SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:3180–3190
Published online 26 March 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1242

RESEARCH ARTICLE

BAFi: a practical cryptographic secure audit logging scheme for digital forensics

Panos Kampanakis[1]* and Attila A. Yavuz[2]

[1] Security Research and Operations, Cisco Systems, San Jose, CA, U.S.A
[2] University of Pittsburgh, Pittsburgh, PA 15260, U.S.A

ABSTRACT

Audit logs provide information about historical states of computer systems. They also contain highly valuable data that can be used by law enforcement in forensic investigations. Thus, ensuring the authenticity and integrity of audit logs is of vital importance. An ideal security mechanism for audit logging must also satisfy security properties such as forward-security (compromise resiliency), compactness, and computational efficiency. Unfortunately, existing secure audit logging schemes lack the computational or storage efficiency for modern performance requirements. Indeed, the practicality of such schemes has not been investigated in real-life systems, where logs generated in various occasions could be terabytes of data per day.

To address this limitation, we developed an efficient, publicly verifiable, forward-secure, privacy-preserving, and aggregate logging scheme called *blind-aggregate-forward improved* (BAFi). BAFi is based on BAF, with new properties and performance improvements as follows: (i) BAFi improves the efficiency of BAF via implementation specific optimizations; (ii) BAFi has the option to not expose sensitive information in logs to protect valuable forensic information; (iii) BAFi was experimentally tested in real-world logs; and (iv) BAFi improves the security of BAF against log substitution. Our analysis shows that BAFi outperforms previous alternatives with similar properties and therefore is an ideal solution for nowadays highly intense logging systems. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS

secure audit logging; applied cryptography; digital forensics; forward-secure; signature aggregation

***Correspondence**

Panos Kampanakis, Security Research and Operations, Cisco Systems, San Jose, CA, U.S.A.
E-mail: pkampana@cisco.com

1. INTRODUCTION

Log auditing is the most widely used forensic analysis and investigation methodology in information systems. Logs can present the previous state of a machine to provide data about events or failures of a system. Along with other tools at their disposal, security engineers use logs when investigating failures, security incidents, or compromises of computer systems. Moreover, after receiving legal authorization to access logs, law enforcement requests and uses logs from service providers to investigate law violations that are relevant to with electronic crimes.

Most logging infrastructures today leverage logging servers that can simultaneously collect data from several logging devices. Data are stored into a remote storage location where it is archived. Logs are purged after the time defined in the organization's retention policy passes. In case the logs need to be retrieved, they are extracted from the storage location for further examination. Logging servers cannot be considered 100% fault-tolerant or secure. In the event of a server compromise, logs can be potentially tampered with and/or modified. It is also possible that false data are spoofed and sent to the server in order to contaminate the log information. Thus, the integrity and security of stored logs are fundamental to ensure the validity of the log analysis results.

1.1. Motivation

As discussed earlier, logs can be used for forensic analysis and criminal investigation. Tamper-resistance is a property that ensures that logs are trustworthy, and thus, they serve their purpose (i.e., keep track of system events). Similarly, log re-ordering should not be possible. Especially in criminal cases in some countries, log evidence can only be used if a warrant is issued for their retrieval from the log holder

Copyright © 2015 John Wiley & Sons, Ltd.

(i.e., service provider). Personal identifiable information (PII) is considered private and subject to privacy laws. Immutable logging that also protects PII is more suitable for a legislative system that protects privacy. For example, a block of logs could be used to verify if a certain event actually happened, but it should not be possible to use it to see all the events that happened during that time. Another example of an application that would benefit from such a logging scheme is a voting system where user information should not be retrievable.

Another vital aspect of modern logging systems is logging performance. Internet service providers (ISP) are an example of the high performance requirements of modern logging aggregation points. As described in Internet Engineering Task Force's behave working group Internet draft [1], average log sizes for NATting and IPv6 technologies vary from 150 to 175 bytes. That could add up to 1.8 petabytes of data per year and 23 Mbps of logs per 50 000 users. These numbers do not consider compression and bulk logging and may vary depending on time of day and season, but they indicate that a log security system must have minimal computational overhead, especially during log generation. The computational efficiency of log verification is not as critical as that of log generation, because the verifications generally occur in an ad hoc manner (e.g., only during an investigation).

Consequently, in this work, we aim to address all these challenges by designing a scheme that is able to provide immutability, tamper resistance, and protection against truncation attacks [2] to optionally protect private information and to be very efficient in terms of log generation. Also, we minimize the storage overhead that is important for the practicality of our scheme in real-world deployments.

1.2. Related work and limitations

The research community has come up with various schemes to address the aforementioned problems. These schemes can be divided into symmetric and public keys. An important limitation of these schemes is the lack of analysis and evaluation in real systems. Some include basic prototypes to prove their correctness, which are not enough to ensure their practicality and applicability in real-life applications. Another limitation of these schemes is that they assume log privacy, which is not always practical.

Symmetric key schemes [3–7] rely on message authentication codes (MACs), one-way hash chains, and pseudo-random number generators in order to provide forward-secrecy and immutability. Even though these schemes are efficient for today's technology, they assume shared secrets between the logging server and the logger or an online trusted third party (TTP) that provides the shared keys. Shared keys pose distribution and significant storage challenges. On the other hand, the requirement of an online TTP is not very practical for today's systems. Moreover, in the event of log verification, all the shared secrets need to be shared with the verifier, which can compromise the security of other logs. Some of these schemes are also vulnerable to truncation and delayed detection attacks [2] and introduce authentication tags on a per log basis.

Bellare and Yee defined and analyzed forward-secure MACs and pseudo-random number generators [3,4]. They also proposed a forward-secure scheme that tags and indexes the logs. Schneier and Kelsey present schemes that leverage one-way hash chains with public key encryption and forward-secure MACs [5,7]. The disadvantages of these schemes have to do with computational, storage, and communication overhead. Internet Engineering Task Force's request for comments 5848 [8] uses these techniques to define secure logging messages. Logcrypt [9], on the other hand, secures the logs by generating MAC-based signatures. MAC-based Logcrypt is vulnerable to truncation attacks.

Public key schemes are based on public key cryptography. They rely on signatures to provide public verifiability, and their advantage is that they can be verified without compromising the log security by revealing shared keys to the verifier.

Ma *et al.* proposed a set of comprehensive secure audit logging schemes [2] based on their BM-forward-secure sequential aggregate authentication (FssAgg), AR-FssAgg, and BLS-FssAgg schemes [6,10]. These provide storage efficiency but introduce considerable processing load to the logging server that can prove to be impractical for the log rates of certain modern log aggregating applications. Identity-based encryption Logcrypt [9] uses ID-based elliptic-curve cryptography (ECC) signatures but lacks performance because of costly ECC extract operations. Yavuz and Ning, on the other hand, present BAF that leverages ECC aggregate signatures to provide an efficient forward-secure scheme [11]. BAF is the scheme that we adjusted for this work in order to produce practical and more efficient BAFi that can provide secure logging for today's infrastructures.

1.3. Our contributions

In order to address the aforementioned limitations, we develop BAFi. BAFi achieves the desirable properties found in the succeeding text.

- We improve BAF's verification performance by avoiding one elliptic-curve (EC) point addition per log. That provides an advantage if the auditor needs to verify a large amount of log entries simultaneously.
- We improve BAF's security against log substitution and known-plaintext attack by introducing truncation of the log hashes and proposing a new key.
- We provide a full-fledged implementation and experimental results, which prove that BAFi is efficient and could provide logging security for real-world systems.
- We provide real-world analysis of byte count overhead and show the advantages of BAFi for today's and future security requirements.
- We demonstrate BAFi log structure in detail, which can guide engineers to implement BAFi properly.

- BAFi can be used for forensic analysis in situations where PII must remain private. We show how a forensic analyst would verify the occurrence of the event without being able to retrieve any other events protected by BAFi.

Blind-aggregate-forward improved also maintains all the desirable properties of BAF [11] as follows:

(1) *Efficient log signing*
(2) *Logger storage/bandwidth efficiency*
(3) *Efficient log verification*
(4) *Public verifiability*
(5) *Off-line TTP and immediate verification*

The rest of this paper is organized as follows: Section 2 provides the preliminary concepts for the scheme. Section 3 presents the updates introduced by BAFi. Section 4 presents performance analysis and compares BAFi with previous approaches. Section 5 discusses the experimental implementation results of BAFi compared with other schemes. Section 6 concludes this paper.

2. PRELIMINARIES

2.1. System model

In a modern logging system, multiple logging devices (aka loggers) send logs to a central entity (e.g., log server) that is responsible for aggregating and storing the logged data usually to a storage location. Logs are stored according to the data retention policy and in case of an investigation or event analysis they are available in order to provide information regarding event history. Given that the logging server (and potentially other devices) has write access to the log storage, a compromise of the logging server or bogus data spoofing could jeopardize the log validity and thus the investigation results. Our scheme is leveraging an off-line TTP that communicates with the logging server to provide keying materials. Communication between the TTP and the logger can then stop until key refreshes are necessary. Figure 1 shows the BAFi log infrastructure. In a BAFi infrastructure, in the event of a logging server compromise, subsequent stored data can be spoofed or tampered with, but all existing logs are secure thus trustworthy.

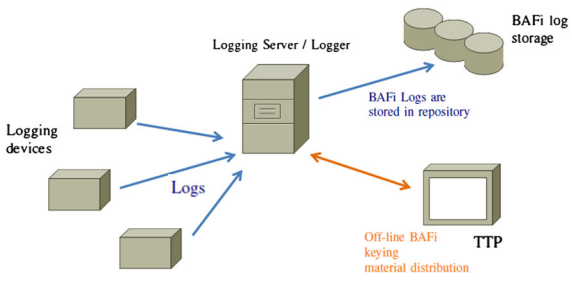


Figure 1. Blind-aggregate-forward improved (BAFi) infrastructure with logging server/aggregator, trusted third party (TTP) and multiple logging devices.

2.2. Notation

In the scheme of this paper, $x \xleftarrow{R} F_p$ denotes that x is selected uniformly from prime field F_p, where p is a large prime number. G is a generator of group $\mathbb{G}$ defined on an EC $E(F_p)$ of F_p. q is the order of G. kG denotes a *scalar multiplication*, where $k \in [1, q-1]$. Additionally, H_1 and H_2 are two distinct full-domain hash functions [12], which are defined as $H_1 : \{0,1\}^{|sk|} \rightarrow \{0,1\}^{|q|}$ and $H_2 : \{0,1\}^{*} \rightarrow \{0,1\}^{|q|}$, respectively, where $sk \xleftarrow{R} F_q$. Operators $\|$ and $|x|$ denote the concatenation operation and the bit length of variable x, respectively. $\lfloor x \rfloor_{y\ bits}$ depicts the truncation of x to its rightmost y-bits.

Our scheme depends on the following semantic security properties [13]: H_1/H_2 are strong collision-resistant and secure full-domain hashes [12], producing indistinguishable outputs from the random uniform distribution (i.e., behaves as a random oracle [14]). EC discrete logarithm problem [15] is intractable with appropriate parameters. That is, for a given random point $Q \in E(F_p)$, it is computationally infeasible to determine an integer k such that $Q = kG$, where $G \in \mathbb{G}$.

2.3. Blind-aggregate-forward[11]

In the succeeding text, we outline our base scheme, BAF [11]. BAF operates in blocks of logs. For this work, we will assume that the blocks are of L logs. For each block, BAF generates an aggregate signature that can be verified if and only if all the logs that participated in the signature have not been tampered with. BAF consists of a four-step algorithm summarized in the succeeding text:

- **BAF.Kg**(L, ID_{si})**:** *BAF.Kg* is the key generation algorithm, which takes the maximum number of key updates L per block and identity ID_{si} of signer si as the input and returns L public keys, initial secret keys, and index $n \xleftarrow{R} F_p$ for ID_{si} as the output. *BAF.Kg* is run by the TTP off-line and practically generates the keys that will be provided to the logging server si in order to secure the logs and to the verifier to verify them. The parameter block size L determines the maximum number of key update operations that a signer can execute, which should be decided according to the application requirements. The identity of the server (ID_{si}) is necessary for the TTP to be able to track keys generated for different servers. The steps are

  (1) The TTP picks two random numbers as $(a_0, b_0) \xleftarrow{R} F_q$, which are the *initial blinding keys* of the block of logs. Also pick a random index number as $n \xleftarrow{R} F_p$, which is used to

DOI: 10.1002/sec

preserve the order (sequentiality) of individual signatures.
(2) It generates two hash chains from the initial secret blinding keys (a_0, b_0) as $a_{j+1} = H_1(a_j) \bmod q$ and $b_{j+1} = H_1(b_j) \bmod q$ for $j = 0, \ldots, L-2$. Also generate a public key for each element of these hash chains as $(A_j = a_j G$ and $B_j = b_j G)$ for $j = 0, \ldots, L-1$.
(3) The TTP, when needed, provides required keys to the signer $\leftarrow \{a_0, b_0\}$ and verifiers $\leftarrow \{ID_{si} : A_0, B_0, \ldots, A_{L-1}, B_{L-1}, n\}$.

Note that depending on the application, keying materials can be provided to the signer for multiple blocks at a time.

- **BAF.Upd**(a_l, b_l)**:** *BAF.Upd* is the key update algorithm, which takes the current secret key as input and returns the next secret key as the output. This algorithm is run by the logger L times for every block of logs, after each *BAF.ASig* operation. The keys are updated as follows: $a_{l+1} = H_1(a_l) \bmod q$ and $b_{l+1} = H_1(b_l) \bmod q$. *BAF.Upd* then deletes (a_l, b_l) from memory.
- **BAF.ASig**$(\sigma_{0,l-1}, D_l, a_l, b_l)$**:** *BAF.ASig* is the aggregate signature generation algorithm, which takes the secret keys a_l, b_l from $BAF.Upd(a_{l-1}, b_{l-1})$, where $l \in [0, L-1]$, a data item $D_l \in \{0,1\}^*$ to be signed, and an aggregate signature $\sigma_{0,l-1}$ (for previously accumulated data items) as input and returns an aggregate signature $\sigma_{0,l}$ by folding the individual signature of the data item into the aggregate signature. This algorithm is run by the log server in order to secure each log as follows:

  (1) Compute the individual signature σ_l as $\sigma_l = a_l{}^* d_l + b_l \bmod q$, where $l \in [0, L-1]$, $d_l = H_2(D_l \| (n+l))$.
  (2) Fold σ_l into $\sigma_{0,l-1}$ as $\sigma_{0,l} = \sigma_{0,l-1} + \sigma_l \bmod q$, where $l \in (0, L-1]$ and $\sigma_{0,0} = \sigma_0$.
  (3) Delete $\sigma_{0,l-1}$ from memory and invoke $BAF.Upd(a_l, b_l)$.

- **BAF.AVer**$(n, D_0, \ldots, D_{L-1}, \sigma_{0,L-1}, ID_{si})$**:** *BAF.AVer* is the aggregate signature verification algorithm, which takes $(D_0, \ldots, D_j) \in \{0,1\}^*$, its associated aggregate signature $\sigma_{0,L-1}$, index, public keys $(A_0, \ldots, A_{L-1})$, $(B_0, \ldots, B_{L-1})$ of ID_{si}, and index n as the input. If the signature is successfully verified, *BAF.AVer* returns *success*. Otherwise, it returns *failure*. This algorithm is run by the forensic analyst every time he or she needs to verify the validity of a block of BAF signed logs. The signature is verified using the following equation:

$$\sigma_{0,L-1} G \stackrel{?}{=} \sum_{j=0}^{L-1} (d_j A_j + B_j) \tag{2.1}$$

where $d_j = H_2(D_j \| (n+j))$.

The security model and a proof sketch of BAF are presented in [11].

3. PROPOSED UPDATES: BLIND-AGGREGATE-FORWARD IMPROVED

Blind-aggregate-forward improved introduces changes that further optimize BAF's performance and makes its use in real-world systems more practical. BAFi also satisfies the properties described in Section 1. Following the syntax given in Section 1.3 in the succeeding text, we only present the updates that BAFi contains compared with BAF that was presented in Section 2.3.

- **BAFi.Kg**(L, ID_{si}):

  (1) *BAFi.Kg* does not generate random index number n. Blinding keys a_0, b_0 are still generated.
  (2) After generating each $B_0, \ldots, B_{L-1}$, the TTP only keeps an aggregate key $B_{0,l} = \sum_{j=0}^{l} B_j$, which in the end is aggregated to $B_{0,L-1} = \sum_{j=0}^{L-1} B_j$ for the block of logs. $B_{0,L-1}$, along with the $A_0, \ldots, A_{L-1}$, are the only public keys needed for the signature verification. $B_0, \ldots, B_{L-1}$ are deleted in the interest of storage space at the TTP.
  (3) After providing keys a_0, b_0 for the block to the signer, in order to minimize storage overhead, the TTP can optionally only store $a_0, B_{0,L-1}$. a_0 is enough to regenerate the public keys $A_0, \ldots, A_{L-1}$ and along with $B_{0,L-1}$ make them available to the verifier. The overhead of this scenario is that for every log verification the TTP needs to recalculate $A_j = a_j G, \forall j \in [0, L-1]$. If there is a concern of the TTP being compromised, the TTP could instead only store $A_0, \ldots, A_{L-1}, B_{0,L-1}$. The storage overhead for that scenario would be the extra $A_1, \ldots, A_{L-1}$ key storage.

  The TTP, when needed, provides required keys to the Signer $\leftarrow \{a_0, b_0\}$ and Verifiers $\leftarrow \{ID_{si} : A_0, \ldots, A_{L-1}, B_{0,L-1}\}$.

- **BAFi.ASig**$(\sigma_{0,l-1}, D_l, a_l, b_l)$:

  (1) The logger generates a timestamp *ts* for the first log D_0. One *ts* is stored for each block of logs. *ts* is used in place of n in BAF, to preserve the order (sequentiality) of individual signatures. It also adds time context to the stored log block and ensures that hashes are not vulnerable to rainbow-table types of attacks. A new block of L logs is not expected to fall in the same

timestamp as the previous block in a real-world system. Thus, the granularity of the timestamp *ts* should be enough to provide distinct timestamps for the minimum time of generating L logs during a busy period. In the event of time moving backwards in a real-world system, timestamps in a block cannot be changed, but the time difference can be accounted for when verifying a block of logs within a time interval.

In BAF, a malicious attacker that compromised the logging server could be substituting log D_l with its congruent modulo q ${D_l}^*$ (${D_l}^* = D_l \bmod q$). Then, σ_l would stay intact, keeping the aggregate signature for the block the same, which would make the scheme susceptible to log substitution going undetected. For that reason, in BAFi, $d_l = H_2(D_l||(ts + l))$ is truncated to $|q| - 1$-bits, which would prevent a collision modulo q of two hashes. As seen later in this section, in practical scenarios $|H_2(\ldots)| \sim |q| \gg H_2(\ldots)/2$, which makes the truncation secure against birthday attacks.

The individual signature σ_l is now computed by the logging server as $\sigma_l = a_l{*}d_l + b_l \bmod q$, where $l \in [0, L-1]$ and $d_l = \lfloor H_2(D_l||(ts + l))\rfloor_{|q|-1\ bits}$. The aggregate signature is computed the same way as in BAF.

In cases where the set of pre-images of the log is small, D_i can still be guessed by a persistent attacker that obtained access to a block of log hashes $d_0, \ldots, d_{L-1}$, timestamp *ts*, and signature. Even though some logging applications generate logs that contain highly fluctuating information in them, like time, addresses, or names, others give a limited number of outputs. To avoid such *known-plaintext attack* in that case, BAFi can incorporate an extra secret key from the TTP used in the hashes. That key could be used in place of the timestamp *ts* in the d_i calculations. That key would also need to be provided to the verifier by the TTP when verifying the logs. Log storage would still need a timestamp per block of logs, which would serve for looking up the right block of logs when verifying BAFi signatures.

To protect PII, BAFi allows for the data items to no longer be stored as they are (in cleartext). Instead, the signer can store $d_l = \lfloor H_2(D_l||(ts + l))\rfloor_{|q|-1\ bits}$ and delete D_l. One more advantage of such an approach is that, for applications that log more bytes than the actual size of H_2 function's output, BAFi reduces storage. Of course, the drawback is that the log D_l cannot be reconstructed back from its hash d_l. If the logs are encrypted for privacy, the encrypted logs should be stored as they are ($Encrypted(D_l)$) where $l \in [0, L-1]$). d_l does not need to be stored.

- **BAFi.AVer**($ts, d_0, \ldots, d_{L-1}, \sigma_{0,L-1}, ID_{si}$): When the verifier wants to verify one or more logs in a block, he or she receives data items $d_0, \ldots, d_{L-1}$ given in a time interval of two timestamps (ts_i, ts_{i+1}) and their associated aggregate signature $\sigma_{0,L-1}$ from the logger. The verifier first ensures that $\forall i \in [0, L) : d_i < q$, which prevents log substitution. Alternatively, the verifier might only have access to the cleartext logs $D_0, \ldots, D_{L-1}$. Then, he or she can easily generate $d_0, \ldots, d_{L-1}$ as $d_l = \lfloor H_2(D_l||(ts_i + l))\rfloor_{|q|-1\ bits}$.

  Then, after receiving public keys (A_j) for $j = 0, \ldots, L-1$ and $B_{0,L-1}$ from the TTP, he or she can verify $\sigma_{0,L-1}$ via the BAFi's verification equation

$$\sigma_{0,L-1} G \stackrel{?}{=} B_{0,L-1} + \sum_{j=0}^{L-1} d_j A_j \tag{3.1}$$

  If the equation holds, *BAFi.AVer* returns *true*. Otherwise it returns *false*.
- **BAFi.LVer**($Dy, d_0, \ldots, d_{L-1}$): If the cleartext logs $D_0, \ldots, D_{L-1}$ of a block that was generated within times (ts_i, ts_{i+1}) are not stored to protect PII, BAFi introduces one more algorithm for a verifier to be able to check if a log D_y exists in a block $d_0, \ldots, d_{L-1}$ in the time interval with associated aggregate signature $\sigma_{0,L-1}$. After checking, *BAFi.AVer* returns *true,* the verifier can be certain that the log content and sequence stored in $d_0, \ldots, d_{L-1}$ have not been tampered with. Then, the verifier can confirm the existence of one specific log D_y in the block using Algorithm 1.

Algorithm 1 Verify log D_y exists in L data items $d_0, \ldots, d_{L-1}$ in a block of logs that happened within the expected time interval between timestamps ts_i, ts_{i+1}.

```
j ← 0
while (j < L) do
  k ← 0
  while (k < L) do
    if (⌊H2(Dy||(tsi + j))⌋|q|−1 bits == dk) then
      return True
    end if
    k++
  end while
  j++
end while
return False
```

Figure 2 illustrates the BAF algorithms described earlier.

Blind-aggregate-forward improved for real-world forensic auditing: For a modern day logging system, the cryptographic algorithms used in BAFi must ensure that today's and future technological advancements will not be able to pose a threat to the scheme's security. At the same time, performance needs to be maximized. Accord-

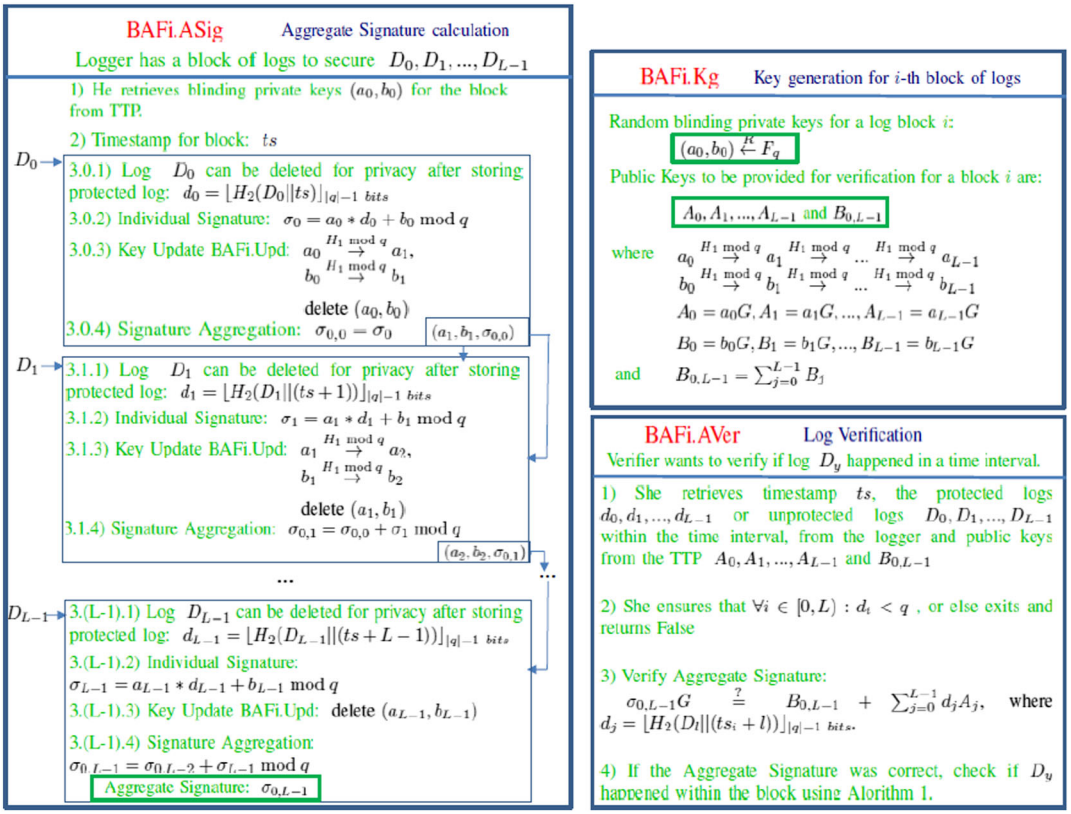


Figure 2. Blind-aggregate-forward improved (BAFi). TTP, trusted third party.

ing to National Institute of Standards and Technology's SP800-131A [16], 128-bit level of security is required for today's systems. Additionally, National Security Agency's Suite B [17] algorithms state that secure hash algorithm (SHA)-256 and elliptic curve Diffie–Hellman (ECDH) over P-256 curves can be used to secure SECRET level data. SHA-384 and ECDH over P-384 curves can be used to secure TOP SECRET level data [18]. Given that the curves over F_p that are defined in [19] to be used in ECDH types of schemes have order q, where $|q| \sim |p|$, choosing H_1 and H_2 hash functions to be SHA-256 or SHA-384 provides levels of security adequate for today's and future processing capabilities. Optionally, the recently selected SHA-3 algorithm, Keccak [20], could be used for BAFi's hash functions as well. The curves to be used that will provide same levels of security for the *ECDH* part of our scheme ($A_i = a_i G, B_i = b_i G$) are the P-256 or P-384 curves [19]. The *ts* timestamp that is added in every block of BAFi logs should be at least 2^{sec_level} [21], where *sec_level* is the assumed security level. So, for today's and future applications timestamps of $|ts| = 256$ will be more than sufficient.

As far as the log size in the applications, where BAFi is used, is concerned, it could vary. The logs could be a few hundred bytes long, or they could grow bigger. For example, the size of just a four-tuple (source and destination address and ports) log in a network application would be 12 and 36 bytes for IPv4 and IPv6, respectively. Such applications could be ISP logs that contain information of customer Internet activity. It is evident that such logs would also contain information like Hypertext Transfer Protocol (HTTP) URLs, Domain Name System (DNS) requests, or Simple Mail Transfer Protocol (SMTP) fields that could vary from 10 bytes to a few hundred bytes. Other cases that involve heavy ISP logging could be Carrier Grade Nat (CGN) logs [1]. The log sizes of such applications are 150–200 bytes long. The rates can be 33 000 per second per subscriber [1] or even higher with the proliferation of more and more smart devices (log compression and bulk logging are not considered for these numbers). Other logging application examples are NAT64 where high-speed [22] logs are of similar size as in CGN.

Rocket-fast system for log processing's [23] author states that experience has shown that the average log size is 60–80 bytes with routinely 200 000 messages per second [24]. Other billing applications produce heavier logs of size (0.5–1.5 kB) with rates of 150 000 messages per second. From the examples earlier, it is evident that log sizes vary and depend on the applications that will be leveraging BAFi. Most applications generate logs that are a few hundred bytes long, and some of them can have very high generation rates (close to millions per second in some cases). The higher the average log size, the less the storage overhead BAFi introduces, which comes with a slight performance drop because of the hash calculations.

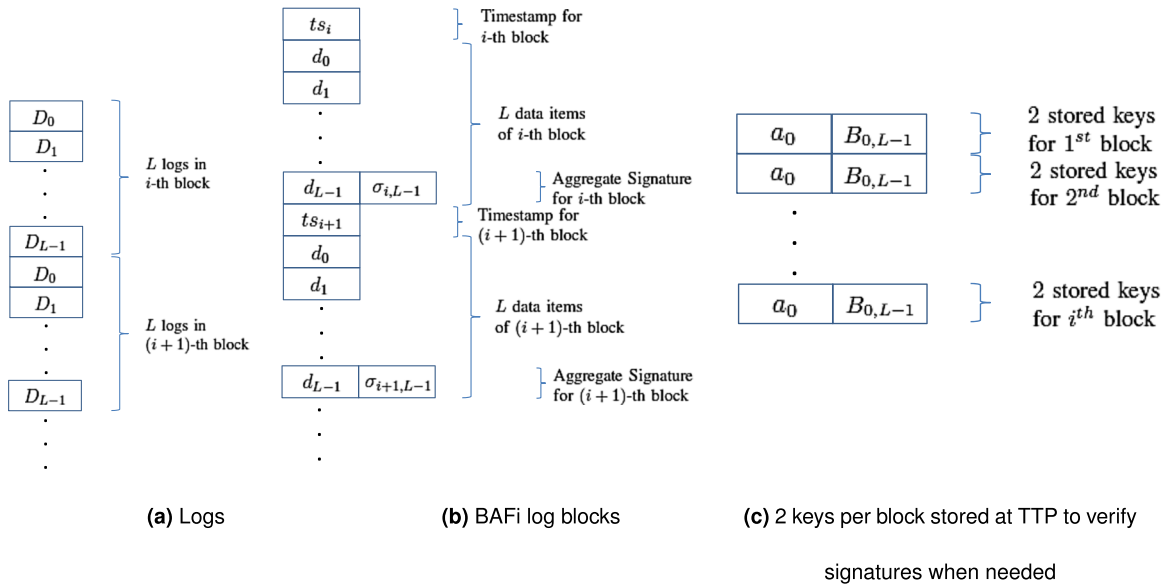
Figure 3 shows the format of the data that is stored by the signer and by the TTP. The reader should note that the cleartext logs $D_0, \ldots, D_{L-1}$ of a block can be deleted by the signer after $\sigma_{0,L-1}$ has been stored. That will ensure that private information in the logs is no longer retrievable.

4. PERFORMANCE ANALYSIS FOR REAL-WORLD APPLICATIONS

With the real-world information on curves and hash algorithms provided in Section 3, it is critical to evaluate the computational load and storage overhead BAFi introduces and to compare it with other publicly verifiable schemes. We compare BAFi with FssAgg-BLS [6], FssAgg-BM [2,10], FssAgg-AR [2,10], and Identity-based encryption Logcrypt [9]. Notice that we do not compare BAFi with FssAgg-MAC [6] or any other symmetric key schemes because they are not publicly verifiable and introduce key management problems as discussed before. Our goal is to provide a publicly verifiable technique without the need of secret shared keys being provided to verifiers.

Table II summarizes the storage and computational overhead that each scheme introduces. Table I provides the notation for Table II. In the computational overhead, we are not calculating modular and scalar addition because it can be considered negligible for today's computation speeds. As seen in Table II, the key and operation size of FssAgg-BM and FssAgg-AR are very large, which increases the key and signature sizes. Thus, the computational and storage overhead of BAFi is much less compared with those schemes. FssAgg-BLS, on the other hand, has less key storage overhead depending on the curves chosen, but the *MtP* and *PR* operations make FssAgg-BLS extremely inefficient compared with BAFi. Thus, it is clear that BAFi introduces a relatively low storage and computational overhead that can serve successfully to secure today's logging infrastructures.

It is also important to note that in most modern day logging systems, the log signature speed is more important than the signature verification because the verification will usually happen ad hoc by the forensic analyst. Thus, the higher computational cost of signature verification and the extra $\mathcal{O}(L^2)$ search Algorithm 1 for log verification (when only d_i is stored) is not a critical limitation for BAFi. Using a dictionary, bloom filter search tree with the log hashes could reduce Algorithm 1's complexity to $\mathcal{O}(L)$, if the overhead introduced with these structures is not a concern for the logging application.



(a) Logs **(b)** BAFi log blocks **(c)** 2 keys per block stored at TTP to verify signatures when needed

Figure 3. Data stored and used by blind-aggregate-forward improved (BAFi). (a) logs, (b) BAFi log blocks, and (c) two keys per block stored at trusted third party to verify signatures when needed.

Table I. Notation for Table II.

Mulx: modular multiplication mod x	*EMul*: ECC scalar multiplication over F_p	l: number of data item to be processed
Exp: modular exponentiation mod p	*MtP*: ECC map-to-point operation	L: maximum number of key updates
Sqr: modular squaring mod n	*PR*: ECC pairing operation	x: number of bits in FssAgg keys
H: hash operation	*GSig*: generic signature generation	

ECC, elliptic-curve cryptography; FssAgg, forward-secure sequential aggregate.

Table II. Computational and storage overhead.

	BAFi	FssAgg-BLS [6]	FssAgg-BM [2,10]	FssAgg-AR [2,10]	Logcrypt [9] *
			Computation per log		
KeyUpd	$2H$ ‡	H	$(x+1)Sqr$	$(2x)Sqr$	—
Sig	$H + Mulq$	$MtP + EMul + Mulp$	$(1+\frac{x}{2})Muln$	$x \cdot Sqr + (2+\frac{x}{2})Muln$	$GSig$
Ver	$(l+1) \cdot EMul$	$l \cdot (EMulPR + PR)$	$L \cdot Sqr + (l+\frac{l \cdot x}{2})Muln$	$x(L+l)Sqr+$ $2l(1+\frac{x}{2})Muln$	$l \cdot GVer$
			Storage cost (in bits)		
Key size	$2\lvert q\rvert + 1$ §	$\lvert q\rvert$	$(x+1)\lvert n\rvert$	$2\lvert n\rvert$	$\lvert q\rvert$
Sig size	$\lvert q\rvert$	$\lvert q\rvert + 1$ ¶	$\lvert n\rvert$	$\lvert n\rvert$	$2\lvert q\rvert + 1$
Extra storage	$3\lvert q\rvert + 1$ †	$2\lvert q\rvert + 1$	$(x+2)\lvert n\rvert$	$3\lvert n\rvert$	$O(L)*\lvert q\rvert$
			Bit sizes for schemes (128-bit security level)		
	$\lvert q\rvert \approx \lvert p\rvert = 256$	$\lvert q\rvert = 160$, $\lvert p\rvert = 197$ (BN curves [25])	$\lvert n\rvert = 2048$	$\lvert n\rvert = 2048$	$\lvert q\rvert = 256$

BAFi, blind-aggregate-forward improved; FssAgg, forward-secure sequential aggregate.

* Identity-based Logcrypt without signature aggregation.

‡ Extra cost of $(2 \cdot Emul)$ (can be generated off-line) for the trusted third party.

§ $B_{0,L-1}$'s size is $\lvert q\rvert + 1$ using point compression.

¶ Elleptic curve point size using point compression.

† $(L+2) \cdot \lvert q\rvert$ if public keys $A_0, \ldots, A_{L-1}, B_{0,L-1}$ are stored.

5. EXPERIMENTAL RESULTS

We now provide experimental results that will substantiate the theoretical performance analysis and comparison given in Section 4. For our experiments, we implemented BAFi using the Multiprecision Integer and Rational Arithmetic C Library (MIRACL) library [26]. Our experiments were run on 32-bit Ubuntu 11.04 Linux VM running in Oracle VirtualBox. The host machine had 64-bit Pentium 4 with 4 GB or RAM.

In our implementation, we stored the logs in files, which were read sequentially in blocks. Then, BAFi log hashes and aggregate signatures were stored in a new file. Verifying the signatures in the new file required to sequentially parse the BAFi file. Moreover, we used the Comba optimization for modular multiplication [27] and optionally used pre-computations for EC point multiplication. We aimed to provide 128-bit and 192-bit security level and thus used National Institute of Standards and Technology's SECP256R1 and SECP384R1 curve parameters [19]. Our implementation can easily be ported to various systems using the MIRACL library.

The logs we used for our experiments were of 32-byte, 128-byte, 256-byte, and 512-byte lengths that span various logging application log sizes. The results were measured with Comba optimization for modular multiplication enabled and without using precomputations for EC point multiplication. The reason is that precomputation significantly increases the time to prepare for the point multiplication, and as public keys change, the benefit of precomputation is lost during the precomputations for multiple public keys. Table III shows the average execution times for key update, signature generation, and verification per log for different log sizes and 256-bit and 384-bit curves. The reader should note that the aggregate log verification times include the file reads from the BAFi file. The key update time in the range of microsecond is average, and therefore, its overhead is insignificant for the logger. Similarly, the signature generation is extremely fast, especially compared with other publicly verifiable alternatives.

As discussed in Section 3, CGN logs can be generated at rates of 33 000 per second, and rocket-fast system for log processing has commonly observed 200 000 per second. That implies that the key update and signature generation times per log entry should not exceed 0.030 and 0.005 ms, respectively. Running BAFi in a regular user computer (for our tests) renders a total of 0.017 + 0.0025 = 0.0195 ms (for 128-byte logs and 256-bit curve). Thus, BAFi can meet the efficiency requirements of even highly demanding real-world applications.

The reader should notice that BAFi signature verification is more computationally demanding; however, as discussed before, in most cases, log verification by forensic analysts is not required to be real-time or nearly as efficient as log generation (especially because only rarely do logs need to be investigated).

Table III. Blind-aggregate-forward improved experimental results.

Average time per log	32-byte log	128-byte log	256-byte log	512-byte log
KeyUpd (ms)	0.0024	0.0025	0.0024	0.0025
Signature (ms)	0.009	0.017	0.029	0.043
Sig Verification (ms) *	6.35	6.35	6.35	6.35
		SECP384R1		
KeyUpd (ms)	0.0028	0.0028	0.0028	0.0029
Signature (ms)	0.012	0.029	0.034	0.05
Sig Verification (ms)	16.4	16.4	16.4	16.4

* After running experiments in a more powerful host computer (Intel i5 2.6 GHz processor, 8 GB RAM), we obtained fourfold improvements in the average signature verification times.

Table IV. Blind-aggregate-forward improved average search time per log.

Log-size (bytes)	Search time (ms)
128	0.134
256	0.155
512	0.215

In BAFi, if the log hashes d_i are chosen to be stored to protect PII, then Algorithm 1 adds one more step in the log verification process. In order to estimate the processing time required for matching a log in a BAFi log block whose aggregate signature has been verified, we implemented the algorithm in MIRACL [26]. We run various tests, and the results are included in Table IV. The table shows the times for both SECP256R1 and SECP384R1 curve parameters because the curve does not affect the algorithm processing. The times include the file read time. We can see that depending on the log-size, the average times range from 135 to 215 μs per log, which is very efficient especially because only rarely do logs need to be investigated. After running experiments in a more powerful host computer (Intel i5 2.6 GHz processor, 8 GB RAM), we obtained sixfold improvement in average log search time per log entry.

We now compare BAFi with existing alternatives using extensive experiments. We compare BAFi with FssAgg-BM and FssAgg-AR, which are publicly verifiable and compromise resilient (forward-secure) alternatives. Even though our tests showed that the BAFi average signature time per log is slightly lower than MAC-based Logcrypt's and the average signature time 30 times more than Logcrypt's, we do not present experiments with symmetric schemes because our goal is to provide a publicly verifiable technique without the need of secret shared keys provided to verifiers.

FssAgg-BM and FssAgg-AR were implemented using number theory library. In order to compare the performance of these schemes with BAFi, we wanted to provide the same security level (128-bit) with BAFi using SECP256R1 curve; thus, we used prime field of 2048 bits. We also wanted to compare BAFi with FssAgg-BLS [6] that is based on ECC. In order to be able to obtain a performance estimate for FssAgg-BLS and because there was no existing implementation of the scheme that we were aware of, we implemented the operations used in FssAgg-BLS, FssAgg.Upd, FssAgg.ASig, and FssAgg.AVer steps in MIRACL, and we were able to obtain the average performance times for each step. In order to provide 128-bit security, we used Barreto Naehrig (BN) curves with k = 12 [25]. For 80-bit security level (even though it is less secure), supersignular curves [28] of k = 2 were tested in order to see if they performed better compared with BN curves. All the tests were run on the same 32-bit Ubuntu 11.04 Linux VM running in Oracle VirtualBox that also runs the BAFi experiments. The host machine had 64-bit Pentium 4 with 4 GB or RAM.

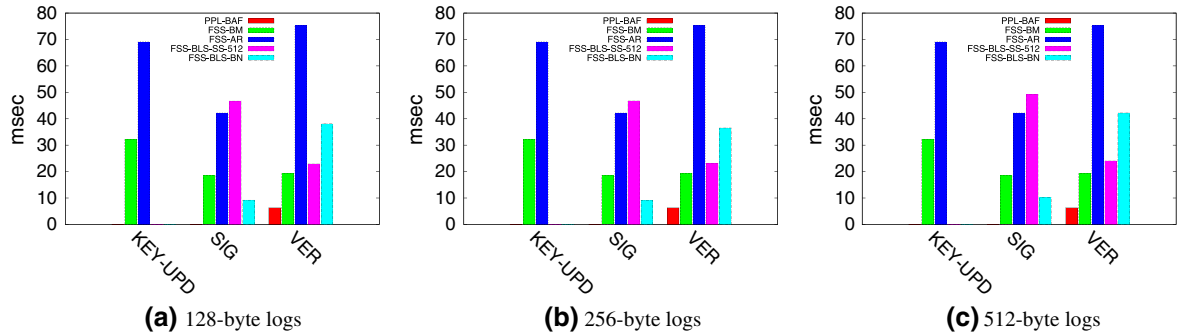


Figure 4. Blind-aggregate-forward improved (BAFi) versus forward-secure sequential aggregate (FssAgg)-BM, FssAgg-AR **[2,10]**, and FssAgg-BLS **[6]**. (a) 128-byte, (b) 256-byte, and (c) 512-byte logs.

DOI: 10.1002/sec

Table V. Key.Upd in BAFi versus FssAgg-BLS.

Average Key.Upd time per log (μs)	BAFi	FssAgg-BLS (SS-512 curve)	FssAgg-BLS (BN curve)
128-bit logs	2.5	1.1	1.2
256-bit logs	2.4	1.3	1.2
512-bit logs	2.5	1.2	1.2

BAFi, Blind-aggregate-forward improved; FssAgg, forward-secure sequential aggregate.

Figure 4 shows the experimental results and the comparison between BAFi and FssAgg-BM, FssAgg-AR, and FssAgg-BLS. Someone might notice that the Key.Upd steps of ECC-based schemes (BAFi and FssAgg-BLS) times are near zero. Key.Upd for these schemes is in the range of microsecond, which is far less than the scheme signature and verification times. Table V shows the actual Key.Upd performance of BAFi compared with FssAgg-BLS. In Figure 4, we can also see that BAFi has significantly better signature times compared with all FssAgg variants. Regardless of the log size, the BAFi average time per log is in the range of microsecond where the FssAgg schemes range from 10 to 50 ms. As far as the FssAgg schemes themselves, the optimal seems to be FssAgg-BLS using BN curves. As for aggregate signature verification, as it is clear in Figure 4, BAFi offers much better performance of around 5 ms where the FssAgg schemes range from 19 to 75 ms. The FssAgg schemes themselves seem to have FssAgg-BM as optimal for signature verification. We can also observe that the log size does not significantly affect the times, which is because the signature calculations are greatly higher than calculating the message hash going in the calculations. From the analysis earlier, it is obvious that BAFi is significantly more efficient in terms of key update, signature, and signature verification compared with FssAgg-BM, FssAgg-AR, and FssAgg-BLS with key sizes that offer the same security level. For key updates, BAFi and FssAgg-BLS have comparable performance of range in microsecond.

6. CONCLUSION

In this paper, we developed a secure audit logging scheme, BAFi, which improves BAF schemes [11,29]. BAFi improves BAF's security, the privacy of audit logs, and implements cryptospecific optimizations. In contrast to previous alternatives just providing basic prototypes, BAFi has been extensively experimented in real-life log scenarios. Our analysis indicates that it is much more computational and storage efficient than previous alternatives (with a signing and verification overhead just in a few microseconds and milliseconds, respectively). Moreover, BAFi preserves desirable properties of BAF schemes such as forward-security, signature aggregation, and public verifiability. Proven with theoretical and experimental analysis, BAFi is an ideal choice to meet the cryptographic secure logging requirements of task intensive real-life applications.

ACKNOWLEDGEMENT

We would like thank Dr. Di Ma for providing us with the FssAgg [6,10].

REFERENCES

1. Donley C, Grundemann C, Sarawat V, Sundaresan K, Vautrin O. *Deterministic address mapping to reduce logging in carrier grade NAT deployments*, RFC-ID 2013. http://datatracker.ietf.org/doc/draft-donley-behave-deterministic-cgn/.
2. Ma D, Tsudik G. A new approach to secure logging, *Proceedings of the 22nd annual IFIP WG 11.3 working conference on data and applications security (dbsec '08)*, London, 2008; 48–63.
3. Bellare M, Yee BS. *Forward Integrity for Secure Audit Logs*. University of California at San Diego: San Diego, CA, USA, 1997.
4. Bellare M, Yee BS. Forward-security in private-key cryptography, *Proceedings of the the cryptographers track at the rsa conference (ct-rsa '03)*, San Francisco, 2003; 1–18.
5. Schneier B, Kelsey J. Cryptographic support for secure logs on untrusted machines, *Proceedings of the 7th conference on USENIX security symposium*, USENIX Association, New Orleans, 1998.
6. Ma D, Tsudik G. Forward-secure sequential aggregate authentication, *Proceedings of the 28th IEEE symposium on security and privacy (S&P '07)*, Oakland, 2007; 86–91.
7. Schneier B, Kelsey J. Secure audit logs to support computer forensics. *ACM Transaction on Information System Security* 1999; **2**(2): 159–176.
8. Kesley S, Clemm A, Callas J. Signed syslog messages. IETF RFC 5848 2010.
9. Holt JE. Logcrypt: forward security and public verification for secure audit logs, *Proceedings of the 4th Australasian workshops on grid computing and e-research (ACSW '06)*, Tasmania, Australia, 2006; 203–211.
10. Ma D. Practical forward secure sequential aggregate signatures, *Proceedings of the 3rd ACM symposium on*

information, computer and communications security (ASIACCS '08), ACM: NY, USA, 2008; 341–352.
11. Yavuz AA, Ning P. BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems, *Proceedings of 25th annual computer security applications conference (ACSAC '09)*, Hawaii, 2009; 219–228.
12. Bellare M, Rogaway P. The exact security of digital signatures: how to sign with RSA and Rabin, Springer-Verlag: 1996; 399–416.
13. Goldreich O. *Foundations of Cryptography*. Cambridge University Press: Cambridge, 2001.
14. Bellare M, Rogaway P. Random oracles are practical: a paradigm for designing efficient protocols, *Proceedings of the 1st ACM conference on computer and communications security (CCS '93)*, ACM: NY, USA, 1993; 62–73.
15. Boneh D. The decision Diffie–Hellman problem, *Proceedings of the third algorithmic number theory symposium, LNCS*, Uppsala, Sweden, 1998; 48–63.
16. National Institute of Standards and Technology (NIST). *Transitions: recommendation for transitioning the use of cryptographic algorithms and key lengths*, 2011. http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf.
17. National Security Agency (NSA). *NSA Suite B Cryptography*, 2009. http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml.
18. Committee on National Security Systems. *National information assurance policy on the use of public standards for the secure sharing of information among national security systems*, 2012. http://www.cnss.gov/Assets/pdf/CNSSP_No.pdf.
19. NIST. *Recommended elliptic curves for federal government use*, 1999.
20. National Institute of Standards and Technology (NIST). *NIST selects winner of secure hash algorithm (SHA-3) competition*, 2012. http://www.nist.gov/itl/csd/sha-100212.cfm.
21. National Institute of Standards and Technology (NIST). *Recommendation for password-based key derivation*, 2010. http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf.
22. Cisco Systems. *High-speed logging for NAT64*, 2012. http://www.cisco.com/en/US/docs/ios-xml/ios/ipaddr_nat/configuration/xe-3s/asr1000/iadnat-stateful-nat64.html#GUID-83AD4883-15EA-4E03-BF57-00F2E592AED6.
23. rsyslog, 2013. http://www.rsyslog.com/ rsyslog log processing.
24. IETF rsyslog discussion, 2013. http://www.ietf.org/mail-archive/web/behave/current/msg10719.html.
25. Pereira GCCF, Naehrig M, Simplicio MA, Jr, Barreto PSLM. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software* 2011; **24**(8).
26. Certivox. *Multiprecision integer and rational arithmetic c/c++ library (MIRACL)*, 2013. https://certivox.com/solutions/miracl-crypto-sdk/.
27. Comba PG. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal* 1990; **29**(4): 526–538.
28. Galbraith SD. Supersingular curves in cryptography, Springer-Verlag: 2001; 495–513.
29. Yavuz AA, Ning P, Reiter MK. BAF and FI-BAF: efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems. *ACM Transaction on Information System Security* 2012; **15**(2).