A preliminary version of this paper appears in the proceedings of ESORICS 2013. This is the full version.

Practical Secure Logging: Seekable Sequential Key Generators

Giorgia Azzurra Marson[1] and Bertram Poettering[2]

[1] CASED & TU Darmstadt
[2] Information Security Group at Royal Holloway, University of London

Abstract. In computer forensics, log files are indispensable resources that support auditors in identifying and understanding system threats and security breaches. If such logs are recorded locally, i.e., stored on the monitored machine itself, the problem of *log authentication* arises: if a system intrusion takes place, the intruder might be able to manipulate the log entries and cover her traces. Mechanisms that cryptographically protect collected log messages from manipulation should ideally have two properties: they should be *forward-secure* (the adversary gets no advantage from learning current keys when aiming at forging past log entries), and they should be *seekable* (the auditor can verify the integrity of log entries in any order or access pattern, at virtually no computational cost).
We propose a new cryptographic primitive, a *seekable sequential key generator* (SSKG), that combines these two properties and has direct application in secure logging. We rigorously formalize the required security properties and give a provably-secure construction based on the integer factorization problem. We further optimize the scheme in various ways, preparing it for real-world deployment. As a byproduct, we develop the notion of a *shortcut one-way permutation* (SCP), which might be of independent interest.
Our work is highly relevant in practice. Indeed, our SSKG implementation has become part of the logging service of the *systemd* system manager, a core component of many modern commercial Linux-based operating systems.

1 Introduction

Pseudorandom generators. A pseudorandom generator (PRG) is an unkeyed cryptographic primitive that deterministically expands a fixed-length random seed to a longer random-looking string [18]. Most often, PRGs find application in environments where truly random bits are a scarce resource; for instance, once a system managed to harvest an initial seed of, say, 128 uniformly distributed bits from a suitable (possibly physical) entropy source, a PRG can securely stretch this seed to a much larger number of bits. While such mechanisms are indispensable for constrained devices like smartcards, (variants of) PRGs are also long-serving components of modern PC operating systems. A well-known example is the `/dev/urandom` device available in virtually all current Linux/UNIX derivates.

Other applications exploit the feature that the output of PRGs can be *re-generated*: as PRGs are deterministic primitives, the entire output sequence can be reconstructed from the initial seed, whenever needed. This directly allows employment of PRGs for symmetric encryption (formally, one could view stream ciphers like RC4[3] or AES-CTR as PRGs with practically infinite output length), but also in distributed systems, where locally separate agents can synchronously generate identical sequences of (pseudo-)random bits.

For PRGs with very large output length (e.g., stream ciphers) we introduce the notion of seekability; a PRG is *seekable* if, for a fixed seed, 'random access' to the output sequence is possible. For example, the PRG obtained by operating a block cipher in counter mode is seekable: one can quickly jump to any part of the output string by setting the counter value

[3] In fact, practical distinguishing attacks against RC4 are known [11]; RC4 is hence a PRG only 'syntax-wise'.

to the right 'address'. In contrast, RC4 is not known to be seekable: presumably, in order to jump to position k in the output string, one has to iterate the cipher k times.

Forward security. The concept of forward security (FS), best-known from the context of cryptographic key establishment (KE), expresses the inability of an adversary to gain advantage from the 'corruption' of entities. For example, consider an instance of a two-party public key-authenticated KE protocol. We say that the established session key enjoys *forward security*[4] if an adversary cannot obtain any useful information about that key, even if participants, after completing the protocol instance, surrender their respective secret keys. In key exchange, forward security is recognized as one of the most fundamental security goals [29,8].

Although less commonly seen, the notion of forward security extends to other cryptographic settings and primitives. For instance, in *forward-secure public key encryption* (FS-PKE, [7]), time is subdivided into a discrete number of epochs $t_0, t_1, \ldots$, and messages are encrypted in respect to a combination (pk, t_k) of public key and time epoch. Recipients, starting in epoch t_0 with an initial key sk_0, use an update procedure $sk_{i+1} \leftarrow f(sk_i)$ to *evolve* the decryption key from epoch to epoch. An FS-PKE is *correct* if a recipient holding key sk_k can decrypt all ciphertexts addressed to corresponding epoch t_k; it is *forward-secure* if secrecy of all messages addressed to 'past' epochs $t_j, j < k$, is preserved even if the adversary obtains a copy of sk_k. Clearly, FS-PKE only offers a security advantage over plain public key encryption if users securely erase 'expired' decryption keys.

Similarly to FS-PKE, also *forward-secure signature schemes* [2] work with time epochs and evolving keys; briefly speaking, their security properties ensure that an adversary holding an epoch's signing key sk_k cannot forge signatures for prior epochs $t_j, j < k$ (i.e., 'old' signatures remain secure).

Secure logging. Computer log files, whether manually or mechanically evaluated, are among the most essential resources that support system administrators in their day-to-day business. Such files are generated on hosts and communication systems, and record a large variety of system events, including users logging on or off, network requests, memory resources reaching their capacity, malfunctioning of disk drives, and crashing software.

While regular analysis of system logs allows administrators to maintain systems' health and long uptimes, log files are also indispensable in computer forensics, for the identification and comprehension of system intrusions and other security breaches. However, if logs are recorded locally (i.e., on the monitored machine itself) the problem of *log authentication* arises: if a system intrusion takes place, the intruder might be able to manipulate the log entries and cover her traces. So-called 'log sanitizers' aim at frustrating computer forensics and are known to be a standard tool in hackers' arsenal.

Two approaches to avert the threat of adversarial modification of audit logs seem promising. One such option is the deployment of *online logging*. Here, log messages are transferred over a network connection to a remote log sink immediately after their creation, in the expectancy that entries caused by system intrusions have reached their destination before they can be tampered with. As a side effect, online logging might also ease security auditing by the fact that log entries are concentrated at a single point. However, as every local buffering of log records increases the risk of their suppression by the intruder, full-time availability of the log sink is an absolute security requirement in this setting. But observe that the intruder might be able provoke downtimes at the sink (e.g., by running a DOS attack against it) or might disrupt the network connection to it (e.g., by injecting reset packets into TCP connections, jamming wireless connections, etc.). An independent problem comes from the difficulty to select an appropriate level of granularity for the events to be logged. For instance, log files created for forensic analysis might ideally contain verbose information like an individual entry for every file opened, every subprocess started, and so on. Network connections and log sinks might quickly reach their capacities if events are routinely reported

[4] in the context of key establishment also known as 'forward secrecy'

in such a high resolution. This holds in particular if log sinks serve multiple monitored hosts simultaneously.

Storing high volume log data is less an issue in *secured local logging* where a networked log sink is not assumed. In such a setting, log messages are protected from adversarial tampering by cryptographic means. It cannot be expected that standard integrity-protecting primitives like message authentication codes (MAC) or signature schemes on their own will suffice to solve the problem of log authentication: a skilled intruder will likely manage to extract corresponding secret keys from corrupted system's memory. Instead, forward-secure signatures and forward-secure message authentication schemes have been proposed for secure logging [28,24,34]. Clearly, local logging can never prevent the intruder from deleting stored entries. However, cryptographic components might ensure that such manipulations are guaranteed to be indicated to the log auditor.

In practice, system administrators are not required to opt for either the one or the other approach: online logging and secured local logging smoothly work in combination. For instance, log events might be captured with high granularity and stored locally, protected by an appropriate forward-secure integrity protection. In addition, a specified subset of events (like login failures) might be forwarded for analysis to a central sink. Every downtime of the sink can be bridged using local buffering, as all entries stored in these buffers are cryptographically integrity-protected.

1.1 Contributions, organization, applications

The key contribution of this paper is the development of a new cryptographic primitive: a *seekable sequential key generator* (SSKG). Briefly, a *sequential key generator* (SKG) is a stateful PRG that outputs a sequence of fixed-length strings — one per invocation. The essential security property is indistinguishability of these strings from uniformly random. For SSKG, we additionally require seekability, i.e., the existence of an efficient algorithm that allows to jump to any position in the output sequence. For both, SKG and SSKG, we demand that indistinguishability hold with forward security.

This paper is organized as follows. We start in Sections 2 and 3 by formalizing the functionality and security properties of SKG and SSKG. We show that a related primitive by Bellare and Yee securely instantiates an SKG; however, it is not seekable. Aiming at constructing an SSKG, we introduce in Section 4 an auxiliary primitive, a *shortcut one-way permutation* (SCP), that we instantiate in the factoring-based setting. In Section 5 we expose our SSKG; it is particularly efficient, taking one modular squaring operation and one hash function evaluation per invocation. We conclude in Section 6 by proposing further optimizations that substantially increase efficiency of our SSKG, making it ready for deployment in practice.

We argue that a (seekable) SKG is the ideal primitive to implement a secured local logging system, as described above. The construction is immediate: the strings output by the SKG are used as keys for a MAC which is applied to all log messages. After each authentication tag has been computed and appended to the particular log message, the SKG is evolved to the next state, making the described authentication forward-secure. The log auditor, starting with a copy of the SKG's original seed, can recover all MAC keys and verify authenticity of all log entries. Typically, log auditors will require random access to these MAC keys — SSKGs provide exactly this functionality.

Further applications for SKGs and SSKGs. Potential applications of SKG and SSKG are given not only by secure logging, but also by digital cameras, voice recorders and backup systems [28]. In more detail, digital cameras could be equipped with an authentication mechanism that individually authenticates every photo taken. Such cameras could support modern journalism that, when reporting from armed conflict zones, is more and more reliant on amateurs for the documentation of events; in such settings, where post-incidental (digital)

manipulation inherently has to be anticipated, cryptographic SKG-like techniques could support the verification of authenticity of reported images.

1.2 Related work

Secured local logging. An early proposal to use forward-secure cryptography to protect locally-stored audit logs is by Kelsey and Schneier [20,21,28]. The core of their scheme is an (evolving) 'authentication key': for each time epoch t_i there is a corresponding authentication key A_i. This key is used for multiple purposes: as a MAC key to authenticate all log messages occurring in epoch t_i, for deriving an epoch-specific encryption key K_i by computing $K_i \leftarrow H_0(A_i)$, and for computing next epoch's authentication key via iteration $A_{i+1} \leftarrow H_1(A_i)$ (where H_0, H_1 are hash functions). The scheme also serves as a basis for the construction of Stathopoulos, Kotzanikolaou, and Magkos [30], who investigate the role of secure logging in public communication networks. Further, an implementation of [28] in tamper-resistant hardware is reported by Chong, Peng and Hartel [9]. Unfortunately, the scheme by Kelsey and Schneier lacks a formal security analysis.[5]

The first rigorous analysis of forward-secure secret key cryptography was given by Bellare and Yee [3]. They propose constructions of forward-secure variants of PRGs, symmetric encryption schemes, and message authentication codes, and analyze them in respect to formal security models. We anticipate here that our security definitions are strictly stronger than those from [3], capturing a larger class of application scenarios.

The work of Holt [14] can be seen as an extension of [28]. With *logcrypt*, the author proposes a symmetric scheme and an asymmetric scheme for secure logging. While the former is similar to [28] (but apparently offers provable security), the latter bases on the forward-secure signature scheme by Bellare and Miner [2]. Holt also discusses the efficiency penalties experienced in the asymmetric variant. We finally note that [14] suggests to store regular *metronome entries* in log files in order to thwart truncation attacks where adversary cuts off the most recent set of log entries.

Ma and Tsudik propose deployment of *forward-secure sequential aggregate signatures* for integrity-protected logging [23,24]. Their provably-secure construction builds on compact constant-size authenticators with all-or-nothing security (i.e., if any single log message is suppressed by the adversary, this will be noticed). Such aggregate signatures naturally defend against truncation attacks, making Holt's metronome entries disposable.

Waters, Balfanz, Durfee, and Smetters [32] identify searchable audit logs as an application of identity-based encryption. Here, in order to increase users' privacy, log entries are not only authenticated but also encrypted. This encryption is done in respect to a set of keywords; records encrypted towards such keywords are identifiable and decryptable by agents who hold keyword-dependent private keys.

Another interesting approach towards forward-secure logging was proposed by Yavuz and Ning [33], and Yavuz, Ning, and Reiter [34]. In their scheme, the key evolving procedure and the computation of (aggregatable) authentication tags take not more than a few hash function evaluations and finite field multiplications each; these steps are hence implementable on sensors and other devices with constrained computing power. However, the required workload on verifier's side is much higher: one exponentiation per log entry.

An IETF-standardized secure logging scheme is *signed syslog messages* by Kelsey, Callas, and Clemm [19]. The authors describe an extension to the standard UNIX *syslog* facility that authenticates log entries via a regular signature scheme (e.g., DSA). The scheme, however, does not provide forward security.

We conclude by recommending Itkis' excellent survey on methods in forward-secure cryptography [16].

[5] It is, in fact, not difficult to see that the scheme is *generically* insecure (i.e., a security proof cannot exist).

Seekable PRGs. We are not aware of any work so far that focuses on the seekability of PRGs. The observation that block ciphers operated in counter mode can be seen as seekable PRGs, in contrast to most other stream ciphers, is certainly folklore. We point out that the famous Blum-Blum-Shub PRG [4,5] is forward-secure. Moreover, its underlying number-theoretic structure seems to allow for seekability. Unfortunately it is not efficient: the computation of each individual output bit requires one modular squaring.

2 Sequential key generators

We introduce *sequential key generators* (SKG) and their security properties. Note that a similar primitive, *stateful generator*, was proposed by Bellare and Yee [3]. However, our syntax is more versatile and our security models are stronger, as we will see. We extend SKGs to (seekable) SSKGs in Section 3.

2.1 Functionality and syntax

An SKG consists of four algorithms: GenSKG generates a set par of public parameters, $\mathsf{GenState0}$ takes par and outputs an initial state st_0, update procedure Evolve maps each state st_i to a successor state st_{i+1}, and GetKey algorithm derives from any state st_i a corresponding (symmetric) key K_i. Keys $K_0, K_1, \ldots$ are supposed to be used in higher level protocols, for example as keys for symmetric encryption or message authentication schemes.

Typically, SKG instances are not run in a single copy; rather, after distributing 'clones' of initial state st_0 to a given set of parties, several copies of the same SKG instance are run concurrently and independently, potentially on different host systems, not necessarily in synchronization. If Evolve and GetKey algorithms are deterministic, respective sequences $K_0, K_1, \ldots$ of computed symmetric keys will be identical for all copies. This setting is illustrated in Figure 1 and formalized as follows.

Definition 1 (Syntax of SKG). *A* sequential key generator *is a tuple* $\mathsf{SKG} = \{\mathsf{GenSKG}, \mathsf{GenState0}, \mathsf{Evolve}, \mathsf{GetKey}\}$ *of efficient algorithms as follows:*

- $\mathsf{GenSKG}(1^\lambda)$. *On input of security parameter* 1^λ*, this algorithm outputs a set* par *of public parameters.*
- $\mathsf{GenState0}(\mathsf{par})$. *On input of public parameters* par*, this algorithm outputs an initial state* st_0.
- $\mathsf{Evolve}(\mathsf{st}_i)$. *On input of state* st_i*, this deterministic algorithm outputs 'next' state* st_{i+1}. *For convenience, for any* $m \in \mathbb{N}$*, by* Evolve^m *we denote the* m*-fold composition of* Evolve*, i.e.,* $\mathsf{Evolve}^m(\mathsf{st}_i) = \mathsf{st}_{i+m}$.
- $\mathsf{GetKey}(\mathsf{st}_i)$. *On input of state* st_i*, this deterministic algorithm outputs key* $K_i \in \{0,1\}^{\ell(\lambda)}$*, for a fixed polynomial* ℓ*. For convenience, for any* $m \in \mathbb{N}$*, we write* $\mathsf{GetKey}^m(\mathsf{st}_i)$ *for* $\mathsf{GetKey}(\mathsf{Evolve}^m(\mathsf{st}_i))$.

We also pose the informal requirement on Evolve algorithm that it securely erase state st_i after deriving state st_{i+1} from it. Note that secure erasure is generally considered difficult to achieve and requires special care [12].

2.2 Security requirements

The fundamental security property of SKGs is the indistinguishability of keys K_i from random strings of the same length. Intuitively, for any n of adversary $\mathcal{A}$'s choosing, target key K_n is required to be indistinguishable from random even if $\mathcal{A}$ has access to all other keys K_i, $i \neq n$. This feature ensures generic composability of SKGs with applications that rely on uniformly and independently distributed keys K_i. In addition to the indistinguishability

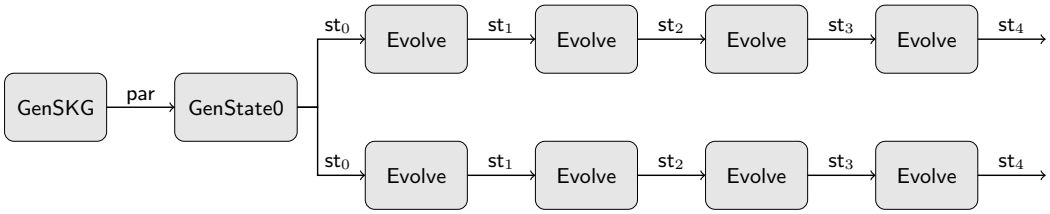

Fig. 1. Interplay of GenSKG, GenState0, and Evolve algorithms of an SKG. The figure shows two copies of the same SKG instance running in parallel. GetKey algorithm can be applied to each intermediate state st_i to derive key K_i.

requirement, forward security demands that an 'old' key K_n remain secure even when $\mathcal{A}$ learns state st_m, for any $m > n$ (e.g., by means of a computer break-in).

We give two game-based definitions of these indistinguishability notions: one with and one without forward security.

Definition 2 (IND and IND-FS security of SKG). *A sequential key generator* SKG *is* indistinguishable against adaptive adversaries (IND) *if for all efficient adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *that interact in experiments* $\mathrm{Expt}^{\mathsf{IND},b}$ *from Figure 2 the following advantage function is negligible, where the probabilities are taken over the random coins of the experiment (including over* $\mathcal{A}$*'s randomness):*

$$\mathrm{Adv}^{\mathsf{IND}}_{\mathsf{SKG},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathrm{Expt}^{\mathsf{IND},1}_{\mathsf{SKG},\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathrm{Expt}^{\mathsf{IND},0}_{\mathsf{SKG},\mathcal{A}}(1^\lambda) = 1\right]\right| \ .$$

The SKG is indistinguishable with forward security against adaptive adversaries (IND-FS) *if analogously defined advantage function* $\mathrm{Adv}^{\mathsf{IND\text{-}FS}}_{\mathsf{SKG},\mathcal{A}}(\lambda)$ *is negligible.*

$\mathrm{Expt}^{\mathsf{IND},b}_{\mathsf{SKG},\mathcal{A}}(1^\lambda)$:
(a) $\mathsf{KList} \leftarrow \emptyset$
(b) $\mathsf{par} \leftarrow_R \mathsf{GenSKG}(1^\lambda)$
(c) $\mathsf{st}_0 \leftarrow_R \mathsf{GenState0}(\mathsf{par})$
(d) $(state, n) \leftarrow_R \mathcal{A}_1^{\mathcal{O}_{\mathsf{Key}}}(\mathsf{par})$
– If $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Key}}(i)$:
(a) $\mathsf{KList} \leftarrow \mathsf{KList} \cup \{i\}$
(b) $K_i \leftarrow \mathsf{GetKey}^i(\mathsf{st}_0)$
(c) Answer $\mathcal{A}$ with K_i
(e) $K_n^0 \leftarrow_R \{0,1\}^{\ell(\lambda)}$
(f) $K_n^1 \leftarrow \mathsf{GetKey}^n(\mathsf{st}_0)$
(g) $b' \leftarrow_R \mathcal{A}_2^{\mathcal{O}_{\mathsf{Key}}}(state, K_n^b)$
– Answer $\mathcal{O}_{\mathsf{Key}}$ queries as above
(h) Return 0 if $n \in \mathsf{KList}$
(i) Return b'

$\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}_{\mathsf{SKG},\mathcal{A}}(1^\lambda)$:
(a) $\mathsf{KList} \leftarrow \emptyset$
(b) $\mathsf{par} \leftarrow_R \mathsf{GenSKG}(1^\lambda)$
(c) $\mathsf{st}_0 \leftarrow_R \mathsf{GenState0}(\mathsf{par})$
(d) $(state, n, m) \leftarrow_R \mathcal{A}_1^{\mathcal{O}_{\mathsf{Key}}}(\mathsf{par})$
– Answer $\mathcal{O}_{\mathsf{Key}}$ queries as in $\mathrm{Expt}^{\mathsf{IND}}$
(e) $K_n^0 \leftarrow_R \{0,1\}^{\ell(\lambda)}$
(f) $K_n^1 \leftarrow \mathsf{GetKey}^n(\mathsf{st}_0)$
(g) $\mathsf{st}_m \leftarrow \mathsf{Evolve}^m(\mathsf{st}_0)$
(h) $b' \leftarrow_R \mathcal{A}_2^{\mathcal{O}_{\mathsf{Key}}}(state, \mathsf{st}_m, K_n^b)$
– Answer $\mathcal{O}_{\mathsf{Key}}$ queries as in $\mathrm{Expt}^{\mathsf{IND}}$
(i) Return 0 if $n \in \mathsf{KList}$ or $m \leq n$
(j) Return b'

Fig. 2. Security experiments for SKG without and with forward security

It is not difficult to see that the IND-FS notion is strictly stronger than the IND notion. The proof of Lemma 1 appears in Appendix A.

Lemma 1 (IND-FS ⇒ IND). *Any sequential key generator* SKG *that is indistinguishable with forward security against adaptive adversaries is also indistinguishable against adaptive adversaries.*

2.3 Comparison with stateful generators

Stateful generators, first described by Bellare and Yee [3, Section 2.2], aim at similar applications as SKGs. Syntactically, the two primitives are essentially identical. However, the security definition of stateful generators is weaker and less versatile than the one of SKGs. Concretely, in the (game-based) security definition for stateful generators, after having incremental access to a sequence $k_0, k_1, \ldots$ of keys that are either all real (i.e., $k_i = K_i \, \forall i$) or all random (i.e., $k_i \in_R \{0,1\}^{\ell(\lambda)} \, \forall i$), the adversary eventually requests to see the 'current' state st_m and, based upon the result, outputs a guess on whether keys $k_0, \ldots, k_{m-1}$ were actually real or random. For reference, a syntactically adjusted version of the model from [3] is reproduced in Appendix B. Important here is the observation that an adversary that corrupts a state st_m *cannot* request access to keys K_i, $i > m$, before making this corruption (in contrast to our model). This is a severe limitation in contexts where multiple parties evolve states of the same SKG instance independently of each other and in an asynchronous manner; for instance, in the secure logging scenario, the adversary might *first* observe the log auditor verifying MAC tags on 'current' time epochs and *then* decide to corrupt a monitored host that is out of synchronization, e.g., because it is powered down and hence didn't evolve its state. As such concurrent and asynchronous conditions are not considered in the model by Bellare and Yee, in some practically relevant settings the security of the constructions from [3] should not be assumed.

2.4 A simple construction

It does not seem difficult to construct SKGs from standard cryptographic primitives. Indeed, many of the stateful generators proposed in [3], constructed from PRGs and PRFs, are in fact IND-FS-secure SKGs. For concreteness, we reproduce a simple PRG-based design. Its security is analysed in [3, Theorem 1].

Construction 1 (PRG-based SKG [3]) *Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell(\lambda)}$ be a PRG, where for each $x \in \{0,1\}^\lambda$ we write $G(x)$ as $G(x) = G_L(x) \parallel G_R(x)$ with $G_L(x) \in \{0,1\}^\lambda$ and $G_R(x) \in \{0,1\}^{\ell(\lambda)}$. Let then* GenSKG *output the empty string,* GenState0 *sample $\mathsf{st}_0 \leftarrow_R \{0,1\}^\lambda$,* Evolve$(\mathsf{st}_i)$ *output $G_L(\mathsf{st}_i)$, and* GetKey(st_i) *output $G_R(\mathsf{st}_i)$.*

3 Seekable sequential key generators

We have seen that secure SKGs exist and are not too difficult to construct. Moreover, the scheme from Construction 1 is efficient. Indeed, if it is instantiated with a hash function-based PRG, invocations of Evolve and GetKey algorithms take only a small (constant) number of hash function evaluations. However, this assessment of efficiency is adequate only if SKG's keys K_i are used (and computed) in sequential order. We argue that in many potential fields of application such access structures are not given; instead, random access to the keys is required, likely implying a considerable efficiency penalty if keys need to be computed iteratively via $K_i \leftarrow \mathsf{GetKey}^i(\mathsf{st}_0)$. The following examples illustrate that random access patterns do not intrinsically contradict the envisioned sequential nature of SKGs.

Consider a host that uses SKG's keys K_i to authenticate continuously incurring log messages. A second copy of the same SKG instance would be run by the log auditor. From time to time the latter might want to check the integrity of an arbitrary selection of these messages[6]. Observe that this scenario does not really correspond to the setting from Figure 1: While the upper SKG copy might represent the host that evolves keys in the expected linear order $K_i \to K_{i+1}$, the auditor (running the independent second copy) would actually need non-sequential access to SKG's keys.

[6] For example, after a zero-day vulnerability in a software product run on the monitored host becomes public, the log auditor might want to retrospectively look for specific irregularities in log entries related to that vulnerability.

For a second example in secure logging, assume SKG's epochs are coupled to absolute time intervals (e.g., one epoch per second). If a host is powered up after a long down-time, in order to resynchronize its SKG state, it is required to do a 'fast-forward' over a large number of epochs. Ideally, an SKG would support the option to skip an arbitrary number of Evolve steps in short time[7].

A variant of SKG that explicitly offers random access capabilities is introduced in this section. We claim that many practical applications can widely benefit from the extended functionality. Observe that the advantage of SSKGs over SKGs is purely efficiency-wise; in particular, the definition of SSKG's security will be (almost) identical to the one for SKGs.

3.1 Functionality and syntax

When comparing to regular SKGs, the distinguishing property of *seekable sequential key generators* (SSKG) is that keys K_i can be computed *directly* from initial state st_0 and index i, i.e., without executing the Evolve procedure i times. The corresponding new algorithm, Seek, and its relation to the other SKG algorithms is visualized in Figure 3. For reasons that will become clear later, when extending SKG's syntax towards SSKG, in addition to introducing the Seek algorithm we also had to slightly adapt the signature of the GenSKG algorithm:

Definition 3 (Syntax of SSKG). *A* seekable sequential key generator *is a tuple* $\mathsf{SSKG} = \{\mathsf{GenSSKG}, \mathsf{GenState0}, \mathsf{Evolve}, \mathsf{Seek}, \mathsf{GetKey}\}$ *of efficient algorithms as follows:*

- $\mathsf{GenSSKG}(1^\lambda)$. *On input of security parameter* 1^λ*, this algorithm outputs a set* par *of public parameters and a seeking key* sk.
- GenState0, Evolve, GetKey *as for SKGs (cf. Definition 1).*
- $\mathsf{Seek}(\mathsf{sk}, \mathsf{st}_0, m)$. *On input of seeking key* sk*, initial state* st_0*, and* $m \in \mathbb{N}$*, this deterministic algorithm returns a state* st_m.

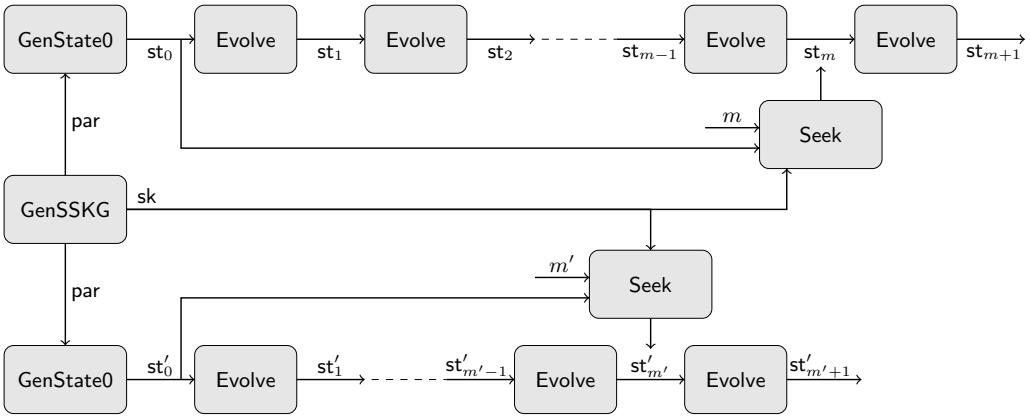

Fig. 3. Interplay of the different SSKG algorithms. The figure shows two independent SSKG instances running in parallel. Given seeking key sk and respective instance's initial state st_0, one can seek directly to any arbitrary state st_m. As in SKGs, GetKey algorithm can be applied to any intermediate state st_i to derive key K_i.

In contrast to SKGs, for SSKGs we need to explicitly require consistency of keys computed with Seek and Evolve algorithms:

[7] Clearly, a (fast-)forward algorithm with execution time linear in the number δ of skipped epochs is trivially achievable. The question is: can we do better than $O(\delta)$?

Definition 4 (Correctness of SSKG). *A seekable sequential key generator* SSKG *is* correct *if, for all* $\lambda \in \mathbb{N}$*, all* $(\mathsf{par}, \mathsf{sk}) \leftarrow_R \mathsf{GenSSKG}(1^\lambda)$*, and all* $\mathsf{st}_0 \leftarrow_R \mathsf{GenState0}(\mathsf{par})$*, we have that* $\mathsf{Seek}(\mathsf{sk}, \mathsf{st}_0, m) = \mathsf{Evolve}^m(\mathsf{st}_0)$ *for all* $m \in \mathbb{N}$*.*

Remark 1 (Security notions IND *and* IND-FS *for SSKG).* Indistinguishability of SSKGs is defined in exactly the same way as for regular SKGs, with one purely syntactical exception: As the new $\mathsf{GenSSKG}$ algorithm outputs the auxiliary seeking key, the experiments in Figure 2 need to be adapted such that the $\mathsf{par} \leftarrow_R \mathsf{GenSKG}(1^\lambda)$ line is replaced by $(\mathsf{par}, \mathsf{sk}) \leftarrow_R \mathsf{GenSSKG}(1^\lambda)$. However, seeking key sk is irrelevant for the rest of the experiment.

Example 1 (Practical SSKG setting). We describe a practical setting of secured local logging with multiple monitored hosts. The system administrator first runs $\mathsf{GenSSKG}$ algorithm to establish system-wide parameters; each host then runs $\mathsf{GenState0}$ algorithm to create its individual initial state st_0, serving as a basis for specific sequences $(\mathsf{st}_i)_{i\in\mathbb{N}}$ and $(K_i)_{i\in\mathbb{N}}$. The log auditor, having access to seeking key sk and to initial states st_0 of all hosts, can reproduce all corresponding keys K_i without restriction. Observe that, as the SSKG instances run on different hosts are independent of each other, authenticated log messages from one host cannot be 'replayed' on other hosts.

In practice, it might be difficult to find 'the right' frequency with which keys should be evolved to the next epoch. Recall that, even if forward-secure log authentication is in place, an intruder cannot be prevented from manipulating the log entries of the epoch in which he got access to a system. This suggests that keys should be updated at least every few seconds — and even more often to obtain protection against fully-automated attack tools. On battery-powered mobile devices, however, too frequently scheduled wakeups from system's sleep mode with the only purpose of evolving keys will noticeably contribute to draining devices' energy reserves.

Remark 2 (On the necessity of seeking trapdoors). For standard SKGs, the secret material managed by users is restricted to one 'current' state st_i. In contrast, for SSKGs, we introduced additional secret information, sk, required to perform the seek operation. One might ask whether this step was really necessary. We fixed the syntax of SSKGs as given in Definition 3 for a technical reason: the SSKG construction we present in Section 5 is factoring-based and its Seek algorithm requires knowledge of modulus' factorization $n = pq$. However, as knowledge of p and q thwarts the one-wayness of designated Evolve operation, we had to formally separate the entities that can and cannot perform the Seek operation. While this property slightly narrows the applicability of SSKGs, it is irrelevant for the intended secure logging scenario as described in Example 1.

4 Shortcut permutations

We introduce a novel primitive, *shortcut one-way permutation* (SCP), that will serve as a building block for our SSKG construction in Section 5. Consider a finite set $\mathcal{D}$ together with an efficient permutation $\pi : \mathcal{D} \to \mathcal{D}$. Clearly, for any $x \in \mathcal{D}$ and $m \in \mathbb{N}$, it is easy to compute the m-fold composition $\pi^m(x) = \pi \circ \cdots \circ \pi(x)$ in linear time $O(m)$, by evaluating the permutation m times. In shortcut permutations, we have the efficiency requirement that the value $\pi^m(x)$ can be computed more efficiently than that, using a dedicated algorithm. In addition, we require one-wayness of π: given $y \in \mathcal{D}$, it should be impossible to compute $\pi^{-1}(y)$.

While we will rigorously specify the one-wayness requirement of SCPs, we do not give a precise definition of what 'more efficiently' means for the computation of π^m. The reason is that we aim at practicality of our construction, and, in general, practical efficiency strongly depends on the concrete parameter sizes and computing platforms in use. However, we anticipate that the SCPs that we construct in Section 4.1 have algorithms that compute $\pi^m(x)$ in constant time.

We next formalize the syntax and functionality of SCPs. For technical reasons, the definition slightly deviates from the above intuition in that the algorithm which efficiently computes π^m also requires an auxiliary input, the *shortcut* information.

Definition 5 (Syntax of SCP). *A* shortcut permutation *is a triple* $\mathsf{SCP} = \{\mathsf{GenSCP}, \mathsf{Eval}, \mathsf{Express}\}$ *of efficient algorithms as follows:*

- $\mathsf{GenSCP}(1^\lambda)$. *This probabilistic algorithm, on input of security parameter* 1^λ*, outputs public parameters* pp *and a corresponding shortcut information* sc*. We assume that each specific value* pp *implicitly defines a finite domain* $\mathcal{D} = \mathcal{D}(\mathsf{pp})$*. We further assume that elements from* $\mathcal{D}$ *can be efficiently sampled with uniform distribution.*
- $\mathsf{Eval}(\mathsf{pp}, x)$. *This deterministic algorithm, given public parameters* pp *and a value* $x \in \mathcal{D}$*, outputs a value* $y \in \mathcal{D}$*.*
- $\mathsf{Express}(\mathsf{sc}, x, m)$. *This deterministic algorithm takes shortcut information* sc*, an element* $x \in \mathcal{D}$*, and a non-negative integer* m*, and returns a value* $y \in \mathcal{D}$*.*

A shortcut permutation SCP *is* correct *if, for all* $\lambda \in \mathbb{N}$ *and all* $(\mathsf{pp}, \mathsf{sc}) \leftarrow_R \mathsf{GenSCP}(1^\lambda)$*, we have that (a)* $\mathsf{Eval}(\mathsf{pp}, \cdot)$ *implements a bijection* $\pi : \mathcal{D} \to \mathcal{D}$*, and (b)* $\mathsf{Express}(\mathsf{sc}, x, m) = \pi^m(x)$*, for all* $x \in \mathcal{D}$ *and* $m \in \mathbb{N}$*.*

As the newly introduced shortcut property is solely an efficiency feature, it does not appear in our specification of one-way security. In fact, the one-wayness definitions of SCPs and of regular one-way permutations [18] are essentially the same. Observe that we model one-wayness only for the case that the adversary does not have access to shortcut information sc.

Definition 6 (One-wayness of SCP). *We say that a shortcut permutation* SCP *is* one-way *if the probability*

$$\Pr\big[(\mathsf{pp}, \mathsf{sc}) \leftarrow_R \mathsf{GenSCP}(1^\lambda); y \leftarrow_R \mathcal{D}(\mathsf{pp}); x \leftarrow_R \mathcal{B}(\mathsf{pp}, y) : \mathsf{Eval}(\mathsf{pp}, x) = y\big]$$

is negligible in λ*, for all efficient adversaries* $\mathcal{B}$*.*

Remark 3 (Comparison of SCPs and TDPs). The syntax of (one-way) SCPs is, to some extent, close to that of trapdoor permutations (TDPs, [18]). However, observe the significant difference between the notions of *trapdoor* and *shortcut*. While a TDP's trapdoor allows efficient *inversion* of the permutation (i.e., computation of π^{-1}), a shortcut in our newly defined primitive allows *acceleration* of the computation of π^m, for arbitrary m. In particular, for SCPs, there might be no way to invert π even if the shortcut information is available. We admit, though, that in our number-theory-based constructions from Section 4.1 one-wayness does not hold for adversaries that obtain the shortcut information: any party knowing the shortcut can also efficiently invert the permutation.

4.1 Constructions based on number theory

We propose two efficient number-theoretic SCP constructions: FACT-SCP and RSA-SCP.

Let N be a Blum integer, i.e., $N = pq$ for primes p, q such that $p \equiv q \equiv 3 \bmod 4$. Let $QR_N = \{x^2 : x \in \mathbb{Z}_N^\times\}$ denote the set of *quadratic residues* modulo N. It is well-known [25] that the squaring operation $x \mapsto x^2 \bmod N$ is a permutation on QR_N. Moreover, computing square roots in QR_N, i.e., inverting this permutation, is as hard as factoring N. This intuition is the basis of the following hardness assumption.

Definition 7 (SQRT assumption). *For probabilistic algorithms* GenSQRT *that take as input security parameter* 1^λ *and output tuples* (N, p, q, φ) *such that* $N = pq$*, factors* p *and*

q are prime and satisfy $p \equiv q \equiv 3 \bmod 4$, and $\varphi = \varphi(N) = |\mathbb{Z}_N^\times|$, the SQRT problem *is said to be* hard *if for all efficient adversaries $\mathcal{A}$ the success probability*

$$\Pr\left[(N, p, q, \varphi) \leftarrow_R \mathsf{GenSQRT}(1^\lambda); y \leftarrow_R QR_N; x \leftarrow_R \mathcal{A}(N, y) : x^2 \equiv y \bmod N\right]$$

is negligible in λ, where the probability is taken over the random coins of the experiment (including $\mathcal{A}$'s randomness). The SQRT assumption *states that there exists an efficient probabilistic algorithm* GenSQRT *for which the* SQRT *problem is hard.*

The construction of an SCP based on the SQRT assumption is now straight-forward:

Construction 2 (FACT-SCP) *Construct* SQRT*-based SCP as follows: Let* $\mathsf{GenSCP}(1^\lambda)$ *run* $\mathsf{GenSQRT}(1^\lambda)$ *and output* $\mathsf{pp} = N$ *and* $\mathsf{sc} = \varphi$*, let $\mathcal{D} = QR_N$, let* $\mathsf{Eval}(N, x)$ *output* $x^2 \bmod N$*, and let* $\mathsf{Express}(\varphi, x, m)$ *output* $x^{(2^m \bmod \varphi)} \bmod N$*.*

Remark 4 (Correctness and security of FACT-SCP*).* Observe that the specified domain $\mathcal{D}$ is efficiently samplable (take $x \leftarrow_R \mathbb{Z}_N^\times$ and square it), that correctness of the SCP follows from standard number-theoretic results (in particular [25, Fact 2.160] and [25, Fact 2.126]), and that every Express operation takes about one exponentiation modulo N. Further, comparing the experiments in Definitions 6 and 7 makes evident that FACT-SCP is one-way if the SQRT problem is hard for GenSQRT, i.e., if integer factorization is hard [25, Fact 3.46].

Similarly to FACT-SCP, in Appendix C we define the RSA-based RSA-SCP. Observe that both constructions rely on different, though related, number-theoretic assumptions. In fact, while the security of FACT-SCP can be shown to be equivalent to the hardness of integer factorization, RSA-SCP can be reduced 'only' to the RSA assumption. Although equivalence of the RSA problem and integer factorization is widely believed, a proof has not been found yet. Hence, in some sense, SQRT-based schemes are more secure than RSA-based schemes. In addition to that, our SQRT-based scheme has a (slight) performance advantage over our RSA-based scheme (squaring is more efficient than raising to the power of e). The only situation we are aware of in which RSA-SCP might have an advantage over FACT-SCP is when the most often executed operation is Express, and deployment of multiprime RSA is acceptable (e.g., $N = pqr$). Briefly, in the multiprime RSA setting [17,13], private key operations can be implemented particularly efficiently, based on the Chinese Remainder Theorem (CRT). Observe that Definition 8 (in Appendix C) is general enough to cover the multiprime setting.

5 Seekable sequential key generation from shortcut permutations

We construct an SSKG from a generic SCP. Briefly, the Evolve operation corresponds to the Eval algorithm, the Seek algorithm is implemented via SCP's Express procedure, and keys K_i are computed by applying a hash function (modeled as a random oracle in the security analysis) to the corresponding state st_i.

Construction 3 (SCP-SSKG) *Let* $\mathsf{SCP} = \{\mathsf{GenSCP}, \mathsf{Eval}, \mathsf{Express}\}$ *be a shortcut permutation, and let $H : \{0,1\}^* \to \{0,1\}^{\ell(\lambda)}$ be a hash function, for a polynomial ℓ. Then the algorithms of our seekable sequential key generator* SCP-SSKG *are specified in Figure 4.*

Correctness of Construction 3 follows by inspection. We state IND-FS security of SCP-SSKG in Theorem 1; the corresponding proof appears in Appendix D. Recall that IND security follows by Lemma 1.

Theorem 1 (Security of SCP-SSKG**).** *The SSKG from Construction 3 offers* IND-FS *security if* SCP *is a one-way shortcut permutation, in the random oracle model.*

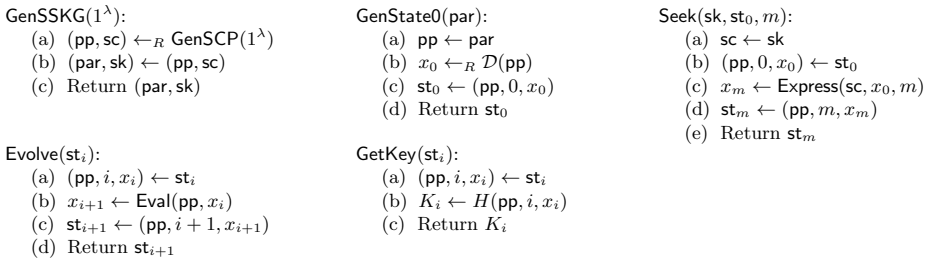

Fig. 4. SCP-based SSKG construction SCP-SSKG

6 Implementing SSKGs

Let FACT-SSKG denote the factorization-based SSKG obtained by combining Constructions 2 and 3. Some implementational details that increase the efficiency and versatility of this construction are discussed next.

We first propose a small tweak to the scheme that affects the storage size of the initial state. Recall that, in foreseen applications of SSKGs, the initial state st_0 is first created by (randomized) GenState0 algorithm and then copied to other parties (cf. discussion in Section 2.1). In FACT-SSKG, between 1024 to 4096 bits would have to be copied, depending on the desired level of security [1], just counting the size of $x_0 \in QR_N$. However, in the specific application we are aiming at, described in detail in Section 6.1, that much bandwidth is not available. We hence propose to make GenState0 algorithm deterministic, now providing it with an explicit random seed of short length (e.g., 80–128 bits); all randomness required by the original GenState0 algorithm is deterministically extracted from that seed via a PRG, and only 128 bits (or less) have to be shared with other parties. We implement this new feature by introducing an auxiliary algorithm, RndQR, that deterministically maps $\mathsf{seed} \in \{0,1\}^{\ell(\lambda)}$ to an element in QR_N such that the distribution of RndQR(N, seed) with random seed is negligibly close to the uniform distribution on QR_N. The new GenState0 and RndQR algorithms are shown in Figure 5. The admissibility of proposed RndQR construction is confirmed by [10] and [26, §B.5.1.3], in the random oracle model.

The second modification of FACT-SSKG improves the efficiency of the Seek operation. A standard trick [17,6] to speed up private operations in factoring-based schemes is via the Chinese Remainder Theorem (CRT). For instance, if an exponentiation $y \leftarrow x^k \bmod N$ is to be computed and the factorization $N = pq$ is known, then y can be obtained by CRT-decomposing x into $x_p \leftarrow x \bmod p$ and $x_q \leftarrow x \bmod q$, by computing $y_p \leftarrow x_p^{k \bmod \varphi(p)} \bmod p$ and $y_q \leftarrow x_q^{k \bmod \varphi(q)} \bmod q$ independently of each other, and by mapping (y_p, y_q) back to $\mathbb{Z}_N$ (by applying the CRT a second time). The described method to compute x^k is approximately four times faster than evaluating the term directly, without the CRT [25, Note 14.75]. The correspondingly modified Seek algorithm is shown in Figure 5.

We combine Remark 4, Theorem 1, and Lemma 1 to obtain:

Corollary 1 (Security of FACT-SSKG). *Under the assumption that integer factorization is hard, our seekable sequential key generator* FACT-SSKG *offers both* IND *and* IND-FS *security, in the random oracle model.*

6.1 Deployment in practice

We implemented FACT-SSKG (incorporating the tweaks described above) [27]. In fact, the code is part of the *journald* logging component of the *systemd* system and service manager, the core piece of many modern commercial Linux-based operating systems [31]. The SSKG is used as described in the introduction: it is combined with a cryptographic MAC in order to implement secured local logging, called *Forward-Secure Sealing* in journald. Generation of

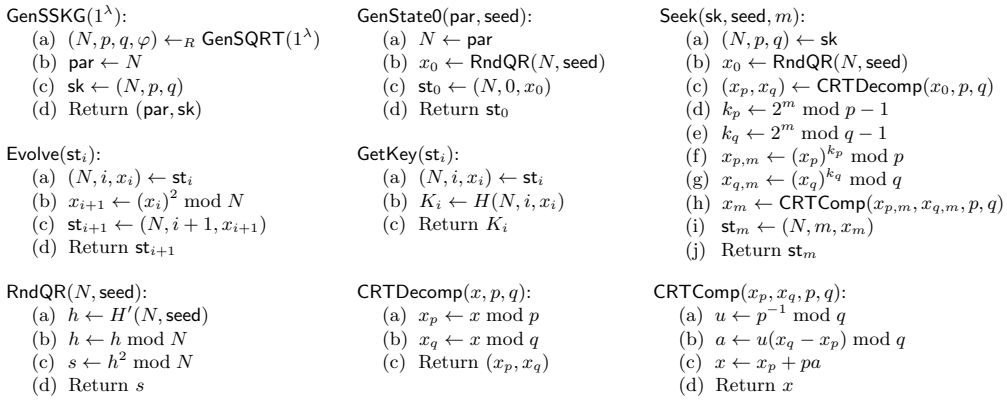

Fig. 5. Algorithms of optimized FACT-SSKG, together with auxiliary RndQR, CRTDecomp, and CRTComp algorithms. In the specification of RndQR we assume a hash function $H' : \{0,1\}^* \to \{0, \ldots, 2^t - 1\}$, where $t = \lceil \log_2 N \rceil + 128$.

initial state st_0 takes place on the system whose logs are to be protected. The corresponding seed is shown on screen only (hence the restriction on seed's size), both in text and as QR code [15]; the latter may be scanned off the screen with devices such as mobile phones. The separation between on-disk storage of public parameters and on-screen display of the seed is done in order to ensure the latter will not remain on the system. Each time the SKG state is evolved, a MAC tag protecting the data written since the previous MAC operation is appended to the log file. An offline verification tool that checks the MAC tag sequence of log files taken from a system is provided. If a log file is corrupted, the verification tool will determine the time range where the integrity of the log file is intact. When the SKG state is evolved, particular care is taken to ensure the previous state is securely deleted from the file system and underlying physical storage, which includes techniques to ensure secure removal even on modern copy-on-write file systems.

On the technical side, our implementation supports modulus sizes of 512–16384 bits (1536 bits is recommended), uses SHA256 for key derivation, and relies on HMAC-SHA256 for integrity protection. The code links against the *gcrypt* library [22] for large integer arithmetic and the SHA256 hash function, and is licensed under an Open Source license (LGPL 2.1).

Conclusion

We review different cryptographic schemes for log file protection and point out that they all lack an important usability feature: *seekability*. In short, seekability allows users of sequential key generators to jump to any position in the otherwise forward-secure keystream, in negligible time. We introduce a new primitive, *seekable sequential key generator* (SSKG), and give two provably-secure factorization-based constructions. As a side product, we introduce the concept of *shortcut one-way permutations* (SCP), which may find independent application.

Acknowledgements

The authors thank the anonymous reviewers of ESORICS 2013 for their valuable comments. Giorgia Azzurra Marson was supported by CASED and Bertram Poettering by EPSRC Leadership Fellowship EP/H005455/1.

References

1. S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT yearly report on algorithms and keysizes, September 2012. http://www.ecrypt.eu.org/documents/D.SPA.20.pdf.
2. M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Berlin, Germany.
3. M. Bellare and B. S. Yee. Forward-security in private-key cryptography. In M. Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 1–18, San Francisco, CA, USA, Apr. 13–17, 2003. Springer, Berlin, Germany.
4. L. Blum, M. Blum, and M. Shub. Comparison of two pseudo-random number generators. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *CRYPTO'82*, pages 61–78, Santa Barbara, CA, USA, 1983. Plenum Press, New York, USA.
5. L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986.
6. D. Boneh and H. Shacham. Fast variants of RSA. *RSA Cryptobytes*, 5(1):1–9, Winter/Spring 2002.
7. R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.
8. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Berlin, Germany.
9. C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *SEC*, volume 250 of *IFIP Conference Proceedings*, pages 73–84. Kluwer, 2003.
10. Y. Desmedt. Securing traceability of ciphertexts – towards a secure software key escrow system (extended abstract). In L. C. Guillou and J.-J. Quisquater, editors, *EUROCRYPT'95*, volume 921 of *LNCS*, pages 147–157, Saint-Malo, France, May 21–25, 1995. Springer, Berlin, Germany.
11. S. R. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In S. Vaudenay and A. M. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 1–24, Toronto, Ontario, Canada, Aug. 16–17, 2001. Springer, Berlin, Germany.
12. P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, 1996.
13. M. J. Hinek, M. K. Low, and E. Teske. On some attacks on multi-prime RSA. In K. Nyberg and H. M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 385–404, St. John's, Newfoundland, Canada, Aug. 15–16, 2003. Springer, Berlin, Germany.
14. J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo, editors, *ACSW Frontiers*, volume 54 of *CRPIT*, pages 203–211. Australian Computer Society, 2006.
15. International Organization for Standardization (ISO). Information Technology – Automatic identification and data capture techniques – QR Code 2005 bar code symbology specification. ISO/IEC 18004:2006, 2006.
16. G. Itkis. Forward security, adaptive cryptography: Time evolution, 2004.
17. J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.
18. J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.
19. J. Kelsey, J. Callas, and A. Clemm. Signed Syslog Messages. RFC 5848 (Proposed Standard), May 2010.
20. J. Kelsey and B. Schneier. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
21. J. Kelsey and B. Schneier. Minimizing bandwidth for remote access to cryptographically protected audit logs. In *Recent Advances in Intrusion Detection*, 1999.
22. W. Koch. GNU Privacy Guard – `gcrypt` library. http://www.gnupg.org/.
23. D. Ma and G. Tsudik. Extended abstract: Forward-secure sequential aggregate authentication. In *2007 IEEE Symposium on Security and Privacy*, pages 86–91, Oakland, California, USA, May 20–23, 2007. IEEE Computer Society Press.

24. D. Ma and G. Tsudik. A new approach to secure logging. *Trans. Storage*, 5(1):2:1–2:21, Mar. 2009.
25. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.
26. National Institute of Standards and Technology (NIST). Recommendation for random number generation using deterministic random bit generators (revised), March 2007. NIST Special Publication 800-90.
27. B. Poettering. `fsprg` – seekable forward-secure pseudorandom generator. `http://cgit.freedesktop.org/systemd/systemd/tree/src/journal/fsprg.c`.
28. B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, 1999.
29. V. Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999.
30. V. Stathopoulos, P. Kotzanikolaou, and E. Magkos. A framework for secure and verifiable logging in public communication networks. In J. López, editor, *CRITIS*, volume 4347 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006.
31. `systemd`. System and service manager. `http://www.freedesktop.org/wiki/Software/systemd/`.
32. B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS 2004*, San Diego, California, USA, Feb. 4–6, 2004. The Internet Society.
33. A. A. Yavuz and P. Ning. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *ACSAC*, pages 219–228. IEEE Computer Society, 2009.
34. A. A. Yavuz, P. Ning, and M. K. Reiter. BAF and FI-BAF: Efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems. *ACM Trans. Inf. Syst. Secur.*, 15(2):9, 2012.

A Proof of Lemma 1

Proof. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an arbitrary adversary against IND. Using $\mathcal{A}$, we construct an $\mathsf{IND\text{-}FS}$ adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ by letting $\mathcal{B}_1$ run $\mathcal{A}_1$ on the same input par and relay all occurring $\mathcal{O}_{\mathsf{Key}}$ queries to its own challenger. When $\mathcal{A}_1$ eventually stops and outputs $(state, n)$, $\mathcal{B}_1$ also stops and outputs $(state, n, n+1)$. Ignoring its st_m input, algorithm $\mathcal{B}_2$ runs $\mathcal{A}_2$ on $(state, K_n^b)$, again relays all $\mathcal{O}_{\mathsf{Key}}$ queries, and finally outputs $\mathcal{A}_2$'s guess b'. It is easy so see that

$$\Pr\left[\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}_{\mathsf{SKG},\mathcal{B}}(1^\lambda) = 1\right] = \Pr\left[\mathrm{Expt}^{\mathsf{IND},b}_{\mathsf{SKG},\mathcal{A}}(1^\lambda) = 1\right] \qquad \text{for } b \in \{0,1\} \ ,$$

as $\mathcal{B}_1$ prevents the '$m \le n$' condition in experiments $\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}$ to hold by returning $m = n + 1$. We hence have

$$\mathrm{Adv}^{\mathsf{IND}}_{\mathsf{SKG},\mathcal{A}}(\lambda) = \mathrm{Adv}^{\mathsf{IND\text{-}FS}}_{\mathsf{SKG},\mathcal{B}}(\lambda) \qquad \forall \lambda \ .$$

As we assume $\mathsf{IND\text{-}FS}$ security of SKG, this effectively bounds $\mathrm{Adv}^{\mathsf{IND}}_{\mathsf{SKG},\mathcal{A}}(\lambda)$ by a negligible function. This proves the statement. □

B Security model for stateful generators

For reference, we reproduce the experiment of Bellare and Yee's security definition for stateful generators [3]. We widely adapted their notation to the standard of our paper.

C RSA-based shortcut permutations

Definition 8 (RSA assumption). *For probabilistic algorithms* GenRSA *that take as input security parameter* 1^λ *and output triples* (N, φ, e) *such that* $\varphi = \varphi(N) = |\mathbb{Z}_N^\times|$ *and*

$\mathrm{Expt}^{\mathsf{IND\text{-}BY},b}_{\mathcal{A}}(1^\lambda)$:
(a) $\mathsf{KList} \leftarrow \emptyset$
(b) $\mathsf{par} \leftarrow_R \mathsf{GenSKG}(1^\lambda)$
(c) $\mathsf{st}_0 \leftarrow_R \mathsf{GenState0}(\mathsf{par})$
(d) $(state, m) \leftarrow_R \mathcal{A}_1^{\mathcal{O}_{\mathsf{Key}}}(\mathsf{par})$
– If $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Key}}(i)$:
(a) $\mathsf{KList} \leftarrow \mathsf{KList} \cup \{i\}$
(b) $K_i^0 \leftarrow_R \{0,1\}^{\ell(\lambda)}$
(c) $K_i^1 \leftarrow \mathsf{GetKey}^t(\mathsf{st}_0)$
(d) Answer $\mathcal{A}$ with K_i^b
(e) $\mathsf{st}_m \leftarrow \mathsf{Evolve}^m(\mathsf{st}_0)$
(f) $b' \leftarrow_R \mathcal{A}_2^{\mathcal{O}_{\mathsf{Key}}}(state, \mathsf{st}_m)$
– Answer $\mathcal{O}_{\mathsf{Key}}$ queries as above
(g) Return 0 if $m \leq \max(\mathsf{KList})$
(h) Return b'

Fig. 6. Security experiment for stateful generators (adapted syntax)

$\gcd(e, \varphi) = 1$, *the* RSA problem *is said to be* hard *if for all efficient adversaries* $\mathcal{A}$ *the success probability*

$$\Pr\left[(N, \varphi, e) \leftarrow_R \mathsf{GenRSA}(1^\lambda); y \leftarrow_R \mathbb{Z}_N; x \leftarrow_R \mathcal{A}(N, e, y) : x^e \equiv y \bmod N\right]$$

is a negligible function, where the probability is taken over the random coins of the experiment (including over $\mathcal{A}$*'s randomness). The* RSA assumption *states that there exists an efficient probabilistic algorithm* GenRSA *for which the* RSA *problem is hard.*

Construction 4 (RSA-SCP) *Construct* RSA*-based SCP as follows: Let* $\mathsf{GenSCP}(1^\lambda)$ *run* $\mathsf{GenRSA}(1^\lambda)$ *and output* $\mathsf{pp} = (N, e)$ *and* $\mathsf{sc} = \varphi$*, let* $\mathcal{D} = \mathbb{Z}_N$*, let* $\mathsf{Eval}(N, x)$ *output* $x^e \bmod N$*, and let* $\mathsf{Express}(\varphi, x, m)$ *output* $x^{(e^m \bmod \varphi)} \bmod N$*.*

Remark 5 (Correctness and security of RSA-SCP*).* Observe that specified domain $\mathcal{D}$ is efficiently samplable, that correctness of the SCP follows from standard number-theoretic results [25], and that every Express operation takes about one exponentiation modulo N. Further, comparing the experiments in Definitions 6 and 8 makes evident that RSA-SCP is one-way if the RSA problem is hard for GenRSA.

D Proof of Theorem 1

Proof. In brief, we want to make formal the intuition that an attacker $\mathcal{A}$ could tell apart a real key K_n from a random one only by querying the oracle H on 'the right value' (n, x_n)[8]: indeed, a proper key K_n is computed by hashing the corresponding state $\mathsf{st}_n = (n, x_n)$. Querying H on state st_n reveals immediately the nature, real or random, of the challenge key K_n^b. However, we will prove that the probability that an attacker does pose such a query to H is negligible, hence the indistinguishability.

By using game hops, we modify the original experiment, $\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}$, into a game that $\mathcal{A}$ can 'win' only by guessing (she has advantage zero). In such a game, both the challenge keys K_n^0 and K_n^1 are randomly chosen, so that no adversary can find a better strategy than guessing b.

We now formalize this intuition. In what follows, for better legibility we do not explicitly consider the public parameter pp generated by the IND-FS challenger, but we instead refer to the permutation $\pi : \mathcal{D} \to \mathcal{D}$ that it defines. Let $\mathsf{st}_0 = (0, x_0)$ denote the initial state the challenger generates. Further states $\mathsf{st}_t = (t, x_t)$ are computed by iterating the shortcut

[8] in the proof we drop the first component par for legibility

permutation, $x_t \leftarrow \pi(x_{t-1})$, while keys are the hash value of these states, $K_t \leftarrow H(t, x_t)$. The attacker is granted oracle access to $\mathcal{O}_{\mathsf{Key}}$ and H.

There are three games, starting from $\mathsf{Game}_0^b = \mathrm{Expt}^{\mathsf{IND\text{-}FS},b}$. In the hop to Game_1^b, we modify the game in such a way that, if $\mathcal{A}$ poses query (n, x_n) to H which corresponds to st_n, i.e. $x_n = \pi^n(x_0)$, then the game aborts. In other words, Game_1^b forbids $\mathcal{A}$ to compute the key K_n (and, by comparing with the challenge key K_n^b, to trivially win the game). Going on, in Game_2^b, we let both potential challenge keys, K_n^0 and K_n^1, be randomly chosen. The latter game is indeed trivial to analyze: Game_2^0 and Game_2^1 are identical!

Let E be the event '$\mathcal{A}$ requests query (n, x_n) to H'. By definition, Game_0^b and Game_1^b are exactly the same as long as E does not happen. As the transition is based on a 'failure event', we can bound the advantage of an attacker $\mathcal{A}$ against the IND-FS experiment as follows:

$$\mathrm{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}FS}}(\lambda) \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{Game}_1^b}(\lambda) + \Pr[\mathsf{E}] \quad .$$

As we later prove $\Pr[\mathsf{E}]$ is negligible, hence Game_0^b and Game_1^b are indistinguishable.

The second transition is fictitious: choosing both challenge keys K_n^0 and K_n^1 uniformly at random, or having the one output by H and the other randomly chosen, is essentially the same, as long as we prevent $\mathcal{A}$ to query H on the input which yield K_n^1. Since, in Game_1^b, we have already excluded such an eventuality (if E happens, then the game aborts), there is no way an attacker can tell whether she is playing against Game_1^b or Game_2^b. Hence, $\mathcal{A}$'s advantage in Game_1^b equals her advantage in Game_2^b. As already observed, the last game is trivial to analyze: any adversary has advantage zero. We end up with the following bound:

$$\mathrm{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}FS}}(\lambda) \leq \Pr[\mathsf{E}] \quad . \tag{1}$$

We now argue that the event E occurs only with negligible probability. Actually, we can prove a stronger statement, namely that *any* query (t, x_t) that $\mathcal{A}$ poses to H, with $x_t = \pi^t(x_0)$ and $t < m$, has only negligible probability to occur. If this holds, the event E has also negligible probability: as n must be smaller than m, excluding *every* query (t, x_t) with $t < m$ automatically excludes, in particular, query (n, x_n). In what follows, we shortly call this kind of queries *lucky*. We prove that E is unlikely to occur by building an inverter $\mathcal{B}$ for the SCP π, which uses $\mathcal{A}$ as a black box, and succeeds whenever $\mathcal{A}$ makes lucky queries to H.

Lemma 2. *Let $\mathcal{A}$ be an* IND-FS *attacker against* SCP-SSKG*, π be the SCP underlying the scheme, and M be an upper bound for the number of iterations the scheme supports. If, within the experiment, $\mathcal{A}$ queries the hash oracle H on values (t, x_t) such that $x_t = \pi^t(x_0)$ and $t < m$ with probability ϵ, then there exists an inverter $\mathcal{B}$ for π, which uses $\mathcal{A}$ as a black box, whose advantage is at least*

$$\mathrm{Adv}_{\pi,\mathcal{B}}^{\mathsf{invert}}(\lambda) \geq \frac{\epsilon}{M} \quad . \tag{2}$$

Putting together the inequalities (1) and (2), we can bound the probability of the event E, that is, the advantage of $\mathcal{A}$ in the IND-FS experiment:

$$\mathrm{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}FS}}(\lambda) \leq M \cdot \mathrm{Adv}_{\pi,\mathcal{B}}^{\mathsf{invert}}(\lambda) \quad .$$

This term is negligible in λ as π is *one way*.

It remains to prove Lemma 2.

Proof (Lemma 2). We want to build an inverter $\mathcal{B}$ for SCP whose strategy relies on $\mathcal{A}$'s ability of making lucky queries.

The main idea is to embed $\mathcal{B}$'s challenge y into the state st_m that $\mathcal{A}$ requests, and wait until she asks a lucky query $q_t = (t, x_t)$: as $t \leq n < m$, it holds that $\pi^{m-t}(x_t) = y$. The inverter outputs $\pi^{m-t-1}(x_t)$, winning the inverting experiment. However, we have to

make sure that $\mathcal{B}$ simulates a suitable environment for $\mathcal{A}$ and answers her queries to $\mathcal{O}_{\mathsf{Key}}$ and H. More precisely, $\mathcal{B}$ has to guarantee that the two oracles are *consistent*, i.e., that $\mathcal{O}_{\mathsf{Key}}(t) = H(q_t)$ for every query q_t which corresponds to a valid state st_t.

Within the proof, we can assume wlog. that the adversary $\mathcal{A}$ makes only 'clever' queries to H, thus trying to hit lucky queries by guessing pairs (t, z). We call a query (t, z) to H *in-line* (with the initial state $(0, x_0)$), and we denote it by (t, x_t), if $z = \pi^t(x_0)$. Recall that, according to the previous definition, a query to H is lucky if it is in-line and $t < m$. In this terms, consistency between the two oracles H and $\mathcal{O}_{\mathsf{Key}}$ means that, for every in-line query (t, x_t), H's reply must equal $\mathcal{O}_{\mathsf{Key}}(t)$.

Now, observe that $\mathcal{B}$ cannot check whether $\mathcal{A}$'s queries are in-line or not, nor whether $\mathcal{A}$ 'accidentally' hits any valid state, until $\mathcal{A}$ declares the value m. More precisely, in-line queries are *not* even well-defined before that: in $\mathcal{B}$'s simulation of the experiment, there is no explicit initial state $(0, x_0)$. From $\mathcal{B}$'s point of view, a value x_0 *does* exist, being implicitly defined by the equality $y = \pi^m(x_0)$, but this becomes true only when $\mathcal{A}$ outputs m. In this sense, in-line queries are well-defined only *after* $\mathcal{A}_1$ has declared which state st_m she wants to corrupt. This may undermine the 'quality' of $\mathcal{B}$'s simulation.

However, we overcome the issue above by letting the inverter guess m, and 'define' in-line queries with respect to its guess and the challenge value y it receives. We now describe $\mathcal{B}$'s strategy in more detail. First, $\mathcal{B}$ chooses $m^\star$ uniformly at random from $\{1, \ldots, M\}$ with the hope to guess the right value m. $\mathcal{B}$ simulates the oracle $\mathcal{O}_{\mathsf{Key}}$ by returning random $\ell(\lambda)$-bit strings, with the constraint that, if a query is asked twice, the answer must be the same. In case $\mathcal{A}$ requests a key K_t for $t \geq m^\star$, the inverter must also assure consistency with the behavior of H on queries q_t that are in-line (but *not* lucky). This can be done by programming the oracle H. Queries to H are more delicate since could help $\mathcal{B}$ to invert π. If a query (t, z) is not in-line, then $\mathcal{B}$ simply replies with a random string, provided that $H(t, z)$ has not been already set (which happens, e.g., if the same query was previously asked or the oracle has been programmed on (t, z)). Regarding in-line queries q_t, there are two cases: if $t < m^\star$, then $\mathcal{B}$ extracts a (potential) preimage of y and interrupts the simulation, otherwise replies with the corresponding key K_t (in case it has been already defined). A more detailed description of how $\mathcal{B}$ simulates the oracles follows. Let $\mathcal{B}$ maintain tables KList and HList, for storing queries and corresponding answers to $\mathcal{O}_{\mathsf{Key}}$ and H respectively.

Queries t to $\mathcal{O}_{\mathsf{Key}}$. Check whether KList already contains an entry (t, K_t) and, if not:
- If $t \geq m^\star$, pick K_t uniformly at random, compute $z_t = \pi^{t-m^\star}(y)$ and *program* the oracle by setting $H(t, z_t) := K_t$. Update KList and HList by adding entries (t, K_t) and $((t, z_t), K_t)$ respectively.
- If $t < m^\star$, choose a random key K_t and add the pair (t, K_t) to KList.

In either case, return K_t.

Queries $q = (t, z)$ to H. Go through HList and, if there is no entry $((t, z), h)$:
- If $t \geq m^\star$, check whether $\pi^{t-m^\star}(y) = z$: if not, choose $h \in \{0,1\}^{\ell(\lambda)}$ uniformly at random, set $H(q) := h$, update HList by appending (q, h). Otherwise, there are two cases:
  (a) If there is an entry $(t, K_t) \in \mathsf{KList}$, *program* the oracle $H(q) := K_t$ and add the pair (q, K_t) to HList.
  (b) Otherwise, pick a random $\ell(\lambda)$-bit string h, set $H(q) := h$, add entries (q, h) and (t, h) to HList and KList respectively.
- If $t < m^\star$, check whether $\pi^{m^\star - t}(z) = y$: if so, set $x := \pi^{m^\star - t - 1}(z)$, interrupt the simulation and output x.

If none of the previous case arises, pick a $\ell(\lambda)$-bit string h uniformly at random, set $H(q) := h$, and update HList by appending (q, h). Eventually, return $H(q)$.

The algorithm $\mathcal{B}$ maintains the simulation until *either* it computes a (potential) preimage of y, *or* $\mathcal{A}_1$ declares values n and m: at this point $\mathcal{B}$ can check whether its guess on m was correct and, if not, it stops and outputs $\mathsf{failure}$. Otherwise, if it guessed $m^\star = m$ correctly, it *does* know the valid state $\mathcal{A}_1$ has requested, as $x_m = y$. The inverter can, thus, continue

the simulation and, from now on, it is sure that any lucky query sent by $\mathcal{A}_2$ will permit to invert y.

Eventually $\mathcal{B}$ gives $\mathsf{st}_m = (m, y)$, together with a random key K_n, to $\mathcal{A}_2$, and proceeds by answering her queries as previously done. As before, if a lucky query q_t to H is made, $\mathcal{B}$ does not need to answer: it computes and outputs a preimage of y and terminates. If there is no lucky query, $\mathcal{B}$ keeps running the simulation until $\mathcal{A}_2$ returns b', thereafter outputs $\mathsf{failure}$ and halts.

We now analyze $\mathcal{B}$'s winning probability. Observe that, in order to win, the inverter has to guess the right value m and, if it does, its strategy succeeds whenever $\mathcal{A}$ requests a lucky query to H. Denote by F the event '$\mathcal{A}$ asks a lucky query to H'. It holds

$$\Pr[\mathcal{B}(\mathsf{par}, y) = x \ : \ \pi(x) = y] = \Pr[m^\star = m \ \wedge \ \mathsf{F}]$$

and, since the two events on the right side are independent, $\Pr[\mathsf{F}] = \epsilon$ and the probability of hitting the right value m by randomly guessing $m^\star$ is $1/M$, we obtain the claimed bound. □