt included only at the signature (which logically comes "after the certificate"). This option would require the smallest change in the certificate-handling code.

The salt as an algorithm parameter. The option that we chose to implement is slightly more complicated, but is more general and can possibly be used for applications other than certificates. Namely, it is possible to specify the salt as a parameter of the signature algorithm. Specifically, the `X509_ALGOR` structure is an implementation of the ASN.1 structure

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm OPTIONAL }
```

as defined in X.509 and RFC 3280 (PKIX). Thus it has an (optional) parameter that we can use to carry the value of the salt. In our implementation we defined the signature algorithms that use RMX (e.g., `OBJ_rmxsha1WithRSAEncryption`) to have a parameter of type OCTET STRING. The signing function contains the following code:

```
if (<this-is-RMX>)
{
    RAND_pseudo_bytes(salt, 32);      // generate a random 32-byte salt
    aprmtype1 = V_ASN1_OCTET_STRING;
    aprm = M_ASN1_OCTET_STRING_new(); // the parameter of the CA key
    M_ASN1_OCTET_STRING_set(aprm, salt, 32);
    aprmtype2 = V_ASN1_NULL;          // the parameter of the key in the cert
}
[...]
X509_ALGOR_set0(algor1, OBJ_nid2obj(signid), aprmtype1, aprm);
```

The verification code looks for an OCTET STRING parameter, and if found it uses it as salt for RMX:

```
X509_ALGOR_get0(NULL, &aprmtype, (void**)&aprm, a);
if (aprmtype==V_ASN1_OCTET_STRING)
{
    rmxprm->salt=M_ASN1_STRING_data(aprm);
    rmxprm->salt_len=M_ASN1_STRING_length(aprm);
}
```

We remark that a certificate contains two `AlgorithmIdentifier`'s, one that describes the algorithm used by the CA to sign the certificate and the other that describes the subject key (i.e. the algorithm that will be used with the public key that is contained in this certificate). Only the former algorithm has the salt as a parameter (and our implementation adds a NULL parameter to the subject key if it is to be used with RMX). This aspect may need attention when using the key in a first certificate, call it cert1, to sign other certificates, say cert2, as in the case of certificate chains. In this case the subject's algorithm identifier in cert1 will have a NULL parameter while in cert2 the issuing CA's algorithm identifier will be the same but this time with a non-NULL random OCTET STRING parameter. We did not check yet whether this causes problems with the `openssl` implementation.

Finally, we note that to actually use the RMX-enabled signatures, the `openssl` command-line program should be enhanced to allow the user to specify using RMX for signatures. In our implementation we bypassed this by making RSA-with-RMX-SHA1 be the default signature algorithm.

4.3 Implementing RMX

The implementation of RMX in `openssl` was fairly straightforward (with most of the implementation time devoted to adding the hooks for RMX into the library and modifying the various makefiles and configuration files). The three main interfaces for the RMX implementation are

```
int RMX_Init(EVP_MD_CTX *ctx, void *param);
int RMX_Update(EVP_MD_CTX *ctx, const void *data, size_t len);
int RMX_Final(EVP_MD_CTX *ctx, unsigned char *md);
```

The `Init` function expects to find the salt in the `param` argument and copies it into the given `EVP_MD_CTX` structure. Then it calls the initialization routine of the underlying hash function `ctx->digest->init` and then it calls `ctx->digest->update` with the salt padded by zeros. The `Update` function only XORs the input message with the appropriate portion of the salt and sends the result to the underlying `ctx->digest->update`. The `Final` function does the RMX padding and length concatenation, then calls `ctx->digest->update` to process the padding, and then calls `ctx->digest->final`.

The parameter c. Recall from the specification in Section 2 that the padding and length-encoding of RMX depend on the parameter c that controls the length-encoding of the underlying hash function. (Namely, when possible we pad the message with as many zero-bytes as possible while ensuring that the length-encoding of the underlying hash function does not overflow from the current block.) Hence to decide how many zero-bytes to pad we need access to the parameter c of the underlying hash function, but this interface is not provided by `openssl`.

In our current implementation we use a hack where we derive the parameter c from the block-length of the underlying hash function (which is available in `openssl` via the macro `EVP_MD_block_size`). Specifically, for block-size of 512-bits we set $c = 64$, and for block-size 1024 bits we set $c = 128$. This works for the current hash functions that are supported in `openssl` (but may not work for other hash functions).

More robust solutions to this issue are either to derive the value of c directly from the name of the underlying hash function, or add an interface to `openssl` that provides this information, or switch to the alternative implementation via "wrapper functions" that was described in Section 4.1 and have the `Final` wrapper functions pass this information to `RMX_Final`.

Hooks in `openssl`. Once we had an implementation of RMX, we needed to add the hooks for it in the library. This includes definitions of object-identifiers for RMX-MD and SIG-with-RMX-MD for every supported SIG-MD combination. This is added to the files `objects.txt` and `obj_mac.num`, which are then processed by the perl scripts `objects.pl`, `obj_dat.pl`, and `objxref.pl` (all in the directory `crypto/objects/`).

Then for every supported SIG-MD combination we had to add to the table of algorithms an entry with the static `EVP_MD` structure that has object-identifiers for RMX-MD and SIG-with-RMX-MD and pointers to the functions that implement the underlying MD hash function (and also have the `TRANSFORM` flag on). And of course we needes to change all the relevant makefiles so that all the new files that we added will be compiled and linked and also all the dynamic-library-definitions to add the new functions that we introduced.

5 Integrating RMX into XML signatures

In this section we briefly report on an ongoing implementation of the RMX transform and its use for XML signatures by Michael McIntosh from IBM Watson [4]. XML signatures already support the idea that the data can be transformed before it is signed, thus adding RMX to XML signatures is fairly simple. Roughly, one only needs to implement the RMX transform itself, and then modify the calling application as follows:

```
// Proceed as usual, including other transformations (envelope, canonicalize)
RMX = get_a_pointer_to_implementation("URI-of-RMX");
salt = call_your_favorite_RNG();
x.addTransform(RMX, salt);
// Proceed as usual with the hashing and signature
```

A minor detail that needs to be resolved is how the transform gets the information about the parameters b and c of the underlying hash function (cf. Section 2). This can be done by either the calling routine supplying these parameters (as additional parameters to the `addTransform` method) or by having a wrapper transformation (e.g., use `"URI-of-RMX-SHA1"` instead of `"URI-of-RMX"`). We also note that the RMX transform *must be the last transform* before the hash (since after XORing with the salt the data is no longer in valid XML format).

Of particular interest in this case is the fact that XML signatures use a two-level hashing scheme: A collection of one or more items is signed by transforming and hashing each item separately, then concatenating and hashing again all the digests from the first level, and finally signing the resulting digest. Hence, in XML one applies RMX to the two levels, the first using the transform described above and the second level via the “cannonicalization transform” applied by XML to the concatenation of the hashes as a last transform before signing. In other words, it is possible to supply a different canonicalization method that would include also the RMX transform. Providing the salt for this second-level hashing is a bit more tricky than for the first level, but this too can be done.

Acknowledgments. We thank Michael McIntosh, Mark Davis, Suresh Chari for their invaluable assistance with our implementation work. In particular, the RMX implementation for XML signatures as reported here is due to Mike McIntosh.

References

[1] Steven M. Bellovin and Eric K. Rescorla, "Deploying a New Hash Algorithm", NDSS'06. `http://www.cs.columbia.edu/~smb/papers/new-hash.pdf`

[2] Digital Signature Standard (DSS), FIPS 186, May 1994.

[3] Shai Halevi and Hugo Krawczyk, "Strengthening Digital Signatures via Randomized Hashing", Crypto'2006.
`http://www.ee.technion.ac.il/~hugo/rhash/`

[4] Michael McIntosh, "Implementing the RMX transform for use in XML signatures", in preparation.

[5] PKCS #1 v2.1: RSA Cryptography Standard, RSA Laboratories, June 14, 2002