Forward Security in Private-Key Cryptography*

Mihir Bellare† Bennet Yee‡

November 2000

Abstract

The damage caused by key-exposure can be mitigated by employing forward-security. This has been common practice in the design of pseudorandom number generators. The motivation of this paper is to return to this basic practice and provide a rigorous analysis of it, including definitions, constructions and proofs in the style of reduction-based modern cryptography. We then broaden the investigation to look at forward-security in the more general context of symmetric-key cryptography, namely for primitives like symmetric encryption or message authentication codes. We apply this to the problem of maintaining secure access logs in the presence of breakins.

Keywords: Pseudorandom number generators, forward security, message authentication, proofs of security, audit logs.

*This paper was previously titled "Forward-secure pseudorandom number generators and applications." The first version of this paper dates to 1997.

†Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-Mail: mihir@cs.ucsd.edu. URL: http://www-cse.ucsd.edu/users/mihir/. Supported in part by NSF CAREER Award CCR-9624439 and a 1996 Packard Foundation Fellowship in Science and Engineering.

‡Dept. of Computer Science & Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: bsy@cs.ucsd.edu. URL: http://www-cse.ucsd.edu/users/bsy. Supported in part by a 1996 National Semiconductor Faculty Development Award, the U. S. Postal Service, NSF CAREER Award CCR-9734243, and a gift from Microsoft Corp.

Contents

1 Introduction

Today we can have more confidence in the strength of our cryptographic algorithms than in the security of the systems that implement them. Thus, in practice, the greatest threat to the security we may hope to obtain from some cryptographic scheme may simply be that an intruder breaks into our system and steals some information that will compromise the scheme, such as an underlying key.

Once the key is exposed, future uses of it are compromised: the best hope is that the intrusion is detected and we change keys. However key loss carries another danger: namely, the security of past uses of the key are compromised. For example data might have been encrypted under this key some time back, and the ciphertexts might be in the hands of the adversary. Now that the adversary obtains the decryption key, it can decrypt the old ciphertexts.

The goal of forward security is to protect against this kind of threat. One endeavors to make sure that security of past uses of a key, in whatever context, is not compromised by loss of some underlying information at the present time.

Forward security seems to have first received explicit attention in the context of session key exchange protocols [16, 12], where it is now a common requirement. More recently, there have been several proposals for forward-secure digital signature schemes [3, 6, 1, 2, 22]. However, the idea of forward-security can be traced further back. It is common practice to design pseudorandom number generators that on each iteration produce a new seed and delete the old one, so that someone breaking into the system and learning the current seed cannot compromise the pseudorandom bits associated with past usage of the generator. Even though formalizations were not provided, the idea of forward-security can clearly be seen to be present.

The motivation of this paper is to return to this basic practice and provide a rigorous analysis of it, including definitions, constructions and proofs in the style of reduction-based modern cryptography. We then broaden the investigation to look at forward-security in the more general context of symmetric-key cryptography, namely for primitives like symmetric encryption and message authentication codes. We apply this to the problem of maintaining secure access logs in the presence of breakins.

The threat of key compromise has been addressed by other means. One is via distribution of the key across multiple machines. Forward-security, in contrast, is a single-machine solution. Another, less cryptographic mechanism is embodied in systems such as the those incorporating FIPS 140-1 certified modules [20]. These protect against key compromise by the use of physical security and tamper detection to guarantee key erasure. Forward security is a method for providing many of the same security properties via software means.

1.1 Forward security in pseudorandom number generation

Random values need to be generated in many cryptographic implementations. For example, session key exchange protocols need to generate session keys. Many signature schemes, such as DSS, require generation of a random value with each signature. Another example is random nonces used as salt in encryption, in particular for probabilistic public key encryption schemes. In these applications, if the random value in question is exposed, even after some time, security is compromised. Data encrypted under the session key might be exposed. In DSS, exposure of the random value yields the secret signing key to an attacker. If the salt is exposed, the ciphertext can be decrypted.

Often, random values are generated from some seed via a pseudorandom number generator. With the threats of exposure discussed above, it is possible this seed will get exposed over time. At this point, all the past pseudorandom values fall into the hands of the adversary.

To prevent this, we want a generator that is forward secure. This generator will be stateful. At each invocation it produces some output bits r as a function of the current state, updates the state, and then deletes the old state. An adversary breaking in at any point in time gets only the current state. The generator is designed so that it is infeasible to recover any previous state or previous output block from the current state. So if the output blocks were used for encryption as above, an adversary breaking in would still be unable to decrypt ciphertexts created earlier.

Our formal notion of forward security for generators actually requires something stronger: the sequence of output blocks generated prior to the break-in must be indistinguishable from a random, independent sequence of blocks even if the adversary holds the current state of the generator. This is a strengthening of the notion of security for standard (i.e. not forward secure) generators that was given by [8, 23].

Our main construction is a way to transform any standard (stateless, not forward secure) pseudorandom generator into a forward secure pseudorandom number generator. The construction is quite simple and cheap. Furthermore we prove that if the original generator is secure in the sense of [8, 23] then our constructed one is secure in the strong sense we define.

We also suggest a design (of forward secure pseudorandom number generators) based on block ciphers which is proven secure assuming the block cipher is a pseudorandom function family. An attractive feature of this design is that the pseudorandom functions need be secure against only a very small number of queries and yet we can generate many pseudorandom blocks.

We also explore the forward security of existing primitives that might qualify or be easily modified to qualify as forward secure generators. An example is the alleged-RC4 stream cipher, which is a stateful pseudorandom generator, and thus a candidate for a forward secure generator. But we show that it is not forward secure because its state update process is easily reversible. We then look at known number-theoretic constructions like those of [8, 7]. These are not presented as stateful, but from the underlying construction one can define an appropriate state and then use the results of the works in question to show that the generators are forward secure. In fact this extends to any generator using the iterated one-way permutation based paradigm of generation that was introduced in [8].

These number-theoretic constructions are however slower than the ones discussed above. The suggested construction of a forward secure pseudorandom number generator is either the block cipher based one discussed above or one formed by taking a fast and seemingly secure normal generator and applying the transformation discussed above.

While forward security via key evolution in stream ciphers or pseudo-random number generators has been "part of the lore", to our knowledge our scheme is the first concretely analyzed construction.

1.2 Forward integrity

In a normal message authentication scheme, data can be MACed and the MAC value verified based on a single common key shared between the users. Forward security may at first glance seem irrelevant (or already implicitly present) for MACs, because if, say a session key being used for data authentication is exposed after the session is over, no harm ensues. But authentication of data across a network is not the only type of usage of MACs. Forward security is relevant in settings like the usage of MACs for secure audit logs, as discussed in [21]. An attacker breaking into a machine currently can modify log entries relating to the past, erasing for example a record of his previous (unsuccessful) attempts at break-in. To prevent this we use a forward secure message authentication scheme to MAC log entries as they are made by the system. Sequence and other information is included to prevent re-ordering and deletion. A description of such a secure audit

log system is in Section 3.

We can use forward pseudorandom number generators to build provably secure forward secure MACs. A forward secure message authentication scheme (in analogy to the case of generators) is stateful: the key is modified, or updated, periodically. The operation of the scheme is divided into stages $i = 1, 2, \ldots$ and in each stage the parties use the current key K_i for MACing and MAC verification. At the end of the stage K_i is updated to K_{i+1} and K_i is deleted. An attacker breaking in gets the current key. The desired security property is that given the current key K_i it is still not possible to forge MACs relative to any of the previous keys $K_1, \ldots, K_{i-1}$.

The parties can agree on some convention as to how frequent the key updates should be. For example, maybe once every minute. The frequency reflects their feelings about the likelihood of break-ins: if fast, automated break-ins are considered more likely the updates should be more frequent. Our unoptimized implementation can update keys at a rate of once every 100mS without noticeably increasing system load.

We construct a forward secure message authentication scheme using a forward secure generator and a normal message authentication scheme. This is in fact a general paradigm which we also works for encryption. Our construction is proven to have the forward security property assuming the two primitives it uses are themselves secure.

1.3 Forward secure symmetric encryption

We design forward secure encryption schemes based on the same paradigm used for MACs. Namely a forward secure pseudorandom number generator, and a symmetric encryption scheme secure in the standard sense of [4], are combined to yield a forward secure encryption scheme. Here the adversary breaking in at some point in time gets the current key but still remains incapable of decrypting data encrypted under the key of any previous stage.

1.4 Related work

A well-studied paradigm to reduce the risk of key exposure is distribution of the key via secret sharing. Specific approaches include threshold cryptography (see the survey [11]) and proactive secret sharing [19]. (In particular a proactive, distributed pseudorandom number generator has been designed by Canetti and Herzberg [9].) But distribution is costly. It might be a good option for (say) the secret signing key of a certification authority since the latter has the resources to invest in multiple machines. But it is hardly an option for an average user. We would like to see what protective measures we can take in a single machine environment. Forward security is one option.

Furthermore note that distribution does not (necessarily) provide forward security, in the sense that if at some point in time a greater than the allowed threshold number of servers is corrupted, past secrets might be revealed. Thus one could consider designing "threshold (or proactive) forward secure" schemes, where a certain threshold of players must be corrupted to compromise security, but where even if more are corrupted, security of the past is not compromised.

An application of forward-secure pseudorandom generators to the design of forward-secure digital signature schemes is given by Krawczyk [18]. (Our work precedes his but has been delayed in publication.)

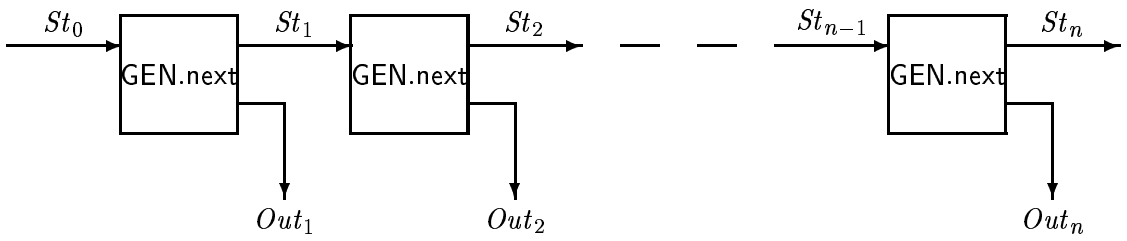

Figure 1: **Forward secure generator operation:** A sequence $Out_1, \ldots, Out_n$ of pseudorandom blocks is produced starting from an initial seed St_0.

2 Forward secure pseudorandom number generators

We first specify how forward secure generators operate and provide a formal notion of security. Then we explore constructions.

STATEFUL GENERATORS. A *stateful generator* takes as input the current state, and returns two things: an output block (consisting of a pseudorandom bits to be used by some overlying application) and an updated state (to be saved for the next invocation of the generator). A key generation algorithm, run only at the very beginning, is responsible for producing the initial state. So formally the generator $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ is a pair of algorithms: the (probabilistic) *key generation* algorithm $\mathsf{GEN.key}$ which produces the initial state (also called the seed) and the (deterministic) *next step* algorithm which given the current state returns a pair consisting of an output block and the next state. We can get a sequence $Out_1, Out_2, \ldots$ of pseudorandom blocks by first picking a seed $St_0 \leftarrow \mathsf{GEN.key}$ and then iterating: $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$ for $i \geq 1$, as depicted in Figure 1.

We can imagine the generation process as application controlled: whenever the application needs another block of pseudorandom bits it makes a request, at which point the generator is run upon the current state to produce the needed bits, and the new state is saved.

We think of St_{i-1} as being the "key" or "seed" at time i. Forward security will require that this key is erased as soon as the next one has been generated, so that someone breaking into the machine gets only the current key.

FORWARD SECURITY. A normal pseudorandom generator is said to be secure if its output (on a hidden, random seed) is computationally indistinguishable from a random string of the same length [8, 23]. For forward security of a stateful generator, more is required. The adversary may break in at some point and learn the current state of the generator. At that point, the future output of the generator certainly cannot be secure. We require however that the bits generated in the past still be secure, in the sense of being computationally indistinguishable from random bits. (This implies in particular that it is computationally infeasible to recover the previous state from the current state). One can check that forward security implies the usual notion of generator security of [8, 23], because the latter corresponds to an adversary that never uses its break-in option.

We allow the adversary to choose, dynamically, when it wants to break in, as a function of the output blocks seen so far. Thus, the adversary is first run in a "find" stage where it is fed output blocks, one at a time, until it says it wants to break in, and at that time is returned the current state. Now, in a "guess" stage, it must decide whether the output blocks it had been fed were really outputs of the generator, or were independent random bits. We capture this below by considering two experiments, the "real" experiment (in which the output blocks come from the generator) and the "random" experiment (in which the output blocks are random strings). Notice that in both cases, the state advances properly with respect the operation of the generator.

We use the following notation for the adversary. $A(\mathsf{find}, Out, h)$ denotes A in the find stage, taking an output block Out and current history h and returning a pair (d, h) where h is an updated history and $d \in \{\mathsf{find}, \mathsf{guess}\}$. This stage continues until $d = \mathsf{guess}$.

Experiment Real(A, GEN)

$St_0 \leftarrow \mathsf{GEN.key}$
$i \leftarrow 0\,;\ h \leftarrow \varepsilon$
Repeat
  $i \leftarrow i + 1$
  $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$
  $(d, h) \leftarrow A(\mathsf{find}, Out_i, h)$
Until $(d = \mathsf{guess})$ or $(i = n)$
$g \leftarrow A(\mathsf{guess}, St_i, h)$
Return g

Experiment Random(A, GEN)

$St_0 \leftarrow \mathsf{GEN.key}$
$i \leftarrow 0\,;\ h \leftarrow \varepsilon$
Repeat
  $i \leftarrow i + 1$
  $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$
  $Out_i \leftarrow \{0,1\}^b$
  $(d, h) \leftarrow A(\mathsf{find}, Out_i, h)$
Until $(d = \mathsf{guess})$ or $(i = n)$
$g \leftarrow A(\mathsf{guess}, St_i, h)$
Return g

Let $\mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real})$ denote the probability that Experiment Real(A, GEN) returns 1 and let $\mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random})$ denote the probability that Experiment Random(A, GEN) returns 1. Let $\mathsf{Adv}(A, \mathsf{GEN}) = \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real}) - \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random})$ be the advantage of A. Finally let

$$\mathbf{FwInSec}(\mathsf{GEN}, t, n) = \max_A \{\mathsf{Adv}(A, \mathsf{GEN})\}\ ,$$

the maximum being over all adversaries A that run in time t. This is the maximum likelyhood of the forward security of the generator being broken in time t. Read it as "forward insecurity" of the generator. Our goal will be to upper bound this quantity.

ALLEGED-RC4. Some stream ciphers like alleged-RC4 are stateful generators. At each invocation, alleged-RC4 uses an existing table and two table indices to return some pseudorandom bits, and then updates its table and indices, so the table and the indices function as the generator state. One can ask whether this stateful generator has the forward security property. It turns out that it does not. We present an attack demonstrating this.

We start by expressing alleged-RC4 in our stateful generator notation. Here, the state variable St is the tuple (s, x, y), where s is a 256-element table encoding a permutation $\phi \in S_{256}$, and x and y are indices into this table. (View s as a map of 8 bits to 8 bits, and x, y as 8 bits long.)

Algorithm $\mathsf{ARC4.next}((s, x, y))$
  $x \leftarrow x + 1 \bmod 256$
  $y \leftarrow y + s[x] \bmod 256$
  Swap $s[x], s[y]$
  Return $(s\,[s[x] + s[y] \bmod 256]\,, (s, x, y))$

Unfortunately, the state updates are extremely simple and completely reversible. Below we describe **Anti-RC4**, which, given the current state, will run alleged-RC4 backwards, recovering the previous state of the generator as well as the corresponding alleged-RC4 output. This shows that alleged-RC4 is not forward secure. Actually it is a much stronger attack than required by our forward security definition, since it recovers the previous states rather than just distinguishing the output of alleged-RC4 from a sequence of random bits.

Algorithm $\mathsf{AntiRC4}((s, x, y))$
  $z \leftarrow s\,[s[x] + s[y] \bmod 256]$
  Swap $s[x], s[y]$

$y \leftarrow y - s[x] \bmod 256$
$x \leftarrow x - 1 \bmod 256$
Return $(z, (s, x, y))$

Here z is the output of the previous state, and (s, x, y) in the output of AntiRC4 is the previous state.

MAKING NORMAL GENERATORS FORWARD SECURE. There are many existing pseudorandom generators which stretch a short seed into a longer pseudorandom sequence. These generators are not (necessarily) stateful, let alone forward secure. We show how to build out of such a generator a new, stateful generator, which has the forward security property as long as the original generator was secure in the standard sense.

Construction 2.1 Let G: $\{0,1\}^k \rightarrow \{0,1\}^{b+k}$ be a normal pseudorandom number generator, secure in the sense of [8, 23], which takes a seed of length k and returns a string that is longer than the seed by b bits. We define a stateful generator $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ as follows. The state is a k-bit string.

Algorithm GEN.key	Algorithm GEN.next(St)
$s \leftarrow \{0,1\}^k$	$r \leftarrow G(St)$
Return s	Return $([r]_{1..b}, [r]_{b+1..b+k})$ ∎

We will show that this stateful generator is forward secure as long as the given generator G met the standard notion of security of pseudorandom generators of [8, 23]. To state the result we first need to recall the latter. We adapt the notion of security of a standard pseudorandom number generator as per [8, 23] to a concrete security setting. Let D be a distinguishing algorithm that given a $b+k$ bit string $x \,\|\, y$ returns a bit. Consider the following experiments:

Experiment Real(D, G)	Experiment Random(D, G)
$s \leftarrow \{0,1\}^k$; $x \,\|\, y \leftarrow G(s)$	$x \,\|\, y \leftarrow \{0,1\}^{b+k}$
$g \leftarrow D(x \,\|\, y)$	$g \leftarrow D(x \,\|\, y)$
Return g	Return g

We let $\mathsf{Succ}(D, G, \text{Real})$ be the probability that experiment Real(D, G) returns 1 and we let $\mathsf{Succ}(D, G, \text{Random})$ be the probability that experiment Random(D, G) returns 1. We let $\mathsf{Adv}(D, G) = \mathsf{Succ}(D, G, \text{Real}) - \mathsf{Succ}(D, G, \text{Random})$ denote the advantage of G and we let

$$\mathbf{InSec}(G, t) \;=\; \max_{D}\{\mathsf{Adv}(D, G)\} \;,$$

the maximum being over all adversaries D that run in time t. This is the maximum likelihood of the security of the generator being compromised in time t. Read it as "insecurity" of the generator. Our assumption is that G is secure so this quantity is small. We will seek to upper bound the forward insecurity of GEN in terms of the insecurity of G. We show that if the original generator G is secure in the standard sense described above, then our construction yields a forward secure generator. The proof of the following is in Appendix A.1.

Theorem 2.2 Let G: $\{0,1\}^k \rightarrow \{0,1\}^{b+k}$ be a pseudorandom number generator. Then the stateful generator GEN constructed based on G as described in Construction 2.1 is a forward secure pseudorandom number generator with

$$\mathbf{FwInSec}(\mathsf{GEN}, t, n) \;\leq\; 2n \cdot \mathbf{InSec}(G, t') \;,$$

where $t' = t + O(n \cdot (b + k + \text{Time}(G)))$. ∎

Here Time(G) is the time to make one computation of the base generator G on a string of length k.

CONSTRUCTION USING BLOCK CIPHERS. The existence of many high-quality block ciphers makes these primitives a natural starting point. A forward secure pseudorandom number generator can be designed quite easily given a block cipher, by first using the block cipher to get a generator G and then using the above. This turns out to be good for cost and security, and is the preferred construction we suggest. Let us specify the construction in more detail and then discuss security.

Construction 2.3 Let F: $\{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$ be the block cipher, having key size k and block size b. Define G: $\{0,1\}^k \to \{0,1\}^{b+k}$ as follows: $G(x)$ is the first $b+k$ bits of the sequence $F(x,1) \,\|\, \cdots \,\|\, F(x, 1+\lceil k/b \rceil)$. (Here we are using x as the key, and the integers input to the cipher are interpreted as b-bit strings in some natural way.) We apply the Construction 2.1 to this standard pseudorandom number generator to get a stateful pseudorandom number generator GEN. ▮

The above construction of GEN is quite efficient, using only $1 + \lceil k/b \rceil$ applications of the block cipher to implement one step of the forward secure generator (which yields one b-bit output block and an updated state). For example with DES we need 2 DES computations at each iteration.

An attractive feature of the block cipher based construction is that the security requierements on the block cipher are low in the sense that as a pseudorandom function or permutation family, it need resist only a small number of queries, much fewer than the number of output blocks the generator can generate. To bring this out we state the theorem below.

Theorem 2.4 Let F: $\{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$ be a pseudorandom function family. Then the stateful generator GEN constructed from F as described in Construction 2.3 is a forward secure pseudorandom number generator with

$$\mathbf{FwInSec}(\mathsf{GEN}, t, n) \;\le\; 2n \cdot \mathbf{InSec}(F, q, t') \;,$$

where $q = 1 + \lceil k/b \rceil$ and $t' = t + O(n \cdot (b + k + \mathrm{Time}(F)))$. ▮

Here Time(F) is the time to make one computation of the pseudorandom function on a k-bit key and b-bit input block. We are measuring the security of F as a pseudorandom function as in [5], which in turn is a quantified version of the original notion of [13]. (Specifically $\mathbf{InSec}(F, q, t')$ represents the maximum possible advantage obtainable by an adversary in distinguishing whether its given oracle is a random function or the cipher under a random key when it makes at most q queries to its oracle and runs in time at most t'.) The feature of the above result we highlight is that q is very low; as we said above, for DES it is only two. The security of block ciphers as pseudorandom functions for such a low value of the number of queries is very high: exhaustive key search is the best known attack. This means it is reasonable to assume $\mathbf{InSec}(F, q, t') \approx t'/2^k$, so that n above can be quite high.

NUMBER-THEORETIC CONSTRUCTIONS. It turns out that existing number-theory based pseudorandom number generators such as the Blum-Micali [8] and Blum-Blum-Shub [7] generators can be modified so that they are forward secure. More generally, this is true of any generator using the paradigm introduced by [8] in which the seed is an input to an injective one-way function, and the output bits are obtained by iteration of the function, dropping one hard-core bit into the output at each iteration. To exemplify this let us look more closely at the Blum-Blum-Shub generator. It is based on repeated squaring of an initial value x modulo a composite N of hidden factorization. We can make it stateful as described below.

```
Algorithm BBS.next((x, N))
   Return (Parity(x), (x^2 mod N, N))
```

Here Parity(x) is 0 if x is even and 1 if x is odd. This is known to be a hardcore bit for the squaring function modulo a composite as long as factoring is hard.

Note that in the original BBS generator one might elect to keep the factorization of N as part of the seed, and use it in computing the outputs of the generator. This is useful for performance reasons: doing the squaring modulo the primes and then using Chinese Remainders is significantly faster than doing the squaring modulo N directly. However this is not an option with the forward secure version. Had the factorization been part of the state, forward security would have been compromised: computing square roots modulo primes is easy. As it is we can only have the modulus is in the state.

Forward security of the modified BBS generator can be easily proven based on the results of [7].

3 Forward secure MACs

We first specify how a forward secure message authentication scheme operates and provide a formal notion of security. Then we provide a construction and finally discuss the secure audit log application. Note that in secure audit logs, we use MACs not for message interchange, but rather as a cheap secret-key "signature" scheme for authenticating the audit data.

THE NOTION. In a standard message authentication scheme, there is a way to tag messages and a way to verify tags based on a single, shared key between the users. To achieve forward security the scheme must operate differently: as with the generators, it must be stateful. Periodically the key is updated, and the old key deleted. Message authentication is performed with the current key, which plays the role of the current state. Thus a *stateful MAC scheme* $\mathsf{MAC} = (\mathsf{MAC.key}, \mathsf{MAC.tag}, \mathsf{MAC.vf}, \mathsf{MAC.update})$ consists of four algorithms. $\mathsf{MAC.key}$ is run to obtain the initial key (state) K_0. The operation of the scheme is divided into stages $i = 1, 2, \ldots$, and in stage i the parties use key K_i. The key at any stage is obtained from the key at the previous stage via the update algorithm: $K_i \leftarrow \mathsf{MAC.update}(K_{i-1})$. (After the update, K_{i-1} should be deleted so that it is no longer available to an attacker who might break in.) Within stage i, the parties can generate a tag (MAC) for message M via $\langle \tau, i \rangle \leftarrow \mathsf{MAC.tag}(K_i, M)$. (Notice that the stage number i is always a part of the tag; this is in order to tell the verifier which key to use for verification. Also notice that in order to put i in the tag, its value must be obtainable from K_i, and indeed we will always make sure K_i contains i.) To verify $(M, \langle \tau, i \rangle))$ we compute $\mathsf{MAC.ver}(K_i, M, \tau)$ and check that this value is 1. Some number q of messages can be authenticated in each stage; the value of q is a parameter of the scheme. The key generation algorithm is probabilistic; the tagging algorithm might be probabilistic; the others are deterministic.

The parties can agree on some convention as to how frequent the key updates should be. For example, maybe once a day. The frequency reflects their feelings about the likelihood of break-ins: if break-ins are considered more likely the updates should be more frequent. But it also gives a means of extending the lifetime of the MAC via re-keying. After a certain number of message have been MACed under K_{i-1}, it might be advisable to change keys, since the security of the MAC under a given key is limited.

The scheme must withstand forgery relative to past keys even if the adversary has broken in and obtained the current key. We allow an adaptive chosen-message attack under which the adversary can first obtain valid MACs of messages of its choice under whatever key the users happen to be using in the current stage, and then based on this decide when to break in. At the point it breaks in, it is returned the current key and then it wins if it can forge a new message relative to any previous key. The experiment measuring the success of an adversary F is described fully below:

Experiment Forge(F, MAC)

$K_0 \leftarrow \mathsf{MAC.key}$; $i \leftarrow 0$; $h \leftarrow \varepsilon$
`Repeat`
  $i \leftarrow i + 1$; $K_i \leftarrow \mathsf{MAC.update}(K_{i-1})$
  $(d, h) \leftarrow F^{\mathsf{MAC.tag}(K_i, \cdot)}(\mathsf{find}, h)$
`Until` $(d = \mathsf{forge})$ or $(i = n)$
$(M, \langle \tau, j \rangle) \leftarrow F(\mathsf{forge}, K_i, h)$
`If` $\mathsf{MAC.vf}(K_j, M, \tau) = 1$ and $1 \le j < i$ and M was not queried of $\mathsf{MAC.tag}(K_j, \cdot)$
  `then return` 1 **`else return`** 0

The forger F runs first in a find stage where it gets an oracle for the tagging algorithm under the current key. At the conclusion of a stage it may decide to output $d = \mathsf{forge}$ thereby saying it is ready to break-in. At that point it is given K_i. Run in its forge stage it now returns a pair $(M, \langle \tau, j \rangle)$, and wins if τ is a valid tag for message M under K_j. Of course it only wins if $j < i$ and also if it had never asked previously for the tag of M under K_j. The input h is a history used by F to maintain information across its own invocations; it might, for example, choose to record the outcome of its oracle queries here for use in the next stage.

We denote by $\mathsf{Succ}(F, \mathsf{MAC})$ the probability that the outcome of experiment $\mathrm{Forge}(F, \mathsf{MAC})$ is 1. We then let

$$\mathbf{FwInSec}(\mathsf{MAC}, q, t, n) = \max_F \{\mathsf{Succ}(F, \mathsf{MAC})\} ,$$

the maximum being over all adversaries F that run in time t and make at most q queries *in each stage*. (So the total number of queries made can reach qn.) This is the maximum likelyhood of the forward security of the message authentication scheme MAC being compromised by an adversary using the indicated resources. Read it as "forward insecurity" of the message authentication scheme.

A FORWARD SECURE MAC SCHEME. We design a forward secure message authentication scheme using a standard message authentication scheme and a forward secure pseudorandom number generator. The base key K_0 for the MAC scheme is a seed (initial state) of the generator. The key for stage i of the MAC scheme will be a triple $K_i = (i, k_i, St_i)$ consisting of the value i indicating the stage for which this is the key, an actual mac key k_i to be used with the standard scheme, and state information St_i for the generator, based on which the next key will be generated by iteration of the generator. We will show that this scheme is forward secure given that the standard message authentication scheme is secure in the standard sense and the generator is forward secure in the sense of Section 2.

Construction 3.1 Let us now specify the construction in full. Let $\mathsf{mac} = (\mathsf{mac.key}, \mathsf{mac.tag}, \mathsf{mac.vf})$ be a standard message authentication scheme consisting of key generation, tagging and verifying algorithms [5]. Say the key length for this scheme is k. Let $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ be a forward secure generator with blocksize k. (Meaning in one iteration it produces a pseudorandom block of $b = k$ bits.) Then we define a stateful message authentication scheme $\mathsf{MAC} = (\mathsf{MAC.key}, \mathsf{MAC.tag}, \mathsf{MAC.vf}, \mathsf{MAC.update})$ as follows:

`Algorithm` MAC.key
  $St_0 \leftarrow \mathsf{GEN.key}$
  `Return` $(0, \varepsilon, St_0)$
	`Algorithm` $\mathsf{MAC.tag}((i, k, St), M)$
	  $\tau \leftarrow \mathsf{mac.tag}(k, M)$
	  `Return` $\langle \tau, i \rangle$

`Algorithm` $\mathsf{MAC.vf}((i, k, St), M, \tau)$
  $d \leftarrow \mathsf{mac.vf}(k, M, \tau)$
  `Return` d
	`Algorithm` $\mathsf{MAC.update}((i, k, St))$
	  $(k, St) \leftarrow \mathsf{GEN.next}(St)$
	  `Return` $(i + 1, k, St)$

Recall that GEN.next returns a pair consisting of a pseudorandom output block (here k bits long) and an updated state; the above is saying the pseudorandom block becomes the effective new key for the underlying mac. ▌

To discuss the security of the construction we first need to briefly recall the standard notion of security for the underlying message authentication scheme.

SECURITY OF A STANDARD MAC. We recall the notion of MAC security of [5], which in turn is a concretization and adaptation of the notion of security of digital signatures from [15]. An adversary f attacking $\mathsf{mac} = (\mathsf{mac.key}, \mathsf{mac.tag}, \mathsf{mac.vf})$ gets an oracle for $\mathsf{mac.tag}(k, \cdot)$ so that it can mount a chosen-message attack. Finally it outputs a pair (M, τ) and wins if this is a valid forgery. That is, consider

Experiment $\mathrm{Forge}(f, \mathsf{mac})$

$k \leftarrow \mathsf{mac.key}$
$(M, \tau) \leftarrow f^{\mathsf{mac.tag}(k,\cdot)}(\mathsf{find})$
`If` $\mathsf{mac.vf}(k, (M, \tau)) = 1$ and M was not queried of $\mathsf{mac.tag}(k, \cdot)$
  `then return` 1 `else return` 0

Let $\mathsf{Succ}(f, \mathsf{mac})$ denote the probability that the above experiment returns 1 and let

$$\mathbf{InSec}(\mathsf{mac}, q, t) \;=\; \max_{f}\{\mathsf{Succ}(f, \mathsf{mac})\}\ ,$$

the maximum being over all forgers f that run in time t and make up to q chosen-message queries of the $\mathsf{mac.tag}(k, \cdot)$ oracle. This is the maximum likelihood of the security of the message authentication scheme being compromised in time t. Read it as "insecurity" of the scheme. Our assumption is that mac is secure so this quantity is small. We will seek to upper bound the forward insecurity of MAC in terms of the insecurity of mac.

SECURITY OF OUR CONSTRUCTION. We claim that the stateful message authentication scheme we constructed above is forward secure as long as the underlying message authentication scheme is secure in the standard sense and the generator is forward secure. The proof of the following is in Appendix A.2.

Theorem 3.2 Let $\mathsf{mac} = (\mathsf{mac.key}, \mathsf{mac.tag}, \mathsf{mac.vf})$ be a standard message authentication scheme with key size k, and $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ a forward secure generator with blocksize k. Let MAC be the stateful message authentication scheme formed out of these two as above. Then for any q

$$\mathbf{FwInSec}(\mathsf{MAC}, q, t, n) \;\le\; \mathbf{FwInSec}(\mathsf{GEN}, t_1, n) + n \cdot \mathbf{InSec}(\mathsf{mac}, q, t_2)\ ,$$

where $t_1 = 2t + O(n + k)$ and $t_2 = 2t + O(n + k)$. ▌

Notice that the forward secure scheme can authenticate up to qn messages (q per stage) even though the base scheme could only handle q; this is an added advantage of the construction.

FORWARD SECURE AUDIT LOGS. Computer audit logs contain descriptions of noteworthy events — crashes of system programs, system resource exhaustion, failed login attempts, etc. The ablity to know about the occurrences of these events is critical for intrusion post-mortem analysis, since it enables the system administrator to determine the extent of the damage and possibly the method(s) of attack. The first target of an experienced attacker will be the audit log system: the attacker wishes to erase traces of the compromise, to elude detection as well as to keep the method of attack secret.

Standard audit log protection techniques typically involve writing the audit log data to some form of append-only media such as continuous-feed printers, CD-R, or DVD-R drives, or sending the log data to remote machines. In the former case, there is some form of dedicated hardware providing append-only storage semantics, preventing attackers from modify the audit log entries. In the latter case, the hope is that with distributed logging the probability that the attackers can break into all the machines unnoticed is far lower than that of breaking into a single machine. In this case, the log data must be distributed to dedicated logging machines (i.e., one which provides no other network services, etc) rather than another "normal" machine. Otherwise common-mode failures, e.g., a bug in the system software common to all the machines, would drastically increase the chances of an attacker evading detection.

In our application of forward secure MACs to audit logs, we use cryptographic means to try to provide the same append-only semantics as dedicated logging hardware. Unlike real append-only media, however, it is impossible to prevent data destruction, and the best that can be achieved is tamper-detection: the audit log is not tamper proof since entries can be modified or destroyed, but modifications cannot occur undetected.

In our scheme, we modify logging services such as Unix's system log daemon (`syslogd`) to write a tag along with the log message to verify its integrity. As with standard `syslogd`, any program may request a log entry be made; `syslogd` serializes the log entries into the appropriate log file(s). The log file(s) are later sent to a separate, secure machine for analysis and verification, perhaps periodically or because intrusion was detected and the system administrator wishes to review the logs to determine the severity of the breach. This log verifier / analysis host need not be network connected, and needs to communicate with the logging service only during a set-up phase and when the log file(s) are actually transferred.

Providing integrity for audit log data is very similar to providing integrity for messages, and our threat model is very similar to that presented above for MACs. First, the attackers first adaptively generate log entries, so that the log entries and corresponding tags are available to them. This may occur if, for example, user accounts can read the system log files, or if network logging is performed. Next, the attackers break into the system and obtains the system's current state information. Lastly, the attackers attempt to create an alternate history by forging or altering previously generated log messages.

In addition to simply trying to forge log messages, the attackers may also try to undetectably delete or reorder previously generated log messages, so in addition to simple message integrity we require *stream integrity*, where in addition to the unforgeability of messages we require that the log messages cannot be reordered or deleted undetectably. When logging with dedicated hardware, reordering and deletion of log messages is naturallly prevented. In our setup, the intruder is allowed full read/write access to the complete state of the compromised machine, and thus may overwrite previously generated, locally stored log entries.

We provide forward secure stream integrity by building our audit log scheme on top of forward secure MACs. We initialize the logging service by using a forward secure key agreement protocol, so that the logging service and the log verifier share an initial secret. This initial secret is used to initialize the MAC scheme.

When a client program requests that an audit log entry M be made, the logging service uses an internally maintained counter j to include a sequence number with the message when generating the tag. The recorded log entry is then the tuple $\langle M, \mathsf{MAC.tag}(0, j, M)\rangle$. The sequence counter is then incremented.

To update the logging system to a new stage, we first record a log entry $\langle \varepsilon, \mathsf{MAC.tag}(1, j)\rangle$ prior to running MAC.update. This log entry serves to mark the end of the stage, so that the actual number of entries made within the stage is known. After running MAC.update, the sequence

number is reset to zero.

Verifying the log entries require knowledge of the initial secret. It does not, however, require that the verifier know what stage the log system should be in: the attacker can only use MAC.update to run forward, so even if the attacker deletes the end-of-stage markers, the attacker cannot obtain the previous keys to make it appear that no roll-back had occurred. To detect such an attack it suffices to have the logging service update to the next stage and log a new (random) message prior to sending the log to the verifier. Only a logging service that has not been rolled back can do this correctly.

The sequence numbers prevent log message reordering, and the end-of-stage log entry prevents audit log truncation within previous stages. Since an attacker must break the forward secure MAC scheme in order to generate bogus log messages or to reorder or delete existing log messages generated in a previous stage, the security of the forward secure audit log is the same as that of the forward secure MAC scheme, with the proviso of the slightly shorter messages and one fewer message per stage. (We are also restricted in the number of log messages per stage by the sequence number's encoding, since it is a fixed width field.) Secure audit logs may also be built using forward secure signatures instead of forward secure message authentication codes, but currently the cost of public key operations would make that prohibitive.

A similar treatment of log security is found in [21]. There, instead of permitting several log entries to be made per stage, rekeying is performed after each log entry is made, so no per-stage sequence numbers are needed; the log entries are also encrypted. Our approach is a more modular design: whether a forward secure encryption scheme should be used is an orthogonal log design decision. Furthermore, our design makes use of an arbitrary forward secure message authentication scheme, thereby separating the underlying cryptographic problem from the application. By using the forward secure message authentication scheme constructed above, our audit log design inherits its provable security.

4 Forward secure encryption

We first provide the security definitions. The construction is based on the same paradigm as used for message authentication.

THE NOTION. In a standard (symmetric) encryption scheme, encryption and decryption are both performed under a single, shared key between the users. As in our treatment of message authentication, the first step is to consider stateful schemes. A *stateful (symmetric) encryption scheme* $\mathsf{SYM} = (\mathsf{SYM.key}, \mathsf{SYM.enc}, \mathsf{SYM.dec}, \mathsf{SYM.update})$ consists of four algorithms. $\mathsf{SYM.key}$ is run to obtain the initial key (state) K_0. The operation of the scheme is divided into stages $i = 1, 2, \ldots$, and in stage i the parties use key K_i. The key at any stage is obtained from the key at the previous stage via the update algorithm: $K_i \leftarrow \mathsf{SYM.update}(K_{i-1})$. Within stage i, the parties can encrypt a message M via $\langle C, i\rangle \leftarrow \mathsf{SYM.enc}(K_i, M)$. (The stage number i is always part of the ciphertext in order to tell the decrypter which key to use. It is required that i be computable from K_i.) We decrypt $\langle C, i\rangle$) via $M \leftarrow \mathsf{SYM.dec}(K_i, C)$. The key generation algorithm is probabilistic; the encryption algorithm is either probabilistic or stateful; the others are deterministic.

We consider privacy under chosen-plaintext attack. Privacy of data encrypted under K_j must be maintained even if the adversary is in possession of K_i for any $i > j$. The experiment measuring the success of an adversary F is described fully below:

Experiment FndGuess(E, SYM)

$K_0 \leftarrow \mathsf{SYM.key}$; $i \leftarrow 0$; $h \leftarrow \varepsilon$

```
Repeat
    i ← i + 1 ; K_i ← SYM.update(K_{i−1})
    (d, (m_0, m_1, j), h) ← E^{SYM.enc(K_i,·)}(find, h)
Until (d = guess) or (i = n)
c ← {0, 1}
If j ≥ i then return c
Else
    C ← SYM.enc(K_j, m_c) ; g ← E(guess, K_i, C, h)
    If g = c then return 1 else return 0
```

The eavesdropper E runs first in a find stage where it gets an oracle for the encryption algorithm under the current key. At the conclusion of a stage it may decide to output $d = \mathsf{guess}$ thereby saying it is ready to break-in. At that point it must also provide a pair m_0, m_1 of equal length messages, together with an indication of the stage j at which it expects to compromise the privacy. One of the messages, namely m_c, is chosen at random and encrypted under K_j to yield a challenge cihpertext C. E is then given the key K_i (from the break-in) and C, and wins if it guesses g.

We denote by $\mathsf{Succ}(E, \mathsf{SYM})$ the probability that the outcome of experiment $\mathrm{FndGuess}(E, \mathsf{SYM})$ is 1. We then let

$$\mathbf{FwInSec}(\mathsf{SYM}, q, t, n) = \max_{E}\{\mathsf{Succ}(E, \mathsf{SYM})\} \ ,$$

the maximum being over all adversaries E that run in time t and make at most q queries *in each stage*. (So the total number of queries made can reach qn.) This is the maximum likelyhood of the forward security of the message authentication scheme SYM being compromised by an adversary using the indicated resources. Read it as "forward insecurity" of the message authentication scheme.

A FORWARD SECURE ENCRYPTION SCHEME. We design a forward secure encryption scheme using a standard encryption scheme and a forward secure pseudorandom number generator, following the same paradigm used in the case of message authentication. The base key K_0 for the encryption scheme is a seed (initial state) of the generator. The key for stage i of the encryption scheme will be a triple $K_i = (i, k_i, St_i)$ consisting of the value i indicating the stage for which this is the key, an actual encryption key k_i to be used with the standard scheme, and state information St_i for the generator, based on which the next key will be generated by iteration of the generator. We will show that this scheme is forward secure given that the standard encryption is secure in the standard sense of [4] and the generator is forward secure in the sense of Section 2.

Construction 4.1 Let us now specify the construction in full. Let $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ be a standard symmetric encryption scheme consisting of key generation, encryption and decryption algorithms [4]. Say the key length for this scheme is k. Let $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ be a forward secure generator with blocksize k. (Meaning in one iteration it produces a pseudorandom block of $b = k$ bits.) Then we define a stateful symmetric encryption scheme $\mathsf{SYM} = (\mathsf{SYM.key}, \mathsf{SYM.enc}, \mathsf{SYM.dec}, \mathsf{SYM.update})$ as follows:

```
Algorithm SYM.key
    St_0 ← GEN.key
    Return (0, ε, St_0)

Algorithm SYM.dec((i, k, St), C)
    M ← sym.dec(k, C)
    Return M
```

```
Algorithm SYM.enc((i, k, St), M)
    C ← sym.enc(k, M)
    Return ⟨C, i⟩

Algorithm SYM.update((i, k, St))
    (k, St) ← GEN.next(St)
    Return (i + 1, k, St)
```

Recall that GEN.next returns a pair consisting of a pseudorandom output block (here k bits long) and an updated state; the above is saying the pseudorandom block becomes the effective new key for the underlying Enc. ▌

To discuss the security of the construction we first need to briefly recall the standard notion of security for the underlying symmetric encryption scheme.

SECURITY OF A STANDARD ENCRYPTION SCHEME. We recall the notion of enc security of [4], which in turn is a concretization and adaptation to the symmetric case of the notion of [14]. An adversary B attacking $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ gets an oracle for $\mathsf{sym.enc}(k, \cdot)$ so that it can mount a chosen-plaintext attack. It eventually outputs a pair (m_0, m_1) of equal length messages. It is then given a challenge ciphertext $C \leftarrow \mathsf{sym.enc}(k, m_c)$ where c is a random bit. It is allowed to continue to use the encryption oracle and must eventually output a guess g as to the value of c. It wins if this guess is correct.

Experiment FndGuess(B, sym)

$k \leftarrow \mathsf{sym.key}$
$(m_0, m_1, h) \leftarrow B^{\mathsf{sym.enc}(k,\cdot)}(\mathsf{find})$
$c \leftarrow \{0,1\}$; $C \leftarrow \mathsf{sym.enc}(k, m_c)$
$g \leftarrow B^{\mathsf{sym.enc}(k,\cdot)}(\mathsf{guess}, C, h)$
`If` $g = c$ `then return` 1 `else return` 0

Let $\mathsf{Succ}(B, \mathsf{sym})$ denote the probability that the above experiment returns 1 and let

$$\mathbf{InSec}(\mathsf{sym}, q, t) \;=\; \max_{B}\{\mathsf{Succ}(B, \mathsf{sym})\} \;,$$

the maximum being over all adversaries B that run in time t and make up to q queries of the $\mathsf{sym.enc}(k, \cdot)$ oracle. This is the maximum likelihood of the security of the symmetric encryption scheme being compromised in time t. Read it as "insecurity" of the scheme. Our assumption is that sym is secure so this quantity is small. We will seek to upper bound the forward insecurity of SYM in terms of the insecurity of sym.

SECURITY OF OUR CONSTRUCTION. We claim that the stateful symmetric encryption scheme we constructed above is forward secure as long as the underlying symmetric encryption scheme is secure in the standard sense and the generator is forward secure. The proof of the following is similar to the proof of Theorem 3.2 and hence is omitted.

Theorem 4.2 Let $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ be a standard symmetric encryption scheme with key size k, and $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next})$ a forward secure generator with blocksize k. Let SYM be the stateful symmetric encryption scheme formed out of these two as above. Then

$$\mathbf{FwInSec}(\mathsf{SYM}, q, t, n) \;\le\; \mathbf{FwInSec}(\mathsf{GEN}, t_1) + n \cdot \mathbf{InSec}(\mathsf{sym}, q, t_2) \;,$$

where $t_1 = 2t + O(n + k)$ and $t_2 = 2t + O(n + k)$.

References

[1] M. ABDALLA AND L. REYZIN, "A new forward-secure digital signature scheme," *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000.

[2] M. ABDALLA, S. MINER AND C. NAMPREMPRE, "Forward secure threshold signature schemes," *Proceedings of the RSA 2001 conference Crypto track*, Lecture Notes in Computer Science Vol. 2020, Springer-Verlag, 2001.

[3] R. Anderson, "Two Remarks on Public-Key Cryptology," Manuscript, 2000, and Invited Lecture at the Fourth Annual Conference on Computer and Communications Security, Zurich, Switzerland, April 1997.

[4] M. Bellare, A. Desai, E. Jokipii and P. Rogaway, "A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation," *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997.

[5] M. Bellare, J. Kilian and P. Rogaway, "The security of cipher block chaining," *Advances in Cryptology – CRYPTO '94*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.

[6] M. Bellare and S. Miner, "A forward-secure digital signature scheme," *Advances in Cryptology – CRYPTO '99*, Lecture Notes in Computer Science Vol. 1666, M. Wiener ed., Springer-Verlag, 1999.

[7] L. Blum, M. Blum and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on Computing* Vol. 15, No. 2, 364-383, May 1986.

[8] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo-random bits," *SIAM Journal on Computing*, Vol. 13, No. 4, 850-864, November 1984.

[9] R. Canetti and A. Herzberg, "Maintaining security in the presence of transient faults," *Advances in Cryptology – CRYPTO '94*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.

[10] C.-S. Chow and A. Herzberg, "Network randomization protocol: A proactive pseudo-random generator," *Proceedings of the 5th Usenix Unix Security Symposium*, June 1995.

[11] Y. Desmedt, "Threshold cryptography," *European Trans. on Telecommunications*, Vol. 5, No. 4, pp. 449-457, July-August 1994.

[12] W. Diffie, P. van Oorschot and M. Wiener, "Authentication and authenticated key exchanges", *Designs, Codes and Cryptography*, 2, 1992, pp. 107–125.

[13] O. Goldreich, S. Goldwasser and S. Micali, "How to construct random functions," *Journal of the ACM*, Vol. 33, No. 4, 1986, pp. 210–217.

[14] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, Vol. 28, 1984, pp. 270–299.

[15] S. Goldwasser, S. Micali and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal of Computing*, Vol. 17, No. 2, pp. 281–308, April 1988.

[16] C. Günther, "An identity-based key-exchange protocol," *Advances in Cryptology – EUROCRYPT '89*, Lecture Notes in Computer Science Vol. 434, J-J. Quisquater, J. Vandewille ed., Springer-Verlag, 1989.

[17] D. Harkins and D. Carrel, ed., "The Internet Key Exchange (IKE)", *Internet Draft*, `draft-ietf-ipsec-isakmp-oakley-06.txt`, February 1998.

[18] H. Krawczyk, "Simple forward-secure signatures from any signature scheme," *Proceedings of the 7th Annual Conference on Computer and Communications Security*, ACM, 2000.

[19] A. Herzberg, S. Jarecki, H. Krawczyk and M. Yung, "Proactive secret sharing, or: How to cope with perpetual leakage," *Advances in Cryptology – CRYPTO '95*, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995.

[20] U. S. National Institute of Standards and Technology, "Federal information processing standards publication 140-1: Security requirements for cryptographic modules", January 1994.

[21] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," *Proceedings of the 7th USENIX Security Symposium*, USENIX Press, 1998.

[22] W. Tzeng and Z. Tzeng, "Robust Forward-Secure Digital Signature with Proactive Security," *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography*, Lecture Notes in Computer Science Vol. ??, K. Kim, ed., Springer-Verlag 2001.

[23] A. Yao, "Theory and applications of trapdoor functions," *Proceedings of the 23rd Symposium on Foundations of Computer Science*, IEEE, 1982.

A Proofs

A.1 Proof of Theorem 2.2

Let A be an adversary attacking the forward security of GEN and having a running time of at most t. We want to upper bound $\mathsf{Adv}(A, \mathsf{GEN})$. We begin by defining the following sequence of hybrid experiments.

Experiment Hybrid1(A, GEN, j) $\quad (0 \le j \le n)$

$St \leftarrow \{0,1\}^k$; $i \leftarrow 0$; $h \leftarrow \varepsilon$
Repeat
  $i \leftarrow i + 1$
  If $i \le j$ **then** $Out_i \leftarrow \{0,1\}^b$ **else** $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
  $(d, h) \leftarrow A(\mathsf{find}, Out_i, h)$
Until $(d = \mathsf{guess})$ or $(i = n)$
$g \leftarrow A(\mathsf{guess}, St, h)$
Return g

We let $P_{1,j}$ be the probability that experiment Hybrid1(A, GEN, j) returns 1, for $j = 0, \ldots, n$. Note that the experiments Real(A, GEN) and Hybrid$(A, \mathsf{GEN}, 0)$ are identical. (Not syntactically, but semantically.) This means that $P_{1,0} = \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real})$. We would like that the experiments Random(A, GEN) and Hybrid1(A, GEN, n) also be identical so that $P_{1,n} = \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random})$, but this in fact is not true. The reason is that in Random(A, GEN) the adversary, although receiving random blocks $Out_1, Out_2, \ldots$, does receive the true state of the pseudorandom generator when it breaks in, while in Hybrid1(A, GEN, n) the state it receives upon breaking in is a random string. We could modify the hybrid experiment to rectify this, but then the hybrid does not seem amenable to an analysis based on the security of the generator G. Instead we introduce another hybrid sequence which begins where Hybrid1(A, GEN, n) left off, and bridges the gap to Random(A, GEN).

Experiment Hybrid2(A, GEN, j) $\quad (0 \le j \le n)$

$St \leftarrow \{0,1\}^k$; $i \leftarrow 0$; $h \leftarrow \varepsilon$
Repeat
  $i \leftarrow i + 1$
  If $i \le j$ **then** $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
  $Out_i \leftarrow \{0,1\}^b$
  $(d, h) \leftarrow A(\mathsf{find}, Out_i, h)$
Until $(d = \mathsf{guess})$ or $(i = n)$
$g \leftarrow A(\mathsf{guess}, St, h)$
Return g

We let $P_{2,j}$ be the probability that experiment Hybrid2(A, GEN, j) returns 1, for $j = 0, \ldots, n$. Note that the experiments Hybrid2$(A, \mathsf{GEN}, 0)$ and Hybrid1(A, GEN, n) are identical. This means

that $P_{2,0} = P_{1,n}$. Also the experiments $\mathrm{Random}(A, \mathsf{GEN})$ and $\mathrm{Hybrid2}(A, \mathsf{GEN}, n)$ are identical, so $P_{2,n} = \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random})$. Putting all this together we have

$$\begin{aligned} \mathsf{Adv}(A, \mathsf{GEN}) &= \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real}) - \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random}) \\ &= P_{1,0} - P_{2,n} \\ &= [\, P_{1,0} - P_{1,n} \,] + [\, P_{2,0} - P_{2,n} \,] \,. \end{aligned} \tag{1}$$

We now claim that

$$P_{1,0} - P_{1,n} \;\leq\; n \cdot \mathbf{InSec}(G, t') \tag{2}$$

$$P_{2,0} - P_{2,n} \;\leq\; n \cdot \mathbf{InSec}(G, t') \,. \tag{3}$$

Combining Equations (1), (2) and (3) we have

$$\mathsf{Adv}(A, \mathsf{GEN}) \;\leq\; 2n \cdot \mathbf{InSec}(G, t') \,.$$

Since A was an arbitrary adversary running in time t we obtain the conclusion of the theorem. It remains to justify Equations (2) and (3). We will do this using the security of G. To justify the first of these bounds consider the following distinguisher D_1.

```
Algorithm D1(x || y)   (|x| = b and |y| = k)
  j ← {1,...,n} ; i ← 0 ; h ← ε
  Repeat
    i ← i + 1
    If i < j then Out_i ← {0,1}^b
    If i = j then (Out_i, St) ← (x, y)
    If i > j then (Out_i, St) ← GEN.next(St)
    (d, h) ← A(find, Out_i, h)
  Until (d = guess) or (i = n)
  g ← A(guess, St, h)
  Return g
```

Suppose we run experiment $\mathrm{Random}(D_1, G)$. We notice that it amounts to running $\mathrm{Hybrid1}(A, \mathsf{GEN}, j)$ where j is the value chosen at random by D_1 in its first step. Similarly if we run experiment $\mathrm{Real}(D_1, G)$ we notice that it amounts to running $\mathrm{Hybrid1}(A, \mathsf{GEN}, j-1)$ where j is the value chosen at random by D_1 in its first step. So

$$\begin{aligned} \mathsf{Succ}(D_1, G, \mathrm{Real}) &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j-1} \\ \mathsf{Succ}(D_1, G, \mathrm{Random}) &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j} \end{aligned}$$

.

Subtract the second sum from the first and exploit the collapse to get

$$\frac{P_{1,0} - P_{1,n}}{n} \;=\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j} \;-\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j-1} \;=\; \mathsf{Adv}(D_1, G) \,.$$

Note that the running time of D_1 is at most the quantity t' in the theorem statement, whence we get Equation (2). Now consider distinguisher D_2 defined below.

```
Algorithm D2(x || y)   (|x| = b and |y| = k)
  j ← {1,...,n} ; i ← 0 ; h ← ε
  Repeat
```

$i \leftarrow i + 1$
If $i = 1$ then $(Out_i, St) \leftarrow (x, y)$
If $1 < i \leq j$ then $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
$Out_i \leftarrow \{0,1\}^b$
$(d, h) \leftarrow A(\mathsf{find}, Out_i, h)$
Until $(d = \mathsf{guess})$ or $(i = n)$
$g \leftarrow A(\mathsf{guess}, St, h)$
Return $1 - g$

Suppose we run experiment $\mathrm{Real}(D_2, G)$. We notice that it amounts to running $\mathrm{Hybrid2}(A, \mathsf{GEN}, j)$ where j is the value chosen at random by D_2 in its first step, and then flipping the value of the answer bit. Similarly if we run experiment $\mathrm{Random}(D_2, G)$ we notice that it amounts to running $\mathrm{Hybrid2}(A, \mathsf{GEN}, j-1)$ where j is the value chosen at random by D_2 in its first step, and then flipping the answer bit. So

$$\begin{aligned} \mathsf{Succ}(D_2, G, \mathrm{Real}) &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} 1 - P_{2,j} \\ \mathsf{Succ}(D_2, G, \mathrm{Random}) &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} 1 - P_{2,j-1} \ . \end{aligned}$$

Subtract the second sum from the first and exploit the collapse to get

$$\frac{P_{2,0} - P_{2,n}}{n} = \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(1 - P_{2,j}) - \tfrac{1}{n}\sum_{j=1}^{n}(1 - P_{2,j-1}) = \mathsf{Adv}(D_2, G) \ .$$

Note that the running time of D_2 is at most the quantity t' in the theorem statement, whence we get Equation (3). This concludes the proof of the theorem.

A.2 Proof of Theorem 3.2

Let F be a forger attacking the forward security of the MAC scheme and having running time at most t. We want to upper bound $\mathsf{Succ}(F, \mathsf{MAC})$. To do this we specify an adversary A attacking the forward security of the generator GEN, and a forger f attacking the mac scheme, and then bound the success of F in terms of the success of f and the advantage of A.

THE ADVERSARY A. A will receive a sequence of blocks, one by one, and must tell whether they are outputs of the generator or truly random, with an option to break-in and get the current state at some point. It will run a simulation of the experiment $\mathrm{Forge}(F, \mathsf{MAC})$ by letting the blocks it receives play the role of the keys k_i that are used by mac in the MAC scheme. A will test whether or not F succeeds on the given sequence of blocks. If so, it bets that the block sequence was pseudorandom, and if not, it bets that the block sequence was random. In adopting the latter opinion, it is assuming that forgery is hard on a random block sequence, which we bear out later by providing a forger which breaks the given (standard) message authentication scheme mac otherwise. The algorithm A is below, and explanations follow.

Algorithm $A(\mathsf{find}, Out, h)$
if $h = \varepsilon$ then $h \leftarrow \varepsilon \,\|\, \varepsilon \,\|\, 0$
Parse h as $h_F \,\|\, h_O \,\|\, h_T \,\|\, i$
$h_O \leftarrow h_O \,\|\, Out$; $i \leftarrow i + 1$
$(d, h_F) \leftarrow F^{\langle \mathsf{mac.tag}(Out, \cdot), i \rangle}(\mathsf{find}, h_F)$
Append to h_T the transcript of oracle queries of F
If $d = \mathsf{forge}$ then $d \leftarrow \mathsf{guess}$
$h \leftarrow h_F \,\|\, h_O \,\|\, h_T \,\|\, i$
Return (d, h)

Algorithm $A(\mathsf{guess}, St, h)$
Parse h as $h_F \,\|\, h_O \,\|\, h_T \,\|\, i$
Parse h_O as $Out_1 \,\|\, \cdots \,\|\, Out_i$
$K \leftarrow (i, Out_i, St)$
$(M, \langle \tau, j \rangle) \leftarrow F(\mathsf{forge}, K, h_F)$
If $\mathsf{mac.vf}(Out_j, M, \tau) = 1$ and $1 \leq j < i$ and M was not queried of $\mathsf{mac.tag}(Out_j, \cdot)$
then return 1 else return 0

In the find stage, A receives the current output block Out and runs F in the latter's find stage. The notation $F^{\langle \mathsf{mac.tag}(Out,\cdot),i\rangle}(\mathsf{find}, h_F)$ means that F is given the oracle that on input M returns $\langle \mathsf{mac.tag}(Out, M), i\rangle$. ($A$ can simulate this oracle since it knows Out.) Thus A is simulating the tagging oracle $\mathsf{MAC.tag}_{K_i}$ that F gets at this stage in the right way given the definition of MAC. Here h_F denotes F's history string, which A maintains. In addition A maintains other histories: h_0 records the sequence $Out_1 \,\|\, Out_2 \cdots$ of output blocks; h_T records the transcripts of oracle queries made by F so that later, in the guess stage, it is possible for A to determine whether or not M was ever queried; and i records the current stage. When F breaks in, A does the same, and from the state St of the generator it gets thereby, returns to F the information the latter would have obtained had it broken in, namely the key K. Then A lets F try to forge, and tests whether or not the forgery is valid and new. If so, it returns one, else zero.

Notice that the experiments $\mathrm{Real}(A, \mathsf{GEN})$ and $\mathrm{Forge}(F, \mathsf{MAC})$ are identical. So

$$\mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real}) \;=\; \mathsf{Succ}(F, \mathsf{MAC}) \;. \tag{4}$$

THE FORGER f. We design a forging algorithm f attacking the given scheme mac. It gets an oracle for $\mathsf{mac.tag}(k,\cdot)$ and eventually outputs a pair (M,τ). It runs F, but on a sequence of random, independent keys rather than keys obtained via the generator. It begins by making a guess l at the stage j at which F forges, and runs F so that the role of k_l is played by k. To do that, it answers oracle queries made by F in the l-th stage using $\mathsf{mac.tag}(k,\cdot)$. If $j \neq l$ then f cannot hope to win and aborts. Also if F requests key K_l at break-in time then f, not knowing $k_l = k$, will be unable to oblige and again aborts. Barring that it gets a valid forgery with respect to k.

Algorithm $f^{\mathsf{mac.tag}(k,\cdot)}$
  $l \leftarrow \{1,\ldots,n\}$; $St_0 \leftarrow \mathsf{GEN.key}$; $i \leftarrow 0$; $h \leftarrow \varepsilon$
  Repeat
    $i \leftarrow i+1$; $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$; $k_i \leftarrow \{0,1\}^k$; $K_i \leftarrow (i, k_i, St_i)$
    If $i = l$
      Then $(d,h) \leftarrow F^{\langle \mathsf{mac.tag}(k,\cdot),i\rangle}(\mathsf{find}, h)$
      Else $(d,h) \leftarrow F^{\langle \mathsf{mac.tag}(k_i,\cdot),i\rangle}(\mathsf{find}, h)$
  Until $(d = \mathsf{forge})$ or $(i = n)$
  If $i = l$ **then abort**
  Else
    $(M, \tau, j) \leftarrow F(\mathsf{forge}, K_i, h)$
    If $j = l$ **then return** (M,τ) **else abort**

Notice that $\mathsf{Succ}(f, \mathsf{mac}) = \mathsf{Succ}(A, \mathsf{GEN.}, \mathrm{Random})/n$. That is

$$\mathsf{Succ}(A, \mathsf{GEN.}, \mathrm{Random}) \;=\; n \cdot \mathsf{Succ}(f, \mathsf{mac}) \;. \tag{5}$$

Now combining Equations (4) and (5) we get

$$\begin{aligned} \mathsf{Succ}(F, \mathsf{MAC}) &= \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Real}) \\ &= \mathsf{Adv}(A, \mathsf{GEN}) + \mathsf{Succ}(A, \mathsf{GEN}, \mathrm{Random}) \\ &= \mathsf{Adv}(A, \mathsf{GEN}) + n \cdot \mathsf{Succ}(f, \mathsf{mac}) \;. \end{aligned}$$

Now observe that the running time of A is at most t_1 and that of f is at most t_2. It follows that

$$\mathbf{FwInSec}(\mathsf{MAC}, t, q) \;\leq\; \mathbf{FwInSec}(\mathsf{GEN}, t_1) + n \cdot \mathbf{InSec}(\mathsf{mac}, q, t_2) \;,$$

as desired.