A preliminary version of this paper appears in *Topics in Cryptology – CT-RSA '03*, Lecture Notes in Computer Science Vol. ?? , M. Joye ed., Springer-Verlag, 2003. This is the full version.

Forward-Security in Private-Key Cryptography

Mihir Bellare[*] Bennet Yee[†]

November 2000

Abstract

This paper provides a comprehensive treatment of forward-security in the context of shared-key based cryptographic primitives, as a practical means to mitigate the damage caused by key-exposure. We provide definitions of security, practical proven-secure constructions, and applications for the main primitives in this area. We identify forward-secure pseudorandom bit generators as the central primitive, providing several constructions and then showing how forward-secure message authentication schemes and symmetric encryption schemes can be built based on standard schemes for these problems coupled with forward-secure pseudorandom bit generators. We then apply forward-secure message authentication schemes to the problem of maintaining secure access logs in the presence of break-ins.

Keywords: Symmetric cryptography, forward security, pseudorandom bit generators, message authentication, proofs of security, audit logs.

[*]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 grants CCR-0098123 and ANR-0129617, and by an IBM Faculty Partnership Development Award.

[†]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.

The focus of this paper is forward-security in the symmetric setting, where parties share a key. We provide a comprehensive treatment including security definitions and practical proven-secure constructions, for the main primitives in this area, namely pseudorandom bit generators, message authentication schemes and symmetric encryption schemes.

Of these, however, we claim the first (namely a forward-secure pseudorandom bit generator) is the most important. Not only is it useful in its own right, but once we have this, the other primitives can be easily and generically built. We thus start with this.

Forward security in pseudorandom bit generation. Random values need to be generated in many cryptographic implementations. For example, session-key exchange protocols may need to generate random session keys. Signature schemes such as DSS require generation of a random value with each signature. Encryption schemes, both in the symmetric and asymmetric settings, are usually probabilistic. In any of these settings, if the random value in question is exposed, even some time after the cryptographic operation using it took place, security is compromised. In particular, data encrypted under the session key will be exposed once the key is exposed. In DSS, exposure of the random value yields the secret signing key to an attacker. In most probabilistic encryption schemes, exposure of the randomness used by the encryption algorithm leads to exposure of the plaintext or at least loss of partial information about the plaintext.

Often, random values are generated from some seed via a pseudorandom bit 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 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, given in Section 2.2, 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 [13, 27].

Construction 2.2 is a way to transform any standard (stateless, not forward secure) pseudorandom bit generator into a forward-secure pseudorandom bit generator. The construction is quite simple and cheap. Furthermore we prove in Theorem 2.3 that if the original generator is secure in the sense of [13, 27] then our constructed one is forward-secure in the strong sense we define. We also suggest, via Construction 2.4, a design of forward-secure pseudorandom bit generators based on pseudorandom functions (PRFs). An attractive feature of this design illustrated by Theorem 2.5 is that the PRF need be secure against only a very small number of queries and yet we can generate many pseudorandom blocks. The PRF can be instantiated via a block cipher or a hash function in practice, as discussed in Section 2.5.

A natural question is whether existing constructs of stateful generators in the literature are forward-secure. One such construct is the alleged-RC4 stream cipher. But we show in Section 2.3 that it is not forward secure because its state update process is easily reversible.

In Section 2.6 we look at number-theoretic constructions like those of [13, 12]. These are not presented as stateful generators, 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 pseudorandom bit generation that was introduced in [13].

These number-theoretic constructions are however slower than the ones discussed above. Our suggested construction of a forward-secure pseudorandom bit generator is the one of Construction 2.2 instantiated with a block cipher, or the one of Construction 2.2 instantiated with a standard pseudorandom bit generator.

As indicated above, forward-secure pseudorandom bit generators can be useful as stand-alone tools for generating the random bits needed by other cryptographic primitives, both symmetric and asymmetric. Now we show that they are also important as building blocks in the design of forward-secure primitives for other problems in symmetric cryptography.

Forward-secure message authentication. Exposure of a key being used for message authentication not only compromises future uses of the key but makes data previously authenticated under this untrustworthy, just as for digital signatures [3, 8]. To remedy this, we can use a forward-secure message authentication scheme. Such a scheme is stateful and key-evolving. The operation of the scheme is divided into stages $i = 1, 2, \ldots, n$ and in each stage the parties use the current key K_i for creation and verification of authentication tags. 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 $K_1, \ldots, K_{i-1}$.

Section 3.2 provides strong, formal definitions of security capturing this. Construction 3.1 then shows how to build a forward-secure message authentication scheme given any standard message authentication scheme (in practice this could be any popular one like a CBC-MAC, HMAC [5] or UMAC [11]) and a forward secure pseudorandom bit generator (in practice this could be any of the secure ones mentioned above). One simply applies the forward-secure generator to update the message-authentication key between stages. Theorem 3.2 proves the forward-security of this construction based on the assumed security of the base primitives.

Secure audit logs. Forward security is relevant in settings like the usage of message authentication for secure audit logs, as discussed in [26]. An attacker breaking into a machine currently can modify log entries relating to the past, erasing for example a record of the attacker's previous (unsuccessful) attempts at break-in. To prevent this we use a forward-secure message authentication scheme to tag 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 4.

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.

Forward secure symmetric encryption. The technique used to construct a forward-secure message authentication scheme, based on a standard message authentication and a forward-secure pseudorandom bit generator, is quite general. In particular, the same technique can be used to construct a forward-secure symmetric encryption scheme based on a standard symmetric encryption scheme (in practice this could be any common block-cipher mode of operation, such as CBC with random IV) and a forward-secure pseudorandom bit generator (in practice this could be any of the secure ones mentioned above.) 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. Definitions, the construction, and provable-security results can be found in Section 5. The construction preserves both security under chosen-plaintext and chosen-ciphertext attack in the sense that, if the given standard symmetric encryption scheme is secure in one of these senses then so is the constructed forward-secure scheme.

Extensions. The paradigm of using a forward-secure pseudorandom bit generator to evolve the key of some standard symmetric primitive is very general, and can also be used to convert standard primitives to forward-secure versions in the case of other primitives like PRFs or authenticated encryption schemes [9]. We do not detail these extensions in this paper since they are quite simple.

Related work. Forward security seems to have first received explicit attention in the context of session key exchange protocols [21, 18], where it is now a common requirement. Forward-security for digital signatures was introduced in [3, 8] and has since then received much attention, and forward-security for asymmetric encryption was considered in [24]. Ours seems to be the first systematic treatment of forward-security in the symmetric setting, but it reflects existing work, practice and design. Forward-secure pseudorandom bit generation has been used in [4, 22]. In practice it is common to design pseudorandom bit generators that on each iteration produce a new seed and delete the old one. Our work can be seen as analyzing, justifying and guiding such existing practice.

The threat of key compromise has been addressed by other means. One is via distribution of the key across multiple machines. Specific approaches include threshold cryptography [17] and proactive secret sharing [23]. (In particular a proactive, distributed pseudorandom bit generator has been designed by [14, 15].) 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. Forward-security in contrast is possible in a single-machine environment.

Another, less cryptographic mechanism is embodied in systems such as the those incorporating FIPS 140-1 certified modules [25]. 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.

Besides providing forward-security, Key-evolving constructs such as those used here can enable more cryptographic operations to be securely implemented with a single key [1]. Other notions of security for pseudorandom bit generators have been considered in [16], and it would be fruitful to augment these with forward security.

2 Forward-secure pseudorandom bit generators

We recall the standard notion of pseudorandom bit generators. We then specify how forward secure generators operate and provide a formal notion of security for them. Then we explore constructions.

2.1 Standard Pseudorandom bit generators

A standard pseudorandom bit generator [13, 27] is a function G: $\{0,1\}^s \to \{0,1\}^{b+s}$ that takes input a s-bit seed and returns a string that is longer than the seed by b bits. We adapt the notion of security of [13, 27] to a concrete security setting [7, 6]. Let D be a distinguishing algorithm that given a $b+s$ bit string $x \,\|\, y$ returns a bit. Consider the following experiments:

```
Experiment $\mathbf{Exp}_G^{\text{prg-1}}(D)$
    $y \xleftarrow{\$} \{0,1\}^s$ ; $x \,\|\, y \leftarrow G(y)$
    $g \xleftarrow{\$} D(x \,\|\, y)$
    Return $g$

Experiment $\mathbf{Exp}_G^{\text{prg-0}}(D)$
    $x \,\|\, y \xleftarrow{\$} \{0,1\}^{b+s}$
    $g \xleftarrow{\$} D(x \,\|\, y)$
    Return $g$
```

We let

$$\begin{aligned}
\mathbf{Adv}_G^{\text{prg}}(D) &= \Pr[\,\mathbf{Exp}_G^{\text{prg-1}}(D) = 1\,] - \Pr[\,\mathbf{Exp}_G^{\text{prg-0}}(D) = 1\,] \\
\mathbf{Adv}_G^{\text{prg}}(t) &= \max_D\{\mathbf{Adv}_G^{\text{prg}}(D)\}\ .
\end{aligned}$$

The first term is the *prg-advantage* of D in attacking G. The second term is the *prg-advantage* of G, defined as the maximum, over all adversaries D that have time-complexity at most t, of the prg-advantage of D in attacking G. This is the maximum likelihood of the security of the generator being compromised by an attack that is restricted to time t. We adopt the convention that the time-complexity is the total worst-case execution time of the first experiment above plus the size of the code of the adversary, all measured in some fixed model of computation. (In particular the time-complexity includes the time for computation of G in the experiment as well as the running time of D.) As usual under the concrete security framework [7, 6], there is no formal notion of G being "secure," but informally it means that G's prg-advantage is "small" for "practical" values of t. All formal results will be in stated in concrete security terms.

2.2 Forward-secure pseudorandom bit generators

Unlike the standard pseudorandom bit generators discussed above, a forward-secure one is a stateful object, and thus we begin by discussing stateful generators.

Stateful generators. A *stateful generator* $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ is specified by a pair of algorithms and a pair of positive integers. The (probabilistic) *key generation* algorithm $\mathsf{GEN.key}$ takes no inputs and outputs an initial state (also called seed). The (deterministic) *next step* algorithm, given the current state, returns a pair consisting of an output block, which is a b-bit string, and the next state. We can get a sequence $Out_1, Out_2, \ldots, Out_n$ of b-bit output blocks by first picking a seed $St_0 \xleftarrow{\$} \mathsf{GEN.key}$ and then iterating $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$ for $i = 1, \ldots, n$as depicted in Figure 1. The integer n is the maximum number of output blocks the generator may be used to produce.

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.

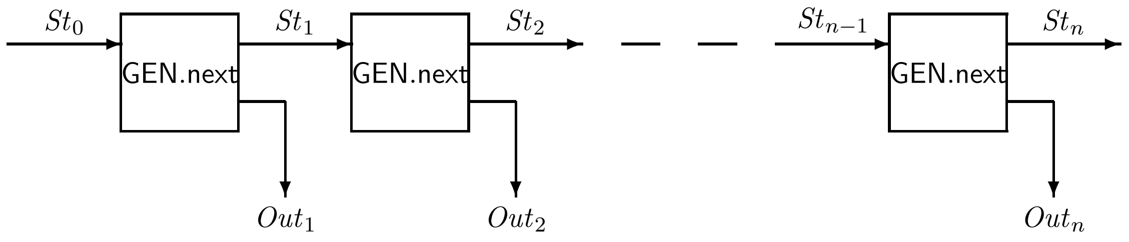

Figure 1: `Operation of a forward-secure pseudorandom bit generator:` A sequence $Out_1, \ldots, Out_n$ of pseudorandom blocks is produced starting from an initial seed St_0.

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. As we have seen above, a standard 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 [13, 27]. For forward security of a stateful generator, more is required. The adversary may at some point break into the machine where the state is being maintained and obtain the current state. At that point, the adversary can certainly compute the future output of the generator. However, we require 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).

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 $d = \mathsf{guess}$ or all n output blocks have been generated. (In the latter case the adversary is given the final state in the guess stage.)

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

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

We let

$$\begin{aligned}\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(A) &= \Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}1}}_{\mathsf{GEN}}(A)=1\,] - \Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}0}}_{\mathsf{GEN}}(A)=1\,] \\ \mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t) &= \max_{D}\{\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(A)\}\ .\end{aligned}$$

The first term is the *fsprg-advantage* of A in attacking GEN. The second term is the *fsprg-advantage* of GEN, defined as the maximum, over all adversaries A that have time-complexity at most t, of the fsprg-advantage of A in attacking GEN. This is the maximum likelihood of the security of the generator being compromised by an attack that is restricted to time t. We adopt the convention that the time-complexity is the total worst-case execution time of the first experiment above plus the size of the code of the adversary, all measured in some fixed model of computation. (In particular the time-complexity includes the time for computations of $\mathsf{GEN.key}, \mathsf{GEN.next}$ in the experiment as well as the running time of D.) As usual under the concrete security framework [7, 6], there is no formal notion of GEN being "secure," but informally it means that GEN's fsprg-advantage is "small" for "practical" values of t. All formal results will be in stated in concrete security terms.

Remark 2.1 Let $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ be a stateful generator whose key-generation algorithm produces random strings of length s bits. Then GEN has a natural associated standard pseudorandom bit generator $G\colon \{0,1\}^s \to \{0,1\}^{bn}$ defined as follows. For any $St_0 \in \{0,1\}^s$ and $i = 1, \ldots, n$ we let $(Out_i, St_i) \leftarrow \mathsf{GEN.next}(St_{i-1})$, and then let $G(St_0) = Out_1 \,\|\, \cdots \,\|\, Out_n$. Note that if GEN is a forward-secure stateful pseudorandom bit generator then G is a secure standard pseudorandom bit generator. In this sense, forward-security implies standard security. ▌

2.3 Alleged-RC4 is not forward-secure

Some stream ciphers like alleged-RC4 (numerous descriptions can be found on the web, for example [2]) 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. It is natural to ask whether this stateful generator has the forward security property. It turns out that it does not. We present an attack demonstrating this.

Below we express 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 viewed as a map of 8 bits to 8 bits, and x and y are inputs to this map, each 8 bits long.

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

```
Algorithm 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 $(z, (s,x,y))$
```

We note that the state updates are reversible. Above we also 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. Here z is the output of the previous state, and (s, x, y) in the output of AntiRC4 is the previous state. 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.

2.4 A construction based on standard generators

There are many existing pseudorandom bit 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 2.2 Let $G\colon \{0,1\}^s \to \{0,1\}^{b+s}$ be a standard pseudorandom bit generator, and let $n \geq 1$ be an integer. We associate to G, n a stateful generator $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ whose constituent algorithms are as follows:

```
Algorithm GEN.key              | Algorithm GEN.next($St$)
    $St_0 \xleftarrow{\$} \{0,1\}^s$      |     $r \leftarrow G(St)$
    Return $St_0$               |     Return $([r]_{1..b}, [r]_{b+1..b+s})$
```

The state of this stateful generator is a s-bit string. We are denoting by $[r]_{i..j}$ the substring of r consisting of the bits in positions i through j. ▮

The following theorem says that this stateful generator is forward secure as long as the given standard generator G met the standard notion of security of pseudorandom generators of Section 2.1. The proof is in Appendix A.

Theorem 2.3 Let $G\colon \{0,1\}^s \to \{0,1\}^{b+s}$ be a standard pseudorandom bit generator, let $n \geq 1$ be an integer, and let GEN be the stateful generator associated to G, n by Construction 2.2. Then

$$\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t) \;\leq\; 2n \cdot \mathbf{Adv}^{\mathrm{prg}}_{G}(t')\ ,$$

where $t' = t + O(n \cdot (b+s))$. ▮

2.5 A construction based on PRFs

We specify a construction of a forward-secure pseudorandom bit generator based on a pseudorandom function (PRF). A special case of practical interest is when we instantiate the PRF via a block cipher. We will see that this turns out to be good for cost and security, and is the preferred construction we suggest. Let us first recall some background about PRFs and their security.

PRFs. A PRF is a map $F\colon \{0,1\}^s \times \{0,1\}^l \to \{0,1\}^L$, where s is the key-length, l is the input length and L is the output length. For each key $S \in \{0,1\}^s$ we let F_S denote the function $F(S, \cdot)$. Let $\mathrm{Func}[l \to L]$ denote the set of all functions with domain $\{0,1\}^l$ and range $\{0,1\}^L$. We now recall the measure of the security of F from [7], which in turn is a quantified version of the original notion of [19]. Let D be a distinguishing algorithm, having access to an oracle for a function $f\colon \{0,1\}^l \to \{0,1\}^L$ and returning a bit. Let

$$\begin{aligned}\mathbf{Adv}^{\mathrm{prf}}_{F}(D) &= \Pr[\, D^f = 1 \,:\, S \xleftarrow{\$} \{0,1\}^s \,;\, f \leftarrow F_S \,] - \Pr[\, D^f = 1 \,:\, f \xleftarrow{\$} \mathrm{Func}[l \to L] \,] \\ \mathbf{Adv}^{\mathrm{prf}}_{F}(q,t) &= \max_{D}\{\mathbf{Adv}^{\mathrm{prf}}_{F}(D)\}\ .\end{aligned}$$

The maximum is over all adversaries D that have time-complexity (as per the usual convention, this is the total worst-case execution time of the experiment underlying the first term in the difference above, plus the size of the code of D) at most t and make at most q oracle queries.

Construction. A PRF can be easily transformed into a standard pseudorandom bit generator, and we can then apply our previous construction. Here are the details.

Construction 2.4 Let F: $\{0,1\}^s \times \{0,1\}^\ell \to \{0,1\}^L$ be a PRF, and let $b, n \geq 1$ be integers such that $\lceil (b+s)/L \rceil \leq 2^l$. For any s-bit S we let $G(S)$ denote the first $b+s$ bits of the sequence $F(S,0) \,\|\, F(S,1) \,\|\, F(S,2) \,\|\, \cdots$. (Here we are using S as the key to the PRF, and the integer inputs to the PRF are interpreted as l-bit strings in some natural way.) This defines a standard pseudorandom bit generator G: $\{0,1\}^s \to \{0,1\}^{b+s}$. The stateful generator associated to F, b, n is defined as the stateful generator associated by Construction 2.2 to G, n. ▌

The above construction of GEN is quite efficient, using only $\lceil (b+s)/L \rceil$ applications of F to implement one step of the forward secure generator (which yields one b-bit output block and an updated state).

Security. An attractive feature of the PRF based construction is that the security requirements on the PRF are low in the sense that it need only resist attacks involving a small number of queries, much fewer than the number of output blocks the generator can generate. This is illustrated by the following theorem.

Theorem 2.5 Let F: $\{0,1\}^s \times \{0,1\}^l \to \{0,1\}^L$ be a PRF, let $b, n \geq 1$ be integers with $\lceil (b+s)/L \rceil \leq 2^l$, and let GEN be the stateful generator associated to F, b, n as per Construction 2.4. Then

$$\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t) \;\leq\; 2n \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(q, t') \;,$$

where $q = \lceil (b+s)/L \rceil$ and $t' = t + O(n \cdot (b+s))$. ▌

Proof of Theorem 2.5: Let G be the standard pseudorandom bit generator associated to F, b as per Construction 2.4. Then

$$\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t) \;\leq\; n \cdot \mathbf{Adv}^{\mathrm{prg}}_{G}(t') \;\leq\; n \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(q, t')$$

where the first inequality is by Theorem 2.3 and the second is standard. ▌▌

AES based instantiation. Let $F = \mathrm{AES}$. In that case, $s = l = L = 128$. Let us choose $b = 128$ and let $n \geq 1$ be an integer. Let GEN be the stateful generator associated to F, b, n as per Construction 2.4. Each iteration of GEN returns $b = 128$ output bits and requires $\lceil (b+s)/L \rceil = 2$ AES computations under the same key, which is quite cheap.

Regarding security, note that $\mathbf{Adv}^{\mathrm{prf}}_{F}(2, t')$ can be expected to be very small, on the order of $t'/2^s$. This is because exhaustive key-search is likely to be the best attack when the attacker has access to only two input-output pairs of the cipher under the key being attacked. (Cryptanalytic attacks typically need many more examples, and so do birthday attacks.) Thus, Theorem 2.5 tells us that the probability of an attacker, having time-complexity t and attacking up to n output blocks, being able to compromise the forward-security of our AES-based stateful generator, is at most around $2n \cdot t/2^s$, meaning we can safely produce up to 2^{64} output blocks.

Hash-function based instantiations. Let H be the cryptographic hash function SHA-1, and let F: $\{0,1\}^{80} \times \{0,1\}^{160} \to \{0,1\}^{160}$ be defined by $F_S(x) = H(S \,\|\, x)$ for all 80-bit S and 160 bit x. We could regard F as a PRF with $s = 80$ and $l = L = 160$. Setting $b = 80$, we could apply Construction 2.4 to get a forward-secure pseudorandom bit generator GEN that outputs 80 pseudorandom bits per stage while using only a single application of the hash function per stage.

Alternatively, and perhaps better for security although more costly, let H be HMAC-SHA-1 [5] and let F: $\{0,1\}^{160} \times \{0,1\}^{160} \to \{0,1\}^{b+160}$ be defined by letting $F_S(x)$ be the first $b+160$ bits of the sequence $H_S(1) \,\|\, H_S(2) \,\|\, \cdots$. We could apply Construction 2.4 to get a forward-secure pseudorandom bit generator GEN that outputs b pseudorandom bits per stage while using $\lceil 1 + b/160 \rceil$ applications of H per stage.

2.6 Number-theoretic constructions

It turns out that existing number-theory based pseudorandom bit generators such as the Blum-Micali [13] and Blum-Blum-Shub [12] generators can be modified so that they are forward secure. More generally, this is true of any generator using the paradigm introduced by [13] 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. We specify the stateful version below.

Construction 2.6 Let $n \geq 1$ be an integer. The constituent algorithms of the stateful BBS generator $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, 1, n)$ are as follows:

```
Algorithm GEN.key
    Pick primes $p, q \equiv 3 \pmod 4$
    $N \leftarrow pq$ ; $x \xleftarrow{\$} \mathsf{Z}_N^*$
    Return $(N, x^2 \bmod N)$
```

```
Algorithm GEN.next($(N, x)$)
    Return $(\text{Parity}(x), (N, x^2 \bmod N))$
```

The key-generation algorithm pick at random primes p, q both congruent to three modulo 4 and having about the same bit-length. The state of this stateful generator is a pair (N, x) where x is a quadratic residue in Z_N^*. Parity(x) is 0 if x is even and 1 if x is odd. The length of an output block is one bit. ▌

Note that in the original generator of [12] 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 in the state.

Forward security of the modified BBS generator can be easily proven, assuming the hardness of factoring numbers N of the form above, based on the results of [12].

3 Forward-secure message authentication

We recall the standard notion of message authentication schemes and their security. We then introduce key-evolving message authentication schemes and a formal notion of forward-security for them. Next we show how a standard message authentication scheme can be transformed into a forward-secure one by using a forward-secure pseudorandom but generator.

3.1 Message authentication schemes

The definitions here follow [7]. A message authentication scheme $\mathsf{mas} = (\mathsf{mas.key}, \mathsf{mas.tag}, \mathsf{mas.vf})$ is specified by its key-generation, tagging and verifying algorithms. We say it has key-length b if the strings (keys) output by $\mathsf{mas.key}$ are always of length b bits. Let f be a forging algorithm that has access to an oracle. Consider the following experiment:

```
Experiment $\mathbf{Exp}^{\mathrm{ma}}_{\mathsf{mas}}(f)$
    $k \xleftarrow{\$} \mathsf{mas.key}$ ; $(M, \tau) \xleftarrow{\$} f^{\mathsf{mas.tag}(k,\cdot)}(\mathsf{find})$
    If $\mathsf{mas.vf}(k, M, \tau) = 1$ and $A$ did not query $M$ to its oracle
        then return 1 else return 0
```

We let

$$\begin{aligned}\mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(f) &= \Pr[\,\mathbf{Exp}^{\mathrm{ma}}_{\mathsf{mas}}(f) = 1\,] \\ \mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(q,t) &= \max_{f}\{\mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(f)\}\ .\end{aligned}$$

The first term is the *ma-advantage* of f in attacking mas. The second term is the *ma-advantage* of mas. The maximum is over all adversaries f that have time-complexity at most t and make at most q oracle queries. As above we adopt the convention that the time-complexity is the total worst-case execution time of the experiment above plus the size of the code of the adversary.

3.2 Forward-secure message authentication schemes

The definitions here are modeled on those for digital signatures [8].

Key-evolving message authentication schemes. A *key-evolving message authentication scheme* $\mathsf{MAS} = (\mathsf{MAS.key}, \mathsf{MAS.tag}, \mathsf{MAS.vf}, \mathsf{MAS.update}, n)$ consists of four algorithms and an integer $n \geq 1$. Randomized algorithm MAS.key is run to obtain the initial key (state) K_0. The operation of the scheme is then divided into stages $i = 1, 2, \ldots, n$, and in stage i the parties use a key denoted K_i. The key at any stage is obtained from the key at the previous stage via the deterministic update algorithm: $K_i \leftarrow \mathsf{MAS.update}(K_{i-1})$. (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 $\langle \tau, i \rangle \leftarrow \mathsf{MAS.tag}(K_i, M)$. (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.) In stage i, a verifier possessing K_i can generate a decision $d \in \{0, 1\}$ to reject or accept a candidate message-tag $(M, \langle \tau, i \rangle))$ via the deterministic verification algorithm: $d \leftarrow \mathsf{MAS.vf}(K_i, M, \tau)$.

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 message authentication scheme via re-keying. After a certain number of message have been tagged under K_{i-1}, it might be advisable to change keys.

Forward-security. 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. Let us now describe and explain the experiment associated to an adversary algorithm F:

```
Experiment $\mathbf{Exp}^{\rm fsma}_{\mathsf{MAS}}(F)$
    $K_0 \stackrel{\$}{\leftarrow} \mathsf{MAS.key}$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i+1$ ; $K_i \leftarrow \mathsf{MAS.update}(K_{i-1})$
        $(d, h) \stackrel{\$}{\leftarrow} F^{\mathsf{MAS.tag}(K_i,\cdot)}(\mathsf{find}, h)$
    Until $(d = \mathsf{forge})$ or $(i = n)$
    $(M, \langle \tau, j\rangle) \stackrel{\$}{\leftarrow} F(\mathsf{forge}, K_i, h)$
    If all the following are true then return 1 else return 0
    – $\mathsf{MAS.vf}(K_j, M, \tau) = 1$
    – $1 \le j < i$
    – $M$ was not queried of $\mathsf{MAS.tag}(K_j, \cdot)$
```

Above, 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 let

$$\begin{aligned} \mathbf{Adv}^{\rm fsma}_{\mathsf{MAS}}(F) &= \Pr[\,\mathbf{Exp}^{\rm fsma}_{\mathsf{MAS}}(F) = 1\,] \\ \mathbf{Adv}^{\rm fsma}_{\mathsf{MAS}}(q,t) &= \max_F\{\mathbf{Adv}^{\rm fsma}_{\mathsf{MAS}}(F)\}\ . \end{aligned}$$

The first term is the *fsma-advantage* of F in attacking MAS. The second term is the *fsma-advantage* of MAS. The maximum is over all adversaries F that have time-complexity at most t and make at most q queries *in each stage.* (So the total number of queries made can reach qn.) This is the maximum likelihood of the forward security of the message authentication scheme MAS being compromised by an adversary using the indicated resources. As above we adopt the convention that the time-complexity is the total worst-case execution time of the experiment above plus the size of the code of the adversary.

3.3 A general construction

We show how a forward secure message authentication scheme can be designed given any secure standard message authentication scheme and forward-secure pseudorandom bit generator. The base key K_0 for the key-evolving message authentication scheme is a seed (initial state) of the generator. The key for stage i 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 key k_i to be used with the standard message authentication scheme, and state information St_i for the generator, based on which the next key will be generated by iteration of the generator. This is detailed below.

Construction 3.1 Let $\mathsf{mas} = (\mathsf{mas.key}, \mathsf{mas.tag}, \mathsf{mas.vf})$ be a standard message authentication scheme with key-length b. Let $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ be a forward secure pseudorandom bit generator with block-length b. We associate to $\mathsf{mas}, \mathsf{GEN}$ the key-evolving message authentication scheme $\mathsf{MAS} = (\mathsf{MAS.key}, \mathsf{MAS.tag}, \mathsf{MAS.vf}, \mathsf{MAS.update}, n)$ whose constituent algorithms are as follows:

```
Algorithm MAS.key
    $St_0 \xleftarrow{\$} \mathsf{GEN.key}$
    Return $(0, \varepsilon, St_0)$

Algorithm MAS.tag$((i, k, St), M)$
    $\tau \leftarrow \mathsf{mas.tag}(k, M)$
    Return $\langle \tau, i \rangle$

Algorithm MAS.vf$((i, k, St), M, \langle \tau, j \rangle)$
    If $j \neq i$ then return 0
    $d \leftarrow \mathsf{mas.vf}(k, M, \tau)$
    Return $d$

Algorithm MAS.update$((i, k, St))$
    $(k, St) \leftarrow \mathsf{GEN.next}(St)$
    Return $(i + 1, k, St)$
```

Above ε denotes the empty string. Recall that GEN.next returns a pair consisting of a pseudorandom output block (here b bits long) and an updated state; the above is saying the pseudorandom block becomes the effective new key for the underlying standard message authentication scheme. ▮

Security. We claim that the key-evolving 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 stateful generator is forward secure. The proof of the following is in Appendix B.

Theorem 3.2 Let $\mathsf{mas} = (\mathsf{mas.key}, \mathsf{mas.tag}, \mathsf{mas.vf})$ be a standard message authentication scheme with key-length b, and $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ a forward secure pseudorandom bit generator with block-length b. Let MAS be the key-evolving message authentication scheme associated to mas, GEN as per Construction 3.1. Then

$$\mathbf{Adv}^{\mathrm{fsma}}_{\mathsf{MAS}}(q, t) \ \leq \ \mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t_1) + n \cdot \mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(q, t_2) \ ,$$

where $t_1 = t_2 = 2t + O(n + b)$. ▮

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 key-evolving constructions already highlighted in [1].

4 Forward-secure audit logs

Computer audit logs contain descriptions of noteworthy events — crashes of system programs, system resource exhaustion, failed login attempts, etc. The ability 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 modifying 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 message authentication schemes 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 message authentication schemes. 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 naturally 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 message authentication schemes. 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 message authentication 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 $\langle M, \mathsf{MAS.tag}(0, j, M)\rangle$. The sequence counter is then incremented.

To update the logging system to a new stage, we first record a log entry $\langle \varepsilon, \mathsf{MAS.tag}(1, j)\rangle$ prior to running $\mathsf{MAS.update}$. 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 $\mathsf{MAS.update}$, the sequence number is reset to zero.

Verifying the log entries requires 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 $\mathsf{MAS.update}$ 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 message authentication 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 message authentication 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 solution is provided in [26]. 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, while the construction of [26] has only heuristic security arguments.

5 Forward-secure encryption

We recall the standard notion of symmetric encryption schemes and their security. We then introduce key-evolving symmetric encryption schemes and a formal notion of forward-security for them. Next we show how a standard symmetric encryption scheme can be transformed into a forward-secure one by using a forward-secure pseudorandom but generator. For simplicity we restrict attention to chosen-plaintext attacks, although the constructions and results extend to the chosen-ciphertext attack case.

5.1 Symmetric encryption schemes

The definitions here follow [6]. A symmetric encryption scheme $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ is specified by its key-generation, encryption and decryption algorithms. We say it has key-length b if the strings (keys) output by $\mathsf{sym.key}$ are always of length b bits. Let B be an adversary algorithm that has access to an oracle. Consider the following experiment:

```
Experiment $\mathbf{Exp}^{\text{ind-cpa}}_{\mathsf{sym}}(B)$
    $k \overset{\$}{\leftarrow} \mathsf{sym.key}$
    $(m_0, m_1, h) \overset{\$}{\leftarrow} B^{\mathsf{sym.enc}(k,\cdot)}(\mathsf{find})$
    $c \overset{\$}{\leftarrow} \{0,1\}$ ; $C \overset{\$}{\leftarrow} \mathsf{sym.enc}(k, m_c)$
    $g \overset{\$}{\leftarrow} B^{\mathsf{sym.enc}(k,\cdot)}(\mathsf{guess}, C, h)$
    If $g = c$ then return 1 else return 0
```

It is required that the messages m_0, m_1 produced by B in its find stage have equal length. We let

$$\begin{aligned}\mathbf{Adv}^{\text{ind-cpa}}_{\mathsf{sym}}(B) &= 2 \cdot \Pr[\,\mathbf{Exp}^{\text{ind-cpa}}_{\mathsf{sym}}(B) = 1\,] - 1 \\ \mathbf{Adv}^{\text{ind-cpa}}_{\mathsf{sym}}(q,t) &= \max_{B}\{\mathbf{Adv}^{\text{ind-cpa}}_{\mathsf{sym}}(B)\} \,.\end{aligned}$$

The first term is the *ind-cpa-advantage* of B in attacking mas. The second term is the *ind-cpa-advantage* of sym. The maximum is over all adversaries B that have time-complexity at most t and make at most q oracle queries. As above we adopt the convention that the time-complexity is the total worst-case execution time of the experiment above plus the size of the code of the adversary.

5.2 Forward-secure symmetric encryption schemes

Key-evolving symmetric encryption schemes. A *key-evolving symmetric encryption scheme* $\mathsf{SYM} = (\mathsf{SYM.key}, \mathsf{SYM.enc}, \mathsf{SYM.dec}, \mathsf{SYM.update}, n)$ consists of four algorithms and an integer $n \geq 1$. Randomized algorithm $\mathsf{SYM.key}$ is run to obtain the initial key (state) K_0. The operation of the scheme is then divided into stages $i = 1, 2, \ldots, n$, and in stage i the parties use a key denoted K_i. The key at any stage is obtained from the key at the previous stage via the deterministic update algorithm: $K_i \leftarrow \mathsf{SYM.update}(K_{i-1})$. (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 encrypt a message M via $\langle C, i\rangle \leftarrow \mathsf{SYM.enc}(K_i, M)$. (Notice that the stage number i is always part of the ciphertext. This is in order to tell the decryptor which key to use for decryption. In order to put i in the ciphertext, its value must be obtainable from K_i, and indeed we will always make sure K_i contains i.) In stage i a decryptor possessing K_i can decrypt $\langle C, i\rangle)$ via $M \leftarrow \mathsf{SYM.dec}(K_i, C)$.

Forward-security. Privacy of data encrypted under K_j must be maintained even if the adversary is in possession of K_i for any $i > j$. To capture this, let us now describe and explain the experiment associated to an adversary algorithm E:

```
Experiment $\mathbf{Exp}^{\text{fsind-cpa}}_{\mathsf{SYM}}(E)$
    $K_0 \xleftarrow{\$} \mathsf{SYM.key}$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i + 1$ ; $K_i \leftarrow \mathsf{SYM.update}(K_{i-1})$
        $(d, (m_0, m_1, j), h) \xleftarrow{\$} E^{\mathsf{SYM.enc}(K_i,\cdot)}(\mathsf{find}, h)$
    Until $(d = \mathsf{guess})$ or $(i = n)$
    $c \xleftarrow{\$} \{0,1\}$
    If $j \geq i$ then return $c$
    Else
        $C \xleftarrow{\$} \mathsf{SYM.enc}(K_j, m_c)$ ; $g \xleftarrow{\$} E(\mathsf{guess}, K_i, C, h)$
        If $g = c$ then return $1$ else return $0$
```

Adversary 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 ciphertext C. E is then given the key K_i (from the break-in) and C, and wins if it guesses g. We let

$$\begin{aligned}
\mathbf{Adv}^{\text{fsind-cpa}}_{\mathsf{SYM}}(E) &= 2 \cdot \Pr[\,\mathbf{Exp}^{\text{fsind-cpa}}_{\mathsf{SYM}}(E) = 1\,] - 1 \\
\mathbf{Adv}^{\text{fsind-cpa}}_{\mathsf{SYM}}(q, t) &= \max_{E}\{\mathbf{Adv}^{\text{fsind-cpa}}_{\mathsf{SYM}}(E)\}\ .
\end{aligned}$$

The first term is the *fsind-cpa-advantage* of E in attacking SYM. The second term is the *fsind-cpa-advantage* of SYM. The maximum is over all adversaries E that have time-complexity at most t and make at most q queries *in each stage.* (So the total number of queries made can reach qn.) This is the maximum likelihood of the forward security of the symmetric encryption scheme SYM being compromised by an adversary using the indicated resources. As above we adopt the convention that the time-complexity is the total worst-case execution time of the experiment above plus the size of the code of the adversary.

5.3 A general construction

We show how a forward secure message authentication scheme can be designed given any secure standard message authentication scheme and forward-secure pseudorandom bit generator, following the same paradigm used in the case of message authentication. The base key K_0 for the key-evolving symmetric encryption scheme is a seed (initial state) of the generator. The key for stage i 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 key k_i to be used with the standard symmetric encryption scheme, and state information St_i for the generator, based on which the next key will be generated by iteration of the generator. This is detailed below.

Construction 5.1 Let $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ be a standard symmetric encryption scheme with key-length b. Let $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ be a forward secure pseudorandom bit generator with block-length b. We associate to $\mathsf{sym}, \mathsf{GEN}$ the key-evolving message symmetric encryption scheme $\mathsf{SYM} = (\mathsf{SYM.key}, \mathsf{SYM.enc}, \mathsf{SYM.dec}, \mathsf{SYM.update}, n)$ whose constituent algorithms are as follows:

```
Algorithm SYM.key
    $St_0 \xleftarrow{\$} \mathsf{GEN.key}$
    Return $(0, \varepsilon, St_0)$
```

```
Algorithm SYM.enc($(i, k, St), M$)
    $C \xleftarrow{\$} \mathsf{sym.enc}(k, M)$
    Return $\langle C, i \rangle$
```

```
Algorithm SYM.dec($(i, k, St), \langle C, j \rangle$)
    If $j \neq i$ then return $\perp$
    $M \leftarrow \mathsf{sym.dec}(k, C)$
    Return $M$
```

```
Algorithm SYM.update($(i, k, St)$)
    $(k, St) \leftarrow \mathsf{GEN.next}(St)$
    Return $(i + 1, k, St)$
```

Recall that $\mathsf{GEN.next}$ returns a pair consisting of a pseudorandom output block (here b bits long) and an updated state; the above is saying the pseudorandom block becomes the effective new key for the underlying standard symmetric encryption scheme. ▮

Security. We claim that the key-evolving 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 5.2 Let $\mathsf{sym} = (\mathsf{sym.key}, \mathsf{sym.enc}, \mathsf{sym.dec})$ be a standard symmetric encryption scheme with key size b, and $\mathsf{GEN} = (\mathsf{GEN.key}, \mathsf{GEN.next}, b, n)$ a forward-secure pseudorandom bit generator with block-length b. Let SYM be the key-evolving symmetric encryption scheme associated to $\mathsf{sym}, \mathsf{GEN}$ as per Construction 5.1. Then

$$\mathbf{Adv}^{\mathsf{fsind\text{-}cpa}}_{\mathsf{SYM}}(q, t) \;\leq\; \mathbf{Adv}^{\mathsf{fsprg}}_{\mathsf{GEN}}(t_1) + n \cdot \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{sym}}(q, t_2) \ ,$$

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

References

[1] M. Abdalla and M. Bellare, "Increasing the lifetime of a key: A comparative analysis of the security of rekeying techniques." *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000.

[2] Alleged RC4. `http://home.earthlink.net/~neilbawd/arcfour.html`.

[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] D. Beaver and S. Haber, "Cryptographic protocols provably secure against dynamic adversaries," *Advances in Cryptology – EUROCRYPT '92*, Lecture Notes in Computer Science Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.

[5] M. Bellare, R. Canetti and H. Krawczyk, "Keying hash functions for message authentication," *Advances in Cryptology – CRYPTO '96*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

[6] 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.

[7] M. Bellare, J. Kilian and P. Rogaway, "The security of cipher block chaining," *Journal of Computer and System Sciences*, Vol. 61, No. 3, Dec 2000, pp. 362–399.

[8] 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.

[9] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm," *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000.

[10] M. Bellare and B. Yee, "Forward-security in private-key cryptography," Preliminary version of this paper, in *Topics in Cryptology – CT-RSA '03*, Lecture Notes in Computer Science Vol. ?? , M. Joye ed., Springer-Verlag, 2003.

[11] J. Black, S. Halevi, H. Krawczyk, T. Krovetz and P. Rogaway, "UMAC: Fast and Secure Message Authentication," *Advances in Cryptology – CRYPTO '99*, Lecture Notes in Computer Science Vol. 1666, M. Wiener ed., Springer-Verlag, 1999.

[12] 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.

[13] 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.

[14] 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.

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

[16] A. Desai, A. Hevia and L. Yin, "A Practice-Oriented Treatment of Pseudorandom Number Generators," *Advances in Cryptology – EUROCRYPT '02*, Lecture Notes in Computer Science Vol. 2332 , L. Knudsen ed., Springer-Verlag, 2002.

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

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

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

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

[21] 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.

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

[23] 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.

[24] J. Katz, "A forward-secure public-key encryption scheme," Cryptology ePrint Archive: Report 2002/060, May 2002, `http://eprint.iacr.org/2002/060/`.

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

[26] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," ACM TISSEC, Vol. 2, 1999. Preliminary version in *Proceedings of the 7th USENIX Security Symposium*, USENIX Press, 1998.

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

A Proof of Theorem 2.3

Let A be an adversary attacking the forward security of GEN and having time-complexity at most t. We want to upper bound $\mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(A)$. We begin by defining the following sequence of hybrid experiments:

```
Experiment $\mathbf{Exp}^{\mathrm{h}\text{-}(1,j)}_{\mathsf{GEN}}(A)$      $(0 \le j \le n)$
    $St \xleftarrow{\$} \{0,1\}^s$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i+1$
        If $i \le j$ then $Out_i \xleftarrow{\$} \{0,1\}^b$ else $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
        $(d, h) \xleftarrow{\$} A(\mathsf{find}, Out_i, h)$
    Until $(d = \mathsf{guess})$ or $(i = n)$
    $g \xleftarrow{\$} A(\mathsf{guess}, St, h)$
    Return $g$
```

For $j = 0, \ldots, n$ we let

$$P_{1,j} \;=\; \Pr[\,\mathbf{Exp}^{\mathrm{h}\text{-}(1,j)}_{\mathsf{GEN}}(A) = 1\,]\ .$$

Note that the experiments $\mathbf{Exp}^{\mathrm{fsprg}\text{-}1}_{\mathsf{GEN}}(A)$ and $\mathbf{Exp}^{\mathrm{h}\text{-}(1,0)}_{\mathsf{GEN}}(A)$ are equivalent. This means that

$$P_{1,0} \;=\; \Pr[\,\mathbf{Exp}^{\mathrm{fsprg}\text{-}1}_{\mathsf{GEN}}(A) = 1\,]\ .$$

Note however that the experiments $\mathbf{Exp}^{\mathrm{fsprg}\text{-}0}_{\mathsf{GEN}}(A)$ and $\mathbf{Exp}^{\mathrm{h}\text{-}(1,n)}_{\mathsf{GEN}}(A)$ also *not* equivalent. The reason is that in the former, 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 the latter, 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 $\mathbf{Exp}^{\mathrm{h}\text{-}(1,n)}_{\mathsf{GEN}}(A)$ left off, and bridges the gap to $\mathbf{Exp}^{\mathrm{fsprg}\text{-}0}_{\mathsf{GEN}}(A)$:

```
Experiment $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,j)}(A)$    $(0 \le j \le n)$
    $St \xleftarrow{\$} \{0,1\}^s$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i+1$
        If $i \le j$ then $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
        $Out_i \xleftarrow{\$} \{0,1\}^b$
        $(d,h) \xleftarrow{\$} A(\mathsf{find}, Out_i, h)$
    Until $(d = \mathsf{guess})$ or $(i = n)$
    $g \xleftarrow{\$} A(\mathsf{guess}, St, h)$
    Return $g$
```

For $j = 0, \ldots, n$ we let

$$P_{2,j} \;=\; \Pr[\,\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,j)}(A) = 1\,]\;.$$

Note that the experiments $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,0)}(A)$ and $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(1,n)}(A)$ are identical. This means that $P_{2,0} = P_{1,n}$. Also the experiments $\mathbf{Exp}_{\mathsf{GEN}}^{\text{fsprg-0}}(A)$ and $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,n)}(A)$ are identical, so

$$P_{2,n} \;=\; \Pr[\,\mathbf{Exp}_{\mathsf{GEN}}^{\text{fsprg-0}}(A) = 1\,]\;.$$

Putting all this together we have

$$\begin{aligned}\mathbf{Adv}_{\mathsf{GEN}}^{\text{fsprg}}(A) &= \Pr[\,\mathbf{Exp}_{\mathsf{GEN}}^{\text{fsprg-1}}(A) = 1\,] - \Pr[\,\mathbf{Exp}_{\mathsf{GEN}}^{\text{fsprg-0}}(A) = 1\,] \\ &= P_{1,0} - P_{2,n} \\ &= [\,P_{1,0} - P_{1,n}\,] + [\,P_{2,0} - P_{2,n}\,]\;. \qquad (1)\end{aligned}$$

We now claim that

$$\begin{aligned}P_{1,0} - P_{1,n} &\le n \cdot \mathbf{Adv}_{G}^{\text{prg}}(t') \qquad (2)\\ P_{2,0} - P_{2,n} &\le n \cdot \mathbf{Adv}_{G}^{\text{prg}}(t')\;. \qquad (3)\end{aligned}$$

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

$$\mathbf{Adv}_{\mathsf{GEN}}^{\text{fsprg}}(A) \;\le\; 2n \cdot \mathbf{Adv}_{G}^{\text{prg}}(t')\;.$$

Since A was an arbitrary adversary with time-complexity at most 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 $D_1(x \,\|\, y)$    $(|x| = b$ and $|y| = s)$
    $j \xleftarrow{\$} \{1, \ldots, n\}$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i+1$
        If $i < j$ then $Out_i \xleftarrow{\$} \{0,1\}^b$
        If $i = j$ then $(Out_i, St) \leftarrow (x, y)$
        If $i > j$ then $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
        $(d,h) \leftarrow A(\mathsf{find}, Out_i, h)$
    Until $(d = \mathsf{guess})$ or $(i = n)$
    $g \leftarrow A(\mathsf{guess}, St, h)$
    Return $g$
```

Suppose we run experiment $\mathbf{Exp}_G^{\text{prg-0}}(D_1)$. We notice that it amounts to running $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(1,j)}(A)$ where j is the value chosen at random by D_1 in its first step. Similarly if we run experiment $\mathbf{Exp}_G^{\text{prg-1}}(D_1)$ we notice that it amounts to running $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(1,j-1)}(A)$ where j is the value chosen at random by D_1 in its first step. So

$$\begin{aligned}\Pr[\,\mathbf{Exp}_G^{\text{prg-1}}(D_1)=1\,] &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j-1}\\ \Pr[\,\mathbf{Exp}_G^{\text{prg-0}}(D_1)=1\,] &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j}\end{aligned}$$

.

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

$$\frac{P_{1,0}-P_{1,n}}{n} \;=\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j-1} \;-\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n} P_{1,j} \;=\; \mathbf{Adv}_G^{\text{prg}}(D_1)\;.$$

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 $D_2(x \,\|\, y)$   $(|x| = b$ and $|y| = s)$
    $j \xleftarrow{\$} \{1,\ldots,n\}$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    Repeat
        $i \leftarrow i+1$
        If $i = 1$ then $(Out_i, St) \leftarrow (x, y)$
        If $1 < i \le j$ then $(Out_i, St) \leftarrow \mathsf{GEN.next}(St)$
        $Out_i \xleftarrow{\$} \{0,1\}^b$
        $(d, h) \xleftarrow{\$} A(\mathsf{find}, Out_i, h)$
    Until $(d = \mathsf{guess})$ or $(i = n)$
    $g \xleftarrow{\$} A(\mathsf{guess}, St, h)$
    Return $1 - g$
```

Suppose we run experiment $\mathbf{Exp}_G^{\text{prg-1}}(D_2)$. We notice that it amounts to running $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,j)}(A)$ 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 $\mathbf{Exp}_G^{\text{prg-0}}(D_2)$ we notice that it amounts to running $\mathbf{Exp}_{\mathsf{GEN}}^{\text{h-}(2,j-1)}(A)$ where j is the value chosen at random by D_2 in its first step, and then flipping the answer bit. So

$$\begin{aligned}\Pr[\,\mathbf{Exp}_G^{\text{prg-1}}(D_2)=1\,] &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(1-P_{2,j})\\ \Pr[\,\mathbf{Exp}_G^{\text{prg-0}}(D_2)=1\,] &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(1-P_{2,j-1})\;.\end{aligned}$$

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

$$\frac{P_{2,0}-P_{2,n}}{n} \;=\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(1-P_{2,j}) \;-\; \tfrac{1}{n}\textstyle\sum_{j=1}^{n}(1-P_{2,j-1}) \;=\; \mathbf{Adv}_G^{\text{prg}}(D_2)\;.$$

Note that the time-complexity 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.

B Proof of Theorem 3.2

Let F be a forger attacking the forward security of the MAS scheme and having time-complexity at most t. We want to upper bound $\mathbf{Adv}_{\mathsf{MAS}}^{\text{fsma}}(F)$. To do this we specify an adversary A attacking the forward security of the generator GEN, and a forger f attacking the mas scheme, and then bound the fsma-advantage of F in terms of the ma-advantage of f and the fsprg-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 $\mathbf{Exp}^{\mathrm{fsma}}_{\mathsf{MAS}}(F)$ by letting the blocks it receives play the role of the keys k_i that are used by mas in the MAS 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 mas otherwise. The algorithm A is below, and explanations follow.

```
Algorithm $A(\mathsf{find}, Out, h)$
    if $h = \varepsilon$ then $h \leftarrow \varepsilon \,\|\, \varepsilon \,\|\, \varepsilon \,\|\, 0$
    Parse $h$ as $h_F \,\|\, h_O \,\|\, h_T \,\|\, i$
    $h_O \leftarrow h_O \,\|\, Out$ ; $i \leftarrow i + 1$
    $(d, h_F) \leftarrow F^{\langle \mathsf{mas.tag}(Out,\cdot), i\rangle}(\mathsf{find}, h_F)$
    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 $\mathsf{mas.vf}(Out_j, M, \tau) = 1$ and $1 \le j < i$ and
            $M$ was not queried of $\mathsf{mas.tag}(Out_j, \cdot)$
        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 $F^{\langle \mathsf{mas.tag}(Out,\cdot), i\rangle}(\mathsf{find}, h_F)$ means that F is given the oracle that on input M returns $\langle \mathsf{mas.tag}(Out, M), i\rangle$. ($A$ can simulate this oracle since it knows Out.) Thus A is simulating the tagging oracle $\mathsf{MAS.tag}_{K_i}$ that F gets at this stage in the right way given the definition of MAS. 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 $\mathbf{Exp}^{\mathrm{fsprg\text{-}1}}_{\mathsf{GEN}}(A)$ and $\mathbf{Exp}^{\mathrm{fsma}}_{\mathsf{MAS}}(F)$ are identical. So

$$\Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}1}}_{\mathsf{GEN}}(A) = 1\,] \;=\; \Pr[\,\mathbf{Exp}^{\mathrm{fsma}}_{\mathsf{MAS}}(F) = 1\,]\,. \tag{4}$$

THE FORGER f. We design a forging algorithm f attacking the given scheme mas. It gets an oracle for $\mathsf{mas.tag}(k,\cdot)$ 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 $\mathsf{mas.tag}(k,\cdot)$. 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 $f^{\mathsf{mas.tag}(k,\cdot)}$
    $l \leftarrow \{1, \ldots, n\}$ ; $St_0 \leftarrow \mathsf{GEN.key}$ ; $i \leftarrow 0$ ; $h \leftarrow \varepsilon$
    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 $(d,h) \leftarrow F^{\langle \mathsf{mas.tag}(k,\cdot), i\rangle}(\mathsf{find}, h)$
            Else $(d,h) \leftarrow F^{\langle \mathsf{mas.tag}(k_i,\cdot), i\rangle}(\mathsf{find}, h)$
    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 $(M, \tau)$ else abort
```

Notice that $\mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(f) = \Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}0}}_{\mathsf{GEN}}(A) = 1\,]/n$. That is

$$\Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}0}}_{\mathsf{GEN}}(A) = 1\,] \;=\; n \cdot \mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(f)\;. \tag{5}$$

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

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{fsma}}_{\mathsf{MAS}}(F) &= \Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}1}}_{\mathsf{GEN}}(A) = 1\,] \\
&= \mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(A) + \Pr[\,\mathbf{Exp}^{\mathrm{fsprg\text{-}0}}_{\mathsf{GEN}}(A) = 1\,] \\
&= \mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(A) + n \cdot \mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(f)\;.
\end{aligned}$$

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

$$\mathbf{Adv}^{\mathrm{fsma}}_{\mathsf{MAS}}(t,q) \;\le\; \mathbf{Adv}^{\mathrm{fsprg}}_{\mathsf{GEN}}(t_1) + n \cdot \mathbf{Adv}^{\mathrm{ma}}_{\mathsf{mas}}(q, t_2)\;,$$

as desired.