Even More Practical Secure Logging: Tree-Based 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. Sequential key generators produce a forward-secure sequence of symmetric cryptographic keys and are traditionally based on hash chains. An inherent disadvantage of such constructions is that they do not offer a fast-forward capability, i.e., lack a way to efficiently skip a large number of keys—a functionality often required in practice. This limitation was overcome only recently, with the introduction of *seekable sequential key generators* (SSKGs). The only currently known construction is based on the iterated evaluation of a *shortcut one-way permutation*, a factoring-based —and hence in practice not too efficient— building block. In this paper we revisit the challenge of marrying forward-secure key generation with seekability and show that symmetric primitives like PRGs, block ciphers, and hash functions suffice for obtaining secure SSKGs. Our scheme is not only considerably more efficient than the prior number-theoretic construction, but also extends the seeking functionality in a way that we believe is important in practice. Our construction is provably (forward-)secure in the standard model.

Keywords: secured logging, forward security, seekable PRGs.

1 Introduction

Computer log files can be configured to record a large variety of system events that occur on network hosts and communication systems, including users logging on or off, memory resources reaching their capacity, malfunctioning of disk drives, etc. Therefore, log files represent one of the most essential sources of information that support system administrators in understanding the activity of systems and keeping them fully functional. Not less important is the role that log files play in computer forensics: events like login failures and software crashes serve as standard indicators for (attempted) intrusions. Unfortunately, as log files are often recorded locally (i.e., on the monitored machine itself), in many practical cases intruders can a posteriori manipulate the log entries related to their attacks.

Online logging and its disadvantages. In a network environment, one obvious strategy to prevent adversarial tampering of audit logs is to forward log messages immediately after their creation to a remote log sink—in the hope that the attacker cannot also corrupt the latter. Necessary in such a setting is that the

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part II, LNCS 8713, pp. 37–54, 2014.

log sink is continuously available, as every otherwise required local buffering of log records would increase the risk that their delivery is suppressed by the adversary. However, in many cases it has to be assumed that the reachability of the log sink can be artificially restrained by the intruder, e.g., by confusing routing protocols with false ARP messages, by sabotaging TCP connections with injected reset packets, by jamming wireless connections, or by directing application-level denial-of-service attacks against the log sink. Independently of these issues, it is inherently difficult to choose an appropriate logging granularity: while the creation of individual records for each established TCP connection, file deletion, or subprocess invocation might be desirable from the point of view of computer forensics, network links and log sinks might quickly reach their capacities if events are routinely reported with such a high resolution. This holds in particular if log sinks serve multiple monitored hosts simultaneously.

Forward-secure cryptography & log file protection. A solution for tamper-resistant log-entry storage that does not require a remote log sink but offers integrity protection via cryptographic means is *secured local logging.* Here, each log entry is stored together with a specific authentication tag that is generated and verified using a secret key. Note that regular message authentication codes (MACs) by themselves seem not to constitute a secure solution: corresponding tags will be forgeable by intruders that succeed in extracting the secret key from the attacked device. Rather, a forward-secure MAC variant is required, as elaborated next.

In a nutshell, a cryptosystem provides *forward security* (FS) if it continues to give meaningful security guarantees after the adversary got a copy of the used keys. A standard example is key exchange: here, all recent security models require established session keys to remain safe when the adversary obtains access to the involved long-term private keys. Likely less known is that the notion of forward security also extends to non-interactive primitives. For instance, in forward-secure public key encryption [1] messages are encrypted in respect to a combination (pk, t), where pk is a public key and $t \in \mathbb{N}$ identifies one out of a set of consecutive time epochs; for each such epoch t, knowledge of a specific decryption key sk_t is necessary for decrypting corresponding ciphertexts. In addition, while by design it is efficiently possible to perform updates $sk_t \mapsto sk_{t+1}$, forward security requires that the reverse mapping be inefficient, i.e., it shall be infeasible to 'go backwards in time'. More precisely, forward security guarantees that plaintexts encrypted for 'expired' epochs remain confidential even if the decryption keys of all later epochs are revealed.

Analogously to the described setting, signatures and authentication tags of the forward-secure variants of signature schemes and MACs, respectively, remain unforgeable for past epochs if only current and future keys are disclosed to the adversary [2,3]. One possible way to obtain such a MAC is to combine a (forward-secure) *sequential key generator* (SKG) with a regular MAC [4,3], where the former can be seen as a stateful *pseudorandom generator* (PRG) that, once initialized with a random seed, deterministically outputs a pseudorandom sequence of fixed-length keys. These keys are then used together with a MAC to ensure unforgeability of messages within the epochs.

The challenge of seekability. Forward-secure SKGs are typically constructed by deterministically evolving an initially random state using a hash chain, i.e., by regularly replacing a 'current' key K_t by $K_{t+1} = H(K_t)$, where H is a cryptographic hash function [4,3]. Although hash chains, in principle, lead to (forward-)secure local logging, they also come with an efficiency penalty on the side of the log auditor: the latter, in order to verify a log record of a certain epoch t, first needs to recover the corresponding key K_t; however, as a high level of security requires a high key update rate, this might involve millions of hash function evaluations. This problem was addressed only recently with the introduction of *seekable sequential key generators* (SSKGs) [5].

We give a rough overview over the ideas in [5]. Essentially, the authors propose a generic construction of an SSKG from a *shortcut one-way permutation* (SCP), a primitive that implements a one-way permutation $\pi\colon D \to D$, for a domain D, with a dedicated shortcut algorithm allowing the computation of the k-fold composition π^k in sublinear time. The concrete SCP considered in [5] is given by the squaring operation modulo a Blum integer N, where applying the shortcut corresponds to reducing a certain exponent modulo $\varphi(N)$. Given an SCP, an SSKG can be obtained by letting its state consist of a single element in D, performing state updates by applying π to this element, and deriving keys by hashing it (more precisely, by applying a random oracle). While it is instructive to observe how the forward security of the SSKG corresponds with the one-wayness of the SCP, and how its seekability is based on the SCP's shortcut property, a notable technical artifact of the squaring-based SCP is that seekability requires knowledge of $\varphi(N)$ while forward security requires this value to be unknown. This dilemma is side-stepped in [5] by giving only the owners of a *seeking key* the ability to fast-forward through the SSKG output sequence.

1.1 Contributions and Organization

The central contribution of this paper is the design of a new seekable sequential key generator. In contrast to the prior SSKG from [5], our scheme relies on just symmetric building blocks; in particular we propose instantiations that exclusively use either PRGs, block ciphers, or hash functions. By consequence, our implementation beats the one from [5] by 1–3 orders of magnitude, on current CPUs. In addition to this efficiency gain, we also identify new and appealing functionality features of our SSKG. In particular, getting rid of the discussed seeking limitations of [5], our scheme allows *every* user to efficiently advance any state by an arbitrary number of epochs. Our SSKG is supported by a security proof in the standard model.

This paper is organized as follows. After starting with preliminaries in Section 2, we formally specify the functionality, syntax, and security requirements of SSKGs in Section 3; this includes a comparison with the (different) formalizations in [5]. In Section 4 we describe our new PRG-based SSKG, including its generalized seekability notion and some possible time-memory trade-offs. Finally, in Section 5, we discuss implementational aspects and efficiency results from our implementation.

1.2 Related Work

The first published work that highlights the importance of *seekability* as a desirable property of sequential key generators in the context of secured local logging is [5,6]. An extensive comparison of the corresponding results with the ones of the current paper can be found in the preceding paragraphs and in Section 3. In the following we discuss further publications on sequential key generation and cryptographic audit log protection. We observe that all considered protocols either are forward-secure or offer seekability, but not both simultaneously.

An early approach towards secured local logging originates from Bellare and Yee [7]; they study the role of forward security in authentication, develop the security notion of *forward integrity*, and realize a corresponding primitive via a PRF chain. Later, the same authors provide the first systematic analysis of forward security in the symmetric setting [3], covering forward-secure variants of pseudorandom generators, symmetric encryption, and MACs, and also providing constructions and formal proofs of security for these primitives.

Shortly after [7], an independent cryptographic scheme specifically targeted at protecting log files was described by Kelsey and Schneier [4,8,9]. Their scheme draws its (forward) security from frequent key updates via iterated hashing, but is not supported by a formal security analysis. A couple of implementations exist, notably the one by Chong, Peng, and Hartel in tamper-resistant hardware [10] and the *logcrypt* system by Holt [11]. The latter improves on [4] by paving the way towards provable security, but also adds new functionality and concepts. Most notable is the suggestion to embed regular metronome entries into log files to thwart *truncation attacks* where the adversary cuts off the most recent set of log entries. Similar work is due to Accorsi [12] who presents *BBox*, a hash-chain-based framework for protecting the integrity and confidentiality of log files in distributed systems.

Ma and Tsudik consider the concept of *forward-secure sequential aggregate authentication* for protecting the integrity of system logs [13,14]. Their constructions build on compact constant-size authenticators with all-or-nothing security (i.e., adversarial deletion of *any* single log message is detected), naturally defend against truncation attacks, and enjoy provable security.

The proposals by Yavuz and Ning [15], and Yavuz, Ning, and Reiter [16], specifically aim at secured logging on constraint devices and support a shift of computation workload from the monitored host to the log auditor. Notably, their key update procedure and the computation of authentication tags takes only a few hash function evaluations and finite field multiplications. In common with the schemes discussed above, their authentication systems are not seekable.

Kelsey, Callas, and Clemm [17] introduced secured logging into the standardization process at IETF. However, their proposal of *signed syslog messages* focuses on remote logging instead of on local logging. Precisely, their extension to the standard UNIX *syslog* facility authenticates log entries via signatures before sending them to a log sink over the network. While this proposal naturally offers seekability, it is bound to the full-time availability of an online log sink.

Indeed, periods where the latter is not reachable are not securely covered, as the scheme is not forward-secure.

2 Preliminaries

We recall basic notions and facts from cryptography, graph theory, and data structures that we require in the course of this paper. Notably, in the section on trees, we define what we understand by the 'co-path' of a node. If not explicitly specified differently, all logarithms are understood to be taken to base 2.

2.1 Pseudorandom Generators

A *pseudorandom generator* (PRG) is a function that maps a random string ('*seed*') to a longer 'random-looking' string. The security property of *pseudorandomness* requires that it be infeasible to distinguish the output of a PRG from random.

Definition 1 (Pseudorandom generator). *For security parameter λ and a polynomial $c\colon \mathbb{N} \to \mathbb{N}^{\geq 1}$, an efficiently computable function $G\colon \{0,1\}^\lambda \to \{0,1\}^{\lambda+c(\lambda)}$ is a* pseudorandom generator *if for all efficient distinguishers $\mathcal{D}$ the following advantage function is negligible, where the probabilities are taken over the random choices of s and y, and over $\mathcal{D}$'s randomness:*

$$\mathrm{Adv}^{\mathsf{PRG}}_{G,\mathcal{D}}(\lambda) = \left|\Pr\left[\mathcal{D}(G(s)) = 1 : s \xleftarrow{\$} \{0,1\}^\lambda\right] - \Pr\left[\mathcal{D}(y) = 1 : y \xleftarrow{\$} \{0,1\}^{\lambda+c(\lambda)}\right]\right| \enspace .$$

2.2 Binary Trees

A *tree* is a simple, undirected, connected graph without cycles. We particularly consider rooted trees, i.e., trees with a distinguished *root* node. The nodes adjacent to the root node are called its *children*; each child can be considered, in turn, the root of a subtree. The *level* L of a node indicates its distance to the root, where we assign level $L = 1$ to the latter. Children of the same node are *siblings* of each other. In this paper we exclusively consider binary trees of constant height H. These are trees in which every node has exactly one sibling, with exception of the root which has no sibling, and where all leaves have the same level $L = H$; such trees have a total of $N = 2^H - 1$ nodes. We assume that the children of each node are ordered; we refer to them as 'left' and 'right'. Nodes that have no children are called *leaves*, all other nodes are called *internal.*

We finally define the notion of *co-path* of a node. Let v denote an arbitrary node of a binary tree. Intuitively speaking, the (right) co-path of v is the list of the right siblings of the nodes on the (unique) path connecting the root node with v. For a formal definition, let L denote the level of $v = v_L$ and let $(v_1, \ldots, v_L)$ denote the path that connects the root (denoted here with v_1) with v_L. For each $1 \leq i \leq L$ let $V_i^{\to}$ be the list of right siblings of node v_i (these lists contain at most one element, and particularly $V_1^{\to}$ is always empty). We define the co-path of v_L to be the list $V_L^{\to} \parallel \ldots \parallel V_1^{\to}$ obtained by combining these lists into a single one using concatenation.

2.3 Stacks and Their Operations

A *stack* is a standard data structure for the storage of objects. Stacks follow the last-in first-out principle: the last element stored in a stack is the first element to be read back (and removed). The following procedures can be used to operate on stacks for storing, reading, and deleting elements. By $\mathbf{Init}(\mathcal{S})$ we denote the initialization of a fresh and empty stack $\mathcal{S}$. To add an element x 'on top of' stack $\mathcal{S}$, operation $\mathbf{Push}(\mathcal{S}, x)$ is used. We write $x \leftarrow \mathbf{Pop}(\mathcal{S})$ for reading and removing the top element of stack $\mathcal{S}$. Finally, with $x \leftarrow \mathbf{Peek}_k(\mathcal{S})$ the k-th element of stack $\mathcal{S}$ can be read without deleting it; here, elements are counted from the top, i.e., $\mathbf{Peek}_1(\mathcal{S})$ reads the top-most element. When using these notations, operations **Init**, **Push**, and **Pop** are understood to modify their argument $\mathcal{S}$ in place, while $\mathbf{Peek}_k$ leaves it unchanged.

3 Seekable Sequential Key Generators

The main contribution of this paper is a new construction of a *seekable sequential key generator* (SSKG). This cryptographic primitive can be seen as a stateful PRG that outputs a sequence of fixed-length keys—one per invocation. The specific property of *seekability* ensures that it is possible to jump directly to any position in the output sequence. At the same time, the security goal of forward security ensures that keys remain indistinguishable from random even upon corruption of the primitive's state. We next recall the syntactical definition and security properties, (mainly) following the notation from [5]. We defer the exposition of our new scheme to Section 4.

3.1 Functionality and Syntax

Generally speaking, a seekable sequential key generator consists of four algorithms: $\mathsf{GenSSKG}$ generates an initial state st_0, the update procedure Evolve maps each state st_i to its successor state st_{i+1}, GetKey derives from any state st_i a corresponding (symmetric) key K_i, and Seek permits to compute any state st_i directly from initial state st_0 and index i. We consider each state associated with a specific period of time, called *epoch*, where the switch from epoch to epoch is carried out precisely with the Evolve algorithm. This setting is illustrated in Figure 1 and formalized in Definition 2.

Definition 2 (Syntax of SSKG). *Let* $\ell\colon \mathbb{N} \to \mathbb{N}$ *be a polynomial. A* seekable sequential key generator *with key length* ℓ *is a tuple* $\mathsf{SSKG} = \{\mathsf{GenSSKG}, \mathsf{Evolve}, \mathsf{GetKey}, \mathsf{Seek}\}$ *of efficient algorithms as follows:*

- $\mathsf{GenSSKG}$. *On input of security parameter* 1^λ *and total number* $N \in \mathbb{N}$ *of supported epochs, this probabilistic algorithm outputs an initial state* st_0.
- Evolve. *On input of a state* st_i*, this deterministic algorithm outputs the 'next' state* st_{i+1}*. For convenience, for* $k \in \mathbb{N}$*, by* Evolve^k *we denote the* k*-fold composition of* Evolve*, i.e.,* $\mathsf{Evolve}^k(\mathsf{st}_i) = \mathsf{st}_{i+k}$.

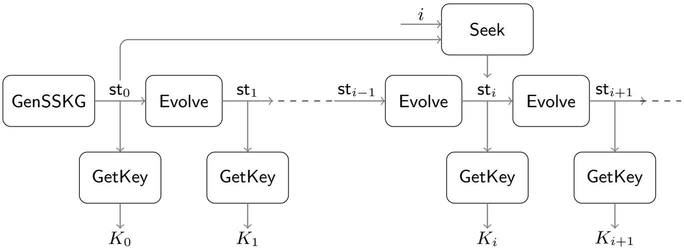

Fig. 1. Illustration of the interplay of the different SSKG algorithms

- GetKey. *On input of state* st_i*, this deterministic algorithm outputs a key* $K_i \in \{0,1\}^{\ell(\lambda)}$*. For* $k \in \mathbb{N}$*, we write* $\mathsf{GetKey}^k(\mathsf{st}_i)$ *for* $\mathsf{GetKey}(\mathsf{Evolve}^k(\mathsf{st}_i))$.
- Seek. *On input of initial state* st_0 *and* $k \in \mathbb{N}$*, this deterministic algorithm returns state* st_k.

Implicit in Definition 2 is the following natural consistency requirement on the interplay of Evolve and Seek algorithms:

Definition 3 (Correctness of SSKG). *A seekable sequential key generator* SSKG *is* correct *if, for all security parameters* λ, $N \in \mathbb{N}$, $\mathsf{st}_0 \xleftarrow{\$} \mathsf{GenSSKG}(1^\lambda, N)$, *and all* $k \in \mathbb{N}$ *we have*

$$0 \le k < N \qquad \Longrightarrow \qquad \mathsf{Evolve}^k(\mathsf{st}_0) = \mathsf{Seek}(\mathsf{st}_0, k) \ .$$

Remark 1 (Comparison with the definition from [5]). The syntax specified in Definition 2 does slightly deviate from the one in [5, Definition 3]: firstly, the SSKG setup routine of [5] has a secret ‘seeking key’ as additional output; it is required as auxiliary input for the Seek algorithm. The necessity of this extra key should be considered an artifact of the number-theory-based construction from [5] (see Section 3.4 for details): the seeking key contains the factorization of the RSA modulus underlying the scheme. As the proposed Evolve algorithm is one-way only if this factorization is not known, the Seek algorithm is available exclusively to those who know the seeking key as a ‘trapdoor’. In contrast to that, our syntax for Seek is not only more natural, we also allow *everybody* to use the Seek algorithm to fast-forward efficiently to future epochs. Secondly, in [5] the number of supported epochs does not have to be specified at the time of SSKG initialization; instead, an infinite number of epochs is supported by every instance. We had to introduce this restriction for technical reasons that become clear in Section 4; however, we believe that the requirement of specifying the number of epochs in advance does not constrain the practical usability of our scheme too much: indeed, regarding our scheme from Section 4, instantiations with, say, $N = 2^{30}$ supported epochs are perfectly practical.

3.2 Security Requirements

As the security property of SSKGs we demand indistinguishability of generated keys from random strings of the same length. This is modeled in [5] via an experiment involving an adversary $\mathcal{A}$ who first gets adaptive access to a set of (real) keys K_i of her choosing, and is then challenged with a string K_n^b that is either the real key K_n or a random string of the same length; the adversary has to distinguish these two cases. This shall model the intuition that keys K_n 'look random' even if the adversary is given (all) other keys K_i, for $i \neq n$. Below we formalize a stronger security notion that also incorporates forward security, i.e., additionally lets the adversary corrupt any state that comes after the challenged epoch.

Definition 4 (IND-FS security of SSKG [5]). *A seekable sequential key generator* SSKG *is* indistinguishable with forward security against adaptive adversaries (IND-FS) *if, for all efficient adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *that interact in experiments* $\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}$ *from Figure 2 and all* $N \in \mathbb{N}$ *bounded by a polynomial in* λ*, the following advantage function is negligible, where the probabilities are taken over the random coins of the experiment (including over* $\mathcal{A}$*'s randomness):*

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

```
$\mathrm{Expt}^{\mathsf{IND\text{-}FS},b}_{\mathsf{SSKG},N,\mathcal{A}}(1^\lambda)$:
  $\mathsf{KList} \leftarrow \emptyset$
  $\mathsf{st}_0 \xleftarrow{\$} \mathsf{GenSSKG}(1^\lambda, N)$
  $(state, n, m) \xleftarrow{\$} \mathcal{A}_1^{\mathcal{O}_{\mathsf{Key}}}(1^\lambda, N)$
  Abort if not $0 \leq n < m < N$
  $K_n^0 \xleftarrow{\$} \{0,1\}^{\ell(\lambda)}$
  $K_n^1 \leftarrow \mathsf{GetKey}^n(\mathsf{st}_0)$
  $\mathsf{st}_m \leftarrow \mathsf{Evolve}^m(\mathsf{st}_0)$
  $b' \xleftarrow{\$} \mathcal{A}_2^{\mathcal{O}_{\mathsf{Key}}}(state, \mathsf{st}_m, K_n^b)$
  Abort if $n \in \mathsf{KList}$
  Return $b'$
```

```
If $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Key}}(i)$:
  Abort if not $0 \leq i < N$
  $\mathsf{KList} \leftarrow \mathsf{KList} \cup \{i\}$
  $K_i \leftarrow \mathsf{GetKey}^i(\mathsf{st}_0)$
  Answer $\mathcal{A}$ with $K_i$
```

Fig. 2. Security experiments for indistinguishability with forward security. The abort operation lets the experiment return 0, disregarding any output of the adversary.

3.3 An Application: Protecting Locally Stored Log Files

Given the definitions from Sections 3.1 and 3.2, the role of SSKGs in the context of secure logging is now immediate: in every epoch i, corresponding key K_i is used to instantiate a message authentication code (MAC) that equips all occurring log messages with an authentication tag. In addition, the Evolve algorithm is regularly invoked to advance from one epoch to the next, burying for all times the previously used keys. In such a setting, an auxiliary copy of initial state st_0 is made available to the log auditor who can use the Seek algorithm to check the integrity of log entries in any order. Clearly, the goal of forward security can be achieved only if the secure erasure of old states is an inherent part of the transition between epochs—for instance using the methods developed in [18].

3.4 Prior Constructions

While general sequential key generators have been considered in a variety of publications [4,9,3,11], the importance of seekability to obtain practical secure logging was only identified very recently [5]. By consequence, we are aware of only a single SSKG that precedes our current work.

Intuitively speaking, the SSKG construction from [5] follows the 'permute-then-hash' paradigm. In more detail, the authors consider so-called *shortcut one-way permutations* $\pi\colon \mathcal{D} \to \mathcal{D}$ that allow the evaluation of the k-fold composition π^k in less than $\mathcal{O}(k)$ time. Given such a primitive, state st_0 consists of a random element $x_0 \in \mathcal{D}$, and keys K_i are computed as $K_i = H(\pi^i(x_0))$, where H is a hash function modeled as a random oracle. The authors propose a number-theory-based shortcut permutation where π implements precisely the squaring operation modulo a Blum integer N; in this case, $\pi^i(x) = x^{2^i} = x^{2^i \bmod \varphi(N)}$ can be evaluated quite efficiently if the factorization of N is known.

4 SSKGs from Pseudorandom Generators

We propose a novel construction of a seekable sequential key generator that assumes only symmetric building blocks. Unlike the scheme in [5] which draws security from shortcut one-way permutations in the random oracle model, our new SSKG assumes just the existence of PRGs, i.e., it relies on a minimal cryptographic assumption. In a nutshell, similarly to the works in [2] and [1] that achieve forward-secure signing and forward-secure public key encryption, respectively, we identify time epochs with the nodes of specially formed trees and let the progression of time correspond to a pre-order visit of these nodes.

4.1 Sequential Key Generator from Binary Trees

From Section 2.2 we know that for any fixed $H \in \mathbb{N}^{\geq 1}$ the binary tree of constant height H has exactly $N = 2^H - 1$ nodes. In our SSKG we identify time epochs with the nodes of such a tree. More precisely, given the pre-order depth-first enumeration $w_0, \ldots, w_{N-1}$ of the nodes (first visit the root, then recursively the left subtree, then recursively the right subtree; cf. Figure 3), we let time epoch i and node w_i correspond.

The idea is to assign to each node w_i a (secret) seed $s_i \in \{0,1\}^\lambda$ from which the corresponding epoch's key K_i and the seeds of all subordinate nodes can be deterministically derived via PRG invocations. Here, exclusively the secret of the root node is assigned at random. Intuitively, pseudorandomness of the PRG ensures that all keys and seeds look random to the adversary.

We proceed with specifying which information the states associated with the epochs shall record. Recall that from each state st_i, $0 \leq i < N$, two pieces of information have to be derivable: the epoch-specific key K_i and the successor state st_{i+1} (and, by induction, also all following states and keys). Clearly, in our construction, the notions of seed and state do not coincide; for instance, in the

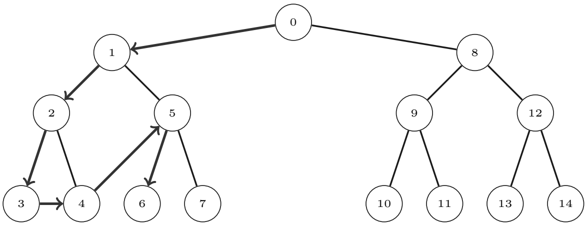

Fig. 3. A binary tree with height $H = 4$ and $N = 2^4 - 1 = 15$ nodes. The latter are numbered according to a pre-order depth-first search, as partially indicated by the arrow from the root node w_0 to node w_6.

tree of Figure 3 key K_9 cannot be computed from just seed s_4. However, if state st_4 contained (s_4, s_5, s_8), then for all $4 \leq i < N$ the keys K_i could be computed from this state. Inspired by this observation, our SSKG stores in each state st_i a collection of seeds, namely the seeds of the roots of the 'remaining subtrees'. The latter set of nodes is precisely what we called in Section 2.2 the *co-path* of node w_i. Intuitively speaking, this construction is forward-secure as each state stores only the minimal information required to compute all succeeding states. In particular, as each node precedes all vertices on its co-path (in the sense of a pre-order visit of the tree), the corresponding key remains secure even if any subsequent epoch's seed is leaked to the adversary.

We present next the algorithms of our SSKG construction. Particularly interesting, we believe, are the details on how the required pre-order depth-first search is implicitly performed by help of a stack data structure.

Construction 1 (TreeSSKG). *Fix a polynomial* $\ell\colon \mathbb{N} \to \mathbb{N}$ *and a PRG* $G\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+\ell(\lambda)}$. *For all* $s \in \{0,1\}^\lambda$ *write* $G(s)$ *as* $G(s) = G_L(s) \,\|\, G_R(s) \,\|\, G_K(s)$ *where* $G_L(s), G_R(s) \in \{0,1\}^\lambda$ *and* $G_K(s) \in \{0,1\}^{\ell(\lambda)}$. *Assuming the notation for stacks from Section 2.3, the algorithms* TreeSSKG = {GenSSKG, Evolve, GetKey, Seek} *of our SSKG are defined by Algorithms 1–4 in Figures 4 and 5.*

Let us discuss the algorithms of TreeSSKG in greater detail.

GenSSKG. Besides picking a random seed $s = s_0$ for the root node, Algorithm 1 computes the minimum number $h \in \mathbb{N}$ such that the binary tree of constant height h consists of at least N nodes (cf. Section 2.2). Observe that this tree might have more than N nodes, i.e., more epochs are supported than required. The algorithm stores in state st_0 a stack $\mathcal{S}$ that contains only a single element: the pair (s, h). Here and in the following such pairs should be understood as 'seed s shall generate a subtree of height h'.

Evolve. The stack $\mathcal{S}$ stored in state st_i generally contains two types of information: the top element is a pair (s, h) associated with the current node w_i, and the remaining elements are associated with the corresponding pairs of the nodes on

w_i's co-path. After taking the current entry (s, h) off the stack, in order to implement the depth-first search idea from Section 4.1, Algorithm 2 distinguishes two cases: if node w_i is an internal node (i.e., $h > 1$), the update step computes the seeds of its two child nodes using PRG G, starting with the right seed as it needs to be prepended to the current co-path. The new seeds $G_L(s)$ and $G_R(s)$ can be considered roots of subtrees of one level less than w_i; they are hence pushed onto the stack with decreased h-value. In the second case, if the current node w_i is a leaf (i.e., $h = 1$), no further action has to be taken: the next required seed is the 'left-most' node on w_i's co-path, which resides on the stack's top position already.

GetKey. Algorithm 3 is particularly simple as it requires only a single evaluation of PRG G. Observe that the $\mathbf{Peek}_1$ operation leaves its argument unchanged.

Seek. Deriving state st_k from the initial state st_0 via iteratively evoking k times the Evolve procedure is equivalent to visiting all nodes of the tree according to a pre-order traversal until reaching node w_k. However, there is an appealing way to obtain seed s_k more directly, without passing through all the intermediate vertices. The idea is to just walk down the path connecting the root node with w_k. Taking this shortcut decreases the seeking cost to only $\mathcal{O}(\log N)$, as opposed to $\mathcal{O}(N)$. This is the intuition behind the design of our Seek algorithm.

Algorithm 1: GenSSKG

```
  Input: $1^\lambda$, integer $N$
  Output: initial state $\mathsf{st}_0$
1 Init($\mathcal{S}$)
2 $s \xleftarrow{\$} \{0,1\}^\lambda$
3 $h \leftarrow \lceil \log_2(N+1) \rceil$
4 Push($\mathcal{S}$, $(s, h)$)
5 return $\mathcal{S}$ as $\mathsf{st}_0$
```

Algorithm 2: Evolve

```
  Input: state $\mathsf{st}_i$ as $\mathcal{S}$
  Output: next state $\mathsf{st}_{i+1}$
1 $(s, h) \leftarrow$ Pop($\mathcal{S}$)
2 if $h > 1$ then
3     Push($\mathcal{S}$, $(G_R(s), h-1)$)
4     Push($\mathcal{S}$, $(G_L(s), h-1)$)
5 return $\mathcal{S}$ as $\mathsf{st}_{i+1}$
```

Algorithm 3: GetKey

```
  Input: state $\mathsf{st}_i$ as $\mathcal{S}$
  Output: key $K_i$
1 $(s, h) \leftarrow$ Peek$_1$($\mathcal{S}$)
2 $K \leftarrow G_K(s)$
3 return $K$ as $K_i$
```

Fig. 4. Algorithms GenSSKG, Evolve, and GetKey. Observe that the number of supported epochs is potentially greater than N due to the rounding operation in line 3 of GenSSKG.

Recall that Seek is required to output the whole state st_k, and not just seed s_k. In other words, the execution of the algorithm needs to comprehend the construction of the co-path of node w_k. We provide details on how Algorithm 4 fulfills this task. Our strategy, illustrated in Figure 6, is to walk down the path from the root to node w_k, recording the right siblings of the visited nodes on a stack. During this process, with a variable δ we keep track of the remaining number of epochs that needs to be skipped. This counter is particularly helpful for deciding whether, in the path towards w_k, the left or the right child node have to be taken. Indeed, the number of nodes covered by the left and right subtrees is $2^h - 1$ each; if $\delta \leq 2^h - 1$ then the left child is the next to consider, but

```
Algorithm 4: Seek
    Input: state $\mathsf{st}_0$ as $\mathcal{S}$, integer $k$
    Output: state $\mathsf{st}_k$
    $\delta \leftarrow k$
    $(s, h) \leftarrow$ Pop($\mathcal{S}$)
    while $\delta > 0$ do
        $h \leftarrow h - 1$
        if $\delta < 2^h$ then
            Push($\mathcal{S}, (G_R(s), h)$)
            $s \leftarrow G_L(s)$
            $\delta \leftarrow \delta - 1$
        else
            $s \leftarrow G_R(s)$
            $\delta \leftarrow \delta - 2^h$
    Push($\mathcal{S}, (s, h)$)
    return $\mathcal{S}$ as $\mathsf{st}_k$
```

```
Algorithm 5: SuperSeek
    Input: state $\mathsf{st}_i$ as $\mathcal{S}$, integer $k$
    Output: state $\mathsf{st}_{i+k}$
    $\delta \leftarrow k$
    $(s, h) \leftarrow$ Pop($\mathcal{S}$)
    while $\delta \geq 2^h - 1$ do
        $\delta \leftarrow \delta - (2^h - 1)$
        $(s, h) \leftarrow$ Pop($\mathcal{S}$)
    while $\delta > 0$ do
        $h \leftarrow h - 1$
        if $\delta < 2^h$ then
            Push($\mathcal{S}, (G_R(s), h)$)
            $s \leftarrow G_L(s)$
            $\delta \leftarrow \delta - 1$
        else
            $s \leftarrow G_R(s)$
            $\delta \leftarrow \delta - 2^h$
    Push($\mathcal{S}, (s, h)$)
    return $\mathcal{S}$ as $\mathsf{st}_{i+k}$
```

Fig. 5. Algorithms Seek and SuperSeek

the right child has to be recorded for the co-path. On the other hand, if $\delta \geq 2^h$, then the left child can be ignored, the co-path doesn't have to be extended, and the walk towards w_k is continued via the right child. The procedure terminates when for the number of remaining epochs we have $\delta = 0$, which means that we arrived at target node w_k.

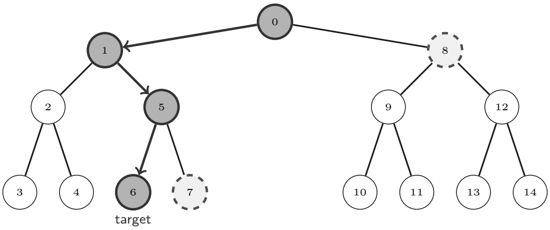

Fig. 6. A visualization of the procedure Seek computing state st_6. As the arrows indicate, the algorithm walks down the path from the root node w_0 to the target node w_6 (thick nodes); simultaneously, it records w_6's co-path, i.e., (w_7, w_8) (dashed nodes).

4.2 Security of Our Tree-Based SSKG

We next formally assess the security of Construction 1. For better legibility, in the following theorem we restrict attention to the setting $N = 2^H - 1$, i.e., where

$\log(N+1)$ is an integer; the extension to the general case is straightforward. We will also shorten the notation for some of the concepts from Definitions 2 and 4 (e.g., we denote $\ell(\lambda)$ simply by ℓ, etc.).

Theorem 1 (Security of TreeSSKG). *Assuming a secure PRG is used, our tree-based SSKG from Construction 1 provides* indistinguishability with forward security (IND-FS). *More precisely, for any efficient adversary $\mathcal{A}$ against the* TreeSSKG *scheme there exist efficient distinguishers $\mathcal{D}_i$ against the underlying PRG such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}FS}}_{N,\mathcal{A}} \leq 2(N-1) \sum_{i=1}^{\log(N+1)} \mathrm{Adv}^{\mathsf{PRG}}_{\mathcal{D}_i} \ .$$

Proof (sketch). The security of our scheme follows from the intuition that every SSKG key K_i, for being (part of) the output of a PRG invocation, looks like a random string to any efficient adversary as long as the corresponding seed remains hidden. Recall the IND-FS experiment (cf. Figure 2): the adversary gets state st_m and a challenge K_n^b —either key K_n or a random ℓ-bit string according to the value of b— for integers $n < m$ of her choosing. Although state st_m reveals seed s_m and subsequent seeds, from these seeds none of the preceding states can be computed. In other words, state st_m is of no help to the adversary in distinguishing keys prior to epoch m; in particular, key K_n remains secure.

To formalize this intuition we use game hops to progressively turn the IND-FS experiment into one for which all adversaries have advantage exactly zero. In the first hop we let the challenger guess the epoch $n < N$ corresponding to the challenge key and chosen by the adversary; this reduces the winning probability by a factor of $(N-1)$. Next, let $(v_1, \ldots, v_L)$ be the path from the root $v_1 = w_0$ to node $v_L = w_n$ in the binary tree associated to the SSKG. Starting from the previous game, we consider a hop for all $i = 1, \ldots, L$ by replacing the output of the PRG invocation associated to node v_i by a random $(2\lambda + \ell)$-bit string. Since each of the hops only involves a single PRG invocation, computational indistinguishability of any two consecutive games directly follows from the pseudorandomness of G. Observe that the last hop leads to a game where both K_n^0 and K_n^1 are uniformly chosen at random: here no adversary can do better than guessing. The fact that we lost a factor of $(N-1)$ in the first hop and we have additional $L \leq \log(N+1)$ intermediates games lets us derive the theorem statement.

A detailed proof appers in the full version of this paper [19]. □

4.3 An Enhanced Seeking Procedure

As required by Definition 2, our Seek algorithm allows computing any state st_k given the initial state st_0. Observe, however, that in many applications this initial state might not be accessible; indeed, forward security can be attained only if states of expired epochs are securely erased. From a practical perspective it is hence appealing to generalize the functionality of Seek to allow efficient computation of st_{i+k} from any state st_i, and not just from st_0. We correspondingly extend the notion of SSKG by introducing a new algorithm, SuperSeek, which realizes the Evolve^k functionality for arbitrary starting points; when invoked on input st_0, the new procedure behaves exactly as Seek.

Definition 5 (SSKG with SuperSeek). *A seekable sequential key generator* SSKG *supports* SuperSeek *if it has an auxiliary algorithm as follows:*

- SuperSeek. *On input of a state* st_i *and* $k \in \mathbb{N}$, *this deterministic algorithm returns state* st_{i+k}.

For correctness we require that for all $N \in \mathbb{N}$, *all* $\mathsf{st}_0 \overset{\$}{\leftarrow} \mathsf{GenSSKG}(1^\lambda, N)$, *all* $i, k \in \mathbb{N}$, *and* $\mathsf{st}_i = \mathsf{Evolve}^i(\mathsf{st}_0)$ *we have*

$$0 \le i \le i+k < N \qquad \Longrightarrow \qquad \mathsf{Evolve}^k(\mathsf{st}_i) = \mathsf{SuperSeek}(\mathsf{st}_i, k) \ .$$

Assume a TreeSSKG instance is in state st_i and an application requests it to seek to position st_{i+k}, for arbitrary $0 \le i \le i+k < N$. Recall from the discussions in Sections 4.1 that state st_i encodes both the seed s_i and the co-path of node w_i. Recall also that, as a property of the employed pre-order visit of the tree, for each state st_j, $j > i$, the co-path of node w_i contains an ancestor w of w_j. Following these observations, our SuperSeek construction consists of two consecutive phases. For seeking to state st_{i+k}, in the first phase the algorithm considers all nodes on the co-path of w_i until it finds the ancestor w of w_{i+k}. The second phase is then a descent from that node to node w_{i+k}, similarly to what we had in the regular Seek algorithm. In both phases care has to be taken that the co-path of target node w_{i+k} is correctly assembled as part of st_{i+k}. The working principle of our new seeking method is also illustrated in Figure 7. We present explicit instructions for implementing SuperSeek in Figure 5. The first while loop identifies the ancestor w of target node w_{i+k} on w_i's co-path by comparing δ (i.e., the remaining number of epochs to be skipped) with the number of nodes in the subtree where w is the root. The second loop is equivalent to the one from Algorithm 4.

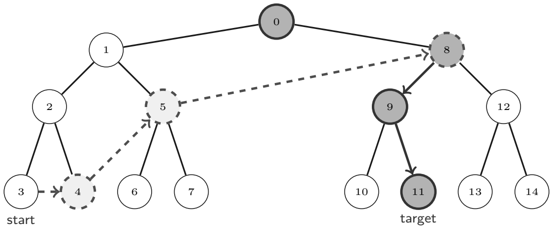

Fig. 7. A visualization of the procedure SuperSeek jumping from epoch 3 to 11. As indicated by the arrows, the algorithm first finds the intersection, here w_8, between the co-path of node w_3 (dashed nodes) and the path that connects the root with the target node w_{11} (thick nodes); from there it proceeds downwards until it reaches node w_{11}.

5 Practical Aspects

In the preceding sections we left open how PRGs can be instantiated in practice; indeed, the well-known recommendations and standards related to symmetric key cryptography exclusively consider block ciphers, stream ciphers, and hash functions. Fortunately, secure PRG instantiations can be boot-strapped from all three named primitives. For instance, a block cipher operated in counter mode can be seen as a PRG where the block cipher's key acts as the PRG's seed. Similar counter-based constructions derived from hash functions or PRFs (e.g., HMAC) are possible. A specific property of PRGs that are constructed by combining a symmetric primitive with a counter is particularly advantageous for efficiently implementing our TreeSSKG scheme. Recall that the PRG used in Construction 1 is effectively evaluated in a blockwise fashion. More precisely, while the PRG is formally defined to output strings of length $2\lambda + \ell(\lambda)$, in our TreeSSKG algorithms it is sufficient to compute only a considerably shorter substring per invocation. This property is perfectly matched by the 'iterated PRGs' proposed above, as the latter allow exactly this kind of evaluation very efficiently.

Implementation. We implemented our TreeSSKG scheme and claim that the level of optimization is sufficient for practical deployment. Our code is written in the C programming language and relies on the OpenSSL library [20] for random number generation and the required cryptographic primitives. We consider a total of four PRG instantiations, using the AES128 and AES256 block ciphers and the MD5 and SHA256 hash functions as described. That is, we have two instantiations at the $\lambda = 128$ security level, and two at the $\lambda = 256$ level.

We experimentally evaluated the performance of our implementation, using the following setup. We generated SSKG instances that support $N = 2^{20} - 1 \approx 10^6$ epochs. We iterated through all epochs in linear order, determining both the average and the worst-case time consumed by the Evolve algorithm. Similarly we measured the average and worst-case time it takes for the Seek algorithm to recover states st_k, ranging over all values $k \in [0, N-1]$. Concerning SuperSeek, we picked random pairs $i, j \in [0, N-1], i < j$, and measured the time required by the algorithm to jump from st_i to st_j. Finally, we performed analogous measurements for GenSSKG and GetKey (here, average and worst-case coincide). The results of our analysis are summarized in Table 1.

For comparison we also include the corresponding timing values of our competitor, the (factoring-based) SSKG from [5][1], for security levels roughly equivalent to ours. We point out that the analogue of GenSSKG from [5] in fact consists of two separate algorithms: one that produces public parameters and an associated 'seeking key', and one that generates the actual initial SSKG state. As any fixed combination of public parameters and corresponding seeking key can be used for many SSKG instances without security compromises, for fairness

---

[1] The reference implementation from [5] can be found at `http://cgit.freedesktop.org/systemd/systemd/tree/src/journal/fsprg.c`.

we decided not to count the generation costs of the former when indicating the GenSSKG performance in Table 1. Instead, we report the results of our timing analysis here as follows: for the costs of parameters and seeking key generation with 2048 bit and 3072 bit RSA moduli we measured $400ms$ and $2300ms$, respectively.

It might be instructive to also study the required state sizes for both our TreeSSKG scheme and the scheme from [5]. In our implementation the (maximum) state size scales roughly linearly in both $\log N$ and the seed length of the used PRG. Concretely, for $N = 2^{20} - 1$ and 128 bit keys (e.g., for AES128- and MD5-based PRGs) the state requires 350 bytes, while for 256 bit security a total of 670 bytes of storage are necessary. In the scheme from [5] the space in the state variable is taken by an RSA modulus N, a value $x \in \mathbb{Z}_N^\times$, a 64 bit epoch counter, and a small header. Precisely, for 2048 and 3072 bit RSA moduli this results in 522 and 778 bytes of state, respectively.

Results and discussion. We discuss the results from Table 1 as follows, beginning with those of our tree-based SSKG (i.e., columns AES128, MD5, AES256, and SHA256). Our first observation is that the GenSSKG time is independent of the respectively used PRG. This is not surprising as the former algorithm never invokes the latter, but spends its time with memory allocation and requesting the random starting seed from OpenSSL's core routines. The timings for Evolve indicate that, as expected, 128-bit cryptographic primitives are faster than 256-bit primitives, and that for a fixed security level the hash-function-based constructions are (slightly) preferable. The hypothesis that the time spent by the individual algorithms is dominated by the internal PRG executions is supported by the observation that the running time of Evolve (on average) and GetKey coincide, and that the worst-case running time of Evolve is twice that value; to see this, recall that Evolve executions perform either two internal PRG invocations or none, and that the average number of invocations is one. We understand that the SuperSeek timings are generally better than the Seek values as the first **while** loop in Algorithm 5 does not comprise a PRG invocation, whereas the second

Table 1. Results of efficiency measurements of our TreeSSKG algorithms when instantiated with different PRGs, and a comparison with the algorithms from [5]. All experiments were performed on an 1.90GHz Intel Core i7-3517U CPU. We used OpenSSL version 0.9.8 for the implementation of our TreeSSKG routines, while for the compilation of the reference code from [5] we used the gcrypt library in version 1.5.0.

	AES128		**MD5**		**[5]/2048 bit**	**AES256**		**SHA256**		**[5]/3072 bit**
	[average]	[max]	[average]	[max]		[average]	[max]	[average]	[max]	
GenSSKG	22μs		22μs		27μs	22μs		22μs		38μs
Evolve	0.2μs	0.5μs	0.2μs	0.4μs	8μs	0.5μs	1μs	0.4μs	0.8μs	13μs
Seek	7μs	9μs	6μs	7μs	4.9ms	14μs	18μs	11μs	15μs	12.6ms
SuperSeek	6μs	9μs	5μs	7μs	–	13μs	18μs	8μs	15μs	–
GetKey	0.2μs		0.2μs		12μs	0.4μs		0.4μs		13μs

while loop requires less iterations on average than the corresponding loop in Algorithm 4.

The routines from [5] are clearly outperformed by the ones from our SSKG. Firstly, for the Evolve algorithm our timing values are about 30 times better than those for [5] (recall that the latter's state update involves a modular squaring operation). Similar results show our tree-based GetKey algorithm to be faster, by a factor between 30 and 60, depending on the considered security level. This might be surprising at first sight, as the algorithm from [5] consists of just hashing the corresponding state variable, but presumably the explication for this difference is that [5] operates with considerably larger state sizes than we do. Finally, the superiority of our tree-based construction in terms of efficiency is made even more evident by studying the performance of the seek Seek algorithms, where we can report our routines to be 700–1000 times faster than those from [5], again depending on the security level.

Conclusion

The recently introduced concept of seekable sequential key generator (SSKG) combines the forward-secure generation of sequences of cryptographic keys with an explicit fast-forward functionality. While prior constructions of this primitive require specific number-theoretic building blocks, we show that symmetric tools like block ciphers or hash functions suffice for obtaining secure SSKGs; this leads to impressive performance improvements in practice, by factors of 30–1000, depending on the considered algorithms. In addition to the performance gain, our scheme enhances the functionality of SSKGs by generalizing the notion of seekability, making it more natural and concise, an improvement that we believe is very relevant for applications. Our scheme enjoys provable security in the standard model.

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

References

1. Canetti, R., Halevi, S., Katz, J.: A forward-secure public-key encryption scheme. Journal of Cryptology 20(3), 265–294 (2007)
2. Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
3. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg (2003)
4. Kelsey, J., Schneier, B.: Cryptographic support for secure logs on untrusted machines. In: Proceedings of the 7th USENIX Security Symposium (1998)
5. Marson, G.A., Poettering, B.: Practical secure logging: Seekable sequential key generators. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 111–128. Springer, Heidelberg (2013)

6. Marson, G.A., Poettering, B.: Practical secure logging: Seekable sequential key generators. Cryptology ePrint Archive, Report 2013/397 (2013), `http://eprint.iacr.org/2013/397`
7. Bellare, M., Yee, B.S.: Forward integrity for secure audit logs. Technical report (1997)
8. Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: Recent Advances in Intrusion Detection (1999)
9. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. 2(2), 159–176 (1999)
10. Chong, C.N., Peng, Z., Hartel, P.H.: Secure audit logging with tamper-resistant hardware. In: Gritzalis, D., di Vimercati, S.D.C., Samarati, P., Katsikas, S.K. (eds.) SEC. IFIP Conference Proceedings, vol. 250, pp. 73–84. Kluwer (2003)
11. Holt, J.E.: Logcrypt: forward security and public verification for secure audit logs. In: Buyya, R., Ma, T., Safavi-Naini, R., Steketee, C., Susilo, W. (eds.) ACSW Frontiers. CRPIT, vol. 54, pp. 203–211. Australian Computer Society (2006)
12. Accorsi, R.: BBox: A distributed secure log architecture. In: Camenisch, J., Lambrinoudakis, C. (eds.) EuroPKI 2010. LNCS, vol. 6711, pp. 109–124. Springer, Heidelberg (2011)
13. Ma, D., Tsudik, G.: Extended abstract: Forward-secure sequential aggregate authentication. In: 2007 IEEE Symposium on Security and Privacy, May 20-23, pp. 86–91. IEEE Computer Society Press, Oakland (2007)
14. Ma, D., Tsudik, G.: A new approach to secure logging. Trans. Storage 5(1), 2:1–2:2 (2009)
15. Yavuz, A.A., Ning, P.: BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In: ACSAC, pp. 219–228. IEEE Computer Society (2009)
16. Yavuz, A.A., Ning, P., Reiter, M.K.: BAF and FI-BAF: Efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems. ACM Trans. Inf. 15(2), 9 (2012)
17. Kelsey, J., Callas, J., Clemm, A.: Signed Syslog Messages. RFC 5848 (Proposed Standard) (May 2010)
18. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA, vol. 14 (1996)
19. Marson, G.A., Poettering, B.: Even more practical secure logging: Tree-based seekable sequential key generators. Cryptology ePrint Archive, Report 2014/479 (2014), `http://eprint.iacr.org/2014/479`
20. Young, E., Hudson, T.: OpenSSL: The Open Source Toolkit for SSL/TLS, `http://www.openssl.org`