CFRG
Internet-Draft
Expires: April 18, 2008

S. Halevi (IBM)
H. Krawczyk (IBM)

October 18, 2007

Strengthening Digital Signatures via Randomized Hashing
draft-irtf-cfrg-rhash-01.txt

Status of this Memo

Abstract

This document describes a randomized hashing scheme consisting of a simple message randomization transform that when used as a front-end to regular hash-then-sign signature schemes, such as RSA and DSS, frees these signatures from their current vulnerability to off-line collision attacks against the underlying hash function. The proposed mechanism can work with any hash function as-is and requires no change to the underlying signature algorithm. Incorporating this mechanism into existing applications requires changes that are comparable in their complexity to accommodating a new (deterministic) hash function such as SHA-256.

Visit http://www.ee.technion.ac.il/~hugo/rhash/ for more information and updates on this work.

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 document 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 hash function and requires no change to the underlying signature algorithms. It consists of a simple message randomization transform, called RMX, which is fully specified in this document and used as a front-end to existing hash-then-sign signature schemes. The analysis in [HK06] shows that breaking a signature scheme that uses RMX requires solving a cryptanalytical problem related to finding second pre-images in the underlying compression function, and hence significantly harder than simply finding (off-line) collisions as in current attacks against standard 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 XORs (exclusive-or) the same random string with every block of the message itself (where a "block" is of the length specified by the underlying hash function or, if such block size is not defined by the hash function, it is the length of the salt). That is, if M=(M1, ...,Mn) where the Mi's are message blocks then:

RMX(r,M) = (r, M1 XOR r, ..., Mn XOR r).

We note that the full specification of RMX includes a simple padding rule for the last message block; also, to save bandwidth and randomness, the scheme accommodates salt strings shorter than a full message block. RMX can be implemented either as a simple front-end interface to the iterated hash function, or it can be integrated with typical implementations of digest functions (in particular, Merkle-Damgard functions such as the SHA family) 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 message M in the original signature scheme with RMX(r,M), i.e., instead of SIG(H(M)) a signature is computed as SIG(H(RMX(r,M))). The salt r is generated for each signature by the signer and transmitted to the verifier together with the message and signature. The verifier uses the regular verification procedure where the original digest function is applied to RMX(r,M) rather than to 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 harder task than finding collisions, and it has to be done on a

---

per-signature basis rather than off-line). Importantly, to gain this security advantage the value of r must be unpredictable to the attacker until the full content of the message to be signed is determined; we recommend that the length of r be at least 128 bits.

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 specifies how to use a block cipher to encrypt any-length message given the value of an IV, but leaves it up to the application to decide how to transmit the IV. In Section 4 we report on some experimental implementations of RMX (openssl, NSS/Firefox, and XML signatures), and 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.

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 initial implementation experience with RMX 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, such as SHA256. Moreover, once the mechanisms are in place to deal with such upgrade (cf. [BR05]), supporting RMX becomes a relatively simple matter (see Section 4 and the references there).

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 signatures 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 principle 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 paper by these authors [HK06] 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").

COMPATIBILITY WITH NIST's RANDOMIZED HASHING PUBLICATION [NIST]. The present document is somewhat wider in scope than [NIST] which defines the same randomized hashing transform but provides less implementation guidance. Also, the specification here of the central RMX transform is slightly more general than [NIST], in particular it

allows for some hash-specific optimizations (especially for Merkle-Damgard and related hash functions). We anticipate that the final specifications of RMX in both documents (here and [NIST]) will be identical.

IETF SCOPE. This Internet Draft is offered to the IETF community for consideration for standardization and adoption in protocols whose security heavily relies on digital signatures. The need for randomized hashing is especially acute in applications (e.g., PKIX, S/Mime) where non-repudiation or third-party-verifiability are crucial; however, it is recommended also as a general way of strengthening signatures in other applications. In particular, protocols that support hash agility (newer TLS versions, for example) should make provisions for the future use of randomized hashing (see Section 4 for some implementation guidance).

Visit http://www.ee.technion.ac.il/~hugo/rhash/ for more information and updates on this work.

2 The Message Randomization Scheme RMX

The RMX message randomization procedure is the core of the randomized hashing scheme and it is specified next. RMX takes as input a message M (to be signed) and a random salt value r, and outputs M', the transformed message that will be input into the hash-then-sign signature module.

RMX should be seen as a mode of operation for hash functions, and as such it may depend on some of the parameters of the underlying hash function such as a block length or the hash function's padding mechanism. Specifically, the RMX definition uses two parameters, block_length and pad_length, that may depend on the underlying hash function.

The first, block_length, is used by RMX to determine the expansion of the input salt r into an internal salt value r'. RMX sets block_length to the block length of the underlying hash function (e.g., it will be 512 for SHA-256). However, if this hash function does not define a block length (or if this length is less that 128 bits) then block_length is set to the length of the salt value r input to RMX. (This provision for a non-blockwise hash function is due to NIST's desire [NIST] to accommodate any, possibly unconventional, hash function that may emerge in the future.)

The parameter pad_length is used by RMX to determine the number of bits with which RMX pads the input message (RMX performs an internal padding in addition to any padding that may be defined by the underlying hash function). In all cases, pad_length is computed as a function of the length of the input message M (we denote this length by |M|) and possibly also as a function of block_length.

After the description of RMX below, we present two concrete instantiations of pad_length: a generic function that does not assume

---

any structure on the underlying hash function (this is the same as the RMX mechanism defined in [NIST]), and a different instantiation that is optimized for Merkle-Damgard hash functions (where a block length is well defined). We expect the latter instantiation to be used with contemporary hash functions such as SHA-1 and SHA-2. (See discussion in Section 5 regarding the security advantages of using the second instantiation when the underlying hash function is a Merkle-Damgard iterated hash function.)

Specification of the RMX transform.

```
Parameters:

  block_length    // Length of the internal salt value r' used in the
                  // algorithm
  pad_length      // A function to determine the number L of padding
                  // bits used internally by RMX (depends on |M|).

Input:

  Message M, Salt r          // r is random of size at least 128 bits

Output:

  Randomized Message M'      // input to hash function and signature
```

1. Let r' be r concatenated with itself as many times as needed (last copy possibly truncated) to cover block_length bits

2. Let L = pad_length(|M|)

3. Define m to be a message obtained by concatenating the original message M, followed by L zeros and by two bytes containing the number L written in big-endian notation. That is:

   m = M || 0^L || L

4. Define R to be the a string of the same length as m (i.e., |M|+L+16 bits) composed by concatenating r' with itself as many times as needed (with the last copy of r' possibly truncated)

   R = (r' || r' || ... || r' || r') truncated to |M|+L+16 bits

5. Define m' to be the bitwise XOR of m and R. That is:

   m' = m XOR R

6. Define M' to be the concatenation of r' followed by m'. That is:

   M' = r' || m'

7. Output M'

NOTE: While the description of RMX above treats the input message M as one unit, actual implementations of RMX do not need to buffer the

whole message but can rather (as in a streaming scenario) read and process M block by block. The same is true for the output message M' that can be output block by block. This is particularly convenient when inputting M' into a hash function that reads its input block-by-block; in this case the RMX and hash computation can be pipelined. Also worth noting is that since the parameter (function) pad_length depends on the length of M, its value will usually be computed as part of the RMX procedure after the whole message is received.

Next, we present two possible instantiations of the parameters pad length and block_length.

GENERIC PARAMETERS. This definition can be used with any hash function regardless of its structure or iteration. It corresponds to the current randomized hashing definition in [NIST].

```
Set block_length = |r| (i.e., r'=r).
pad_length is set to:
    0              if |r'| <= 16+|M|
   |r'|-(16+|M|)   otherwise.
```

PARAMETERS FOR MERKLE-DAMGARD HASHING. This specification of the parameters is intended for Merkle-Damgard hash functions (and related iterated functions) and it provides the maximal benefits of RMX for such functions (by maximizing the number of random bits XOR-ed to the last padded block). The specification is optimized by setting r' to be of the length of a hash block. We use b to denote this block length and c to denote the number of padding bits added by the underlying hash function (for the so-called MD-strengthening). For example, b=512, c=64 in SHA-256 and b=1024, c=128 in SHA-512. Specifically, in the Merkle-Damgard case we define:

```
Set block_length to b.
Let b' = |M| mod b and b''=b'+c+24.
pad_length is set to:
   2b-b''     if b'' > b
   b-b''      otherwise
```

(The first case corresponds to the cases where the hash function adds a full new padding block and hence pad_length = (b-b')+(b-24-c); in the other case pad_length = b-(b'+c+24)).)

IMPLEMENTATION. As we already said, 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. We discuss more implementation issues in Section 4.

3 Building Signatures using RMX

The main purpose of randomized hashing, 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 (we assume pad_length and block_length are set by the hash function H):

1. Choose a random value r as the input salt for the RMX transform (the length of r may be specified by an application or by the signer itself; in all cases, it has to be in the range [128,block_length]).

2. Set M' = RMX(r,M) according to the RMX message randomization scheme defined in Section 2.

3. Apply H to M'

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

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

NOTE: For block-based iterative hash functions such as Merkle-Damgard Steps 2 and 3 are block-wise computations and can be interleaved (or pipelined). In these cases 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 standards: RSA (both deterministic and PSS encoding) [PKCS1] and DSS [DSS].

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 salt r; and (2) the ability of the application to accommodate the transmission of r. We expect most applications to meet requirement (1), 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. Moreover, note that it is only the signer that needs to generate randomness, not the verifier (see Section 5). 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 accommodate the sending of the salt in different ways, we discuss a general mechanism that may work for many different application in Section 4. A few more comments are in order here:

1. A receiver of a 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 necessary to send the salt before the message (rather than with the signature itself that is often transmitted after the message).

Also, we stress that an application using RMX must ensure that an attacker cannot choose, or influence, the contents of the message to be signed after seeing the salt. Hence the salt, even if sent before the message, will be sent to the verifier only after the message to be signed 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 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 (and kept secret until the message with which r will be used is fully determined).

In the case of RSA-PSS [PKCS1] 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 [PKCS1]; 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 recipient of the signature can recover the salt via the RSAverification procedure. This, however, requires a change in the message encoding of PKCS#1 v1.5, and hence less appealing as a general solution.

3. Using an independent salt value has the additional advantage that it allows for the pre-computation of the randomized hash value. That is, one can choose r, compute d = H(RMX(r,M)) and store the triple (r, M, d), such that upon a request for a signature on M one computes the signature directly on the pre-computed d. Also, using an independent salt value supports multi-level hashing which is required, for example, in XML signatures [RMX].

4. One cannot overstate the importance of not disclosing the value of the salt r before a message to be signed is fully determined. However, while it will be the general case that a new random string is used for each signature, there is no impediment to reusing the same r for several messages as long as r is not disclosed before all these messages are fully determined (this may be appropriate for applications, maybe a CA, that sign a batch of messages before disclosing the signatures).

4 Implementation Notes

The fact that one can use RMX while leaving the hash-then-sign module intact makes its adoption and implementation quite straight-forward. Our implementation experience shows that adding support for RMX to existing applications and software libraries entails the same complexity as adding a new deterministic hash function (e.g. SHA256).

The one issue that requires attention is the transmission of the salt r, which needs to be done at the application level. Several experimental implementations of RMX deal with this issue. These include works of Boneh and Shao [BS06] for the NSS Library and the Firefox browser, by us [RMX] for the openssl library, and by McIntosh for XML signatures [RMX]. All these projects implemented RMX-enabled X.509 certificates, and in particular needed to specify how the salt r is communicated from the signer to the verifier of the certificate. The two alternatives that were considered are as follows:

THE SALT AS PART OF THE SIGNATURE. A natural solution for transporting the salt with a signature is to append the salt to the signature itself (i.e., in this case the salt becomes an additional component of the signature string). As discussed in Section 3, this approach may require the buffering of the whole message at the sender or receiver, and hence it is less desirable as a general solution. In the specific case of certificates this buffering may be practical, in which case this transport solution is the one to require the smallest change to the certificate-handling code.

THE SALT AS AN ALGORITHM PARAMETER. The projects mentioned above chose to implement a more general, and slightly more involved, option; i.e., to specify the salt as a parameter of the signature's algorithm identifier. For example, the certificate structure as defined in X.509 and RFC 3280 (PKIX) includes an AlgorithmIdentifier, whose ASN.1 definition is as follows:

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

This structure has an (optional) parameter that can be used to carry the value of the salt. One can define the signature algorithms that use RMX (e.g., OBJ rmxsha1WithRSAEncryption) to have a parameter of type OCTET STRING. The signing function would copy the salt that it used for hashing into that parameter, and the verification code will

extract it from there. (See above references for more details.)

NOTE. A previous version of this document required to concatenate the salt r to the result of the randomized hash operation before signing; this resulted in the need to change encoding schemes for standards such as PKCS1. The current specification simplifies the scheme (and eases adoption) by dropping this requirement without jeopardizing the security.

5 Security Considerations

This whole document is about security. Here we highlight some important rules to observe, and comment on some of the analysis work that backs up the security of the proposed mechanisms.

Any application that implements digital signatures using the randomized hashing scheme described here must ensure that an attacker

cannot choose, or influence, the contents of a message to be signed after seeing the random salt. In particular, the salt must be unpredictable by the attacker before the message is determined. Consequently, the input r to the RMX function must be generated by a strong random number generator, or by a cryptographically-strong pseudo-random generator, and should be of length at least 128 bits but no more than the parameter block_length. Note that it is only the signer who generates randomness; the verifier receives it as part of the signature (or message). For example, in the important case of certificates, the CA will generate randomness to sign a certificate but verifiers of the certificate (which may be many and computationally restricted) do not need to generate randomness at all.

We note that while it would have been advantageous to specify the RMX transform independently of any parameter of the hash function (such as in the "default" specification of RMX discussed at the end of Section 2), there are substantial advantages to synchronizing the RMX parameters with those of the underlying hash function. Specifically, for block-based iterative hash functions, such as Merkle-Damgard, it is best to align the (extended) salt r' defined by RMX with a full block of the hash function; this ensures that the r' value, that occupies the first block of M', is hashed into a randomized IV before starting the processing of the message m0 (this is not the case if r' and the beginning of m0 occupy the same block). Another reason to have the length of r' (i.e., block_length) be the same as a hash block length is that it allows for a more efficient implementation of RMX in the case of iterative hash functions (as noted in Section 2).

Another hash-dependent parameter is the amount of padding RMX appends to message M, and which is determined by the function pad_length. This is best exemplified (and motivated) using Merkle-Damgard (M-D) hash functions. Recall that M-D functions specify a padding rule in which a pad of the form 100...0 is added to the end of a message followed by an encoding of the message length. When applied to a message whose length is (close to) an integral number of blocks this padding results in the addition of a full padding block by the M-D

processing. In the case of RMX, this padding block is added to M' after the RMX procedure has finished, and hence this added block is NOT randomized at all. Similarly, a message of length just slightly over an integral number of blocks will be padded by a M-D function to a full block, where only very few bits of that last block will be randomized by RMX. In these cases, the attacker has a predictable (i.e., little randomized) block to attack, thus reducing the level of security offered by the randomization scheme. We take care of these

issues, i.e., maximizing the number of bits randomized by RMX in the last block, via the function pad_length. Specifically, for the case of M-D functions, this maximization is achieved using the implementation of pad_length presented at the end of Section 2.

In general, it is best to consider the randomized hashing mechanism specified in this document as a mode of operation of hash functions. In this case, the dependency on specific parameters of the underlying hash function is natural and appropriate (similarly to the case of CBC mode where both the padding and the IV have lengths that depend on the block length of the underlying block cipher).

ANALYSIS: The randomized hashing mechanism specified here is presented and analyzed in [HK06] (in that paper this scheme is referred to as the "eTCR construction"; the RMX scheme itself, instantiated for M-D functions, is presented in the full web version of the paper). We refer the reader to that document for the mathematical analysis. In a nutshell, it is shown that finding off-line collisions against the underlying hash function (as in recent attacks) is insufficient to break RMX-based signatures. Instead, an attacker against such signatures needs to solve a cryptanalytical problem much harder than finding off-line collisions, namely, one that is equivalent to a variant of second-preimage finding (called e-SPR in [HK06]). Thus, RMX provides a "safety net" for digital signatures in the case that the hash function in use turns out to be vulnerable to collision attacks. (It may be worth pointing out that if a hash function is collision resistant then the same hash function used with RMX is also e-SPR; hence, RMX schemes can only add security to a signature scheme, never decrease it.)

Acknowledgments.

We thank Michael McIntosh, Dan Boneh and Weidong Shao for their RMX implementation work, and Mark Davis and Suresh Chari for their assistance with our own implementation. We are also indebted to Quynh Dang for documenting RMX for NIST and for many fruitful discussions on the subject.

References

[BR05] Steven M. Bellovin and Eric K. Rescorla, "Deploying a New Hash Algorithm", NDSS'06. http://www.cs.columbia.edu/~smb/papers/new-hash.pdf

[BS06] Dan Boneh and Weidong Shao, "Randomized Hashing for Digital Certificates: Halevi-Krawczyk Hash, An implementation in Firefox". http://crypto.stanford.edu/firefox-rhash/

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

[HK06] Shai Halevi and Hugo Krawczyk, "Strengthening Digital Signatures via Randomized Hashing", Crypto'2006. http://www.ee.technion.ac.il/~hugo/rhash/

[RMX] Shai Halevi and Hugo Krawczyk, "The RMX Transform and Digital Signatures". http://www.ee.technion.ac.il/~hugo/rhash/.

[NIST] "Randomized Hashing Digital Signatures", NIST Special Publication SP 800-106, Draft, July 2007

[PKCS1] PKCS #1 v2.1: RSA Cryptography Standard, RSA Laboratories, June 14, 2002

Full Copyright Statement