LAVA: Log Authentication and Verification Algorithm

EDITA BAJRAMOVIC, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
CHRISTOFER FEIN, Albstadt-Sigmaringen University, Germany
MARIUS FRINKEN, PAUL RÖSLER, and FELIX FREILING, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Log files provide essential information regarding the actions of processes in critical computer systems. If an attacker modifies log entries, then critical digital evidence is lost. Therefore, many algorithms for secure logging have been devised, each achieving different security goals under different assumptions. We analyze these algorithms and identify their essential security features. Within a common system and attacker model, we integrate these algorithms into a single (parameterizable) "meta" algorithm called LAVA that possesses the union of the security features and can be parameterized to yield the security features of former algorithms. We present a security and efficiency analysis and provide a Python module that can be used to provide secure logging for forensics and incident response.

CCS Concepts: • **Security and privacy** → **Public key encryption**; *Tamper-proof and tamper-resistant designs*; *Security protocols*; Database activity monitoring;

Additional Key Words and Phrases: secure logging, authenticity, incident management, forensics

ACM Reference format:
Edita Bajramovic, Christofer Fein, Marius Frinken, Paul Rösler, and Felix Freiling. 2023. LAVA: Log Authentication and Verification Algorithm. *Digit. Threat. Res. Pract.* 4, 3, Article 35 (October 2023), 17 pages.
https://doi.org/10.1145/3609233

1 INTRODUCTION

Secure logging is the process of generating log entries with security properties that make any alteration or manipulation easily detectable. The usual models of secure logging [Accorsi 2009; Schneier and Kelsey 1999] distinguish between two entities: a *logging device* that executes the logging algorithm and the actual *log* holding the log entries. While the logging device is usually trusted, the log is untrusted and is subject to attacks. This corresponds to practical settings where logs are kept on external storage that can be accessed by more entities than the logging device. Secure logging protocols generally must ensure that the log itself still satisfies some security guarantees.

Work was also supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019).
Authors' addresses: E. Bajramovic, M. Frinken, P. Rösler, and F. Freiling, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstrasse 3, Erlangen, Germany; emails: {edita.bajramovic, marius.frinken, paul.roesler, felix.freiling}@fau.de; C. Fein, Albstadt-Sigmaringen University, Anton-Guenther-Straße 51, Sigmaringen, Germany; email: fein@hs-albsig.de.

2576-5337/2023/10-ART35 $15.00
https://doi.org/10.1145/3609233

According to a survey by Accorsi [2009], there are many different guarantees offered by secure logging protocols. Apart from trivial ones like *entry accountability* (the fact that log entries carry information about the logged event), the main security requirements are *accuracy* (“log entries were not modified”), *completeness* (“log entries were not deleted”), and *compactness* (“log entries were not illegally appended”). An auditor who checks the logs must be able to detect any violation of these properties (*tamper evidence*). Sometimes the trust assumptions on this auditor also vary: While in most cases the auditor is fully trusted (and can also potentially change log records), in other cases the auditor is not trusted with this ability and can only verify the properties of the log (*semi-trusted verifier*). Some logging protocols even ensure *log entry confidentiality*, i.e., the fact that the content of the log is only accessible to authorized entities.

1.1 Previous Protocols for Secure Logging

There are many secure logging protocols with different security properties. To the best of our knowledge, Bellare and Yee [1997] initially evaluated the security logs generated and stored on physically unprotected and untrusted logging system. They used **message authentication codes (MACs)** built on secret key cryptography and established the role of *verifier* to validate the security properties of log entries. Bellare and Yee [1997] identified *forward-integrity* as a fundamental security property of log files, meaning that the compromise of a key must not pose the risk to the integrity of (too many) already protected messages. Forward-integrity is achieved through regular renewal of keys that are used to secure log entries.

Thus, considering this idea, Schneier and Kelsey [1999] developed an approach that combines a hash-chain and forward-secure MACs to detect log alterations and protect its integrity and confidentiality. In addition to a fully trusted verifier, for the purpose of evaluating the integrity of the log without making any changes, they propose a *semi-trusted* verifier. Still, their secure logging protocol does not provide adequate protection related to removing finite suffixes of log entries (*truncation attacks*).

Using asymmetric cryptography, Holt’s Logcrypt system [Holt 2006] extends Schneier and Kelsey [1999]. This secure logging approach makes sure that integrity of the logs can be checked by any system that holds an (initial) authentic public key. Holt [2006] also indicated that *metronome messages* can be used to protect against truncation attacks. Accorsi [2008, 2010] introduced a secure logging protocol called BBox. This protocol uses trusted hardware (in the form of Trusted Platform Modules [Trusted Computing Group 2005]) and public key cryptography to protect against truncation attacks. All the above protocols allow for *partial verification*, i.e., a verifier can verify arbitrary prefixes of logs without having to look at the entire log.

The aim of Ma and Tsudik [2009] is also to prevent truncation attacks, but they use a continuously updated aggregate signature and therefore sacrifice partial verification. Since their aggregation technique relies on hiding earlier versions of the (aggregated) signature, this approach does not work for append-only logs (that reveal the entire history of the log). Blass and Noubir [2017] propose to map log messages to random entries in the log so that the attacker cannot distinguish new from old entries. Based on some secret, the verifier is able to replay the (random) entry sequence and thereby detect whether a log entry is missing.

Freiling and Bajramovic [2018] established a framework in which the major secure logging approaches can be compared. They reformulated the protocols of Schneier and Kelsey [1999], Holt [2006], Accorsi [2010], and Ma and Tsudik [2009] within a unified system and attacker model, thus exhibiting many structural similarities. They did only focus on *integrity* properties of the log and not on *confidentiality*.

We also must mention the family of syslog protocols that are the most commonly implemented solution on Unix systems for logging system events. Syslog was first described by Lonvick [2001] and later standardized by Gerhards [2009]. Apart from adding the role of a *relay*, the original syslog offers no possibilities for secure logging. This is in line with the Windows Event Log format introduced by the operating system *Microsoft Windows Vista* and all succeeding Microsoft Windows operating systems and described by Schuster [2007] and Metz [2016].

The original syslog family was extended in several ways. The idea of Syslog-sign [Kelsey et al. 2010] is to provide origin authentication, message integrity, replay resistance, message sequencing, and discovery of missing messages to syslog. Each signature block in Syslog-sign is cryptographically signed and contains the hashes of already sent syslog messages [Kelsey et al. 2010]. However, the keys are not regularly renewed. A similar approach was proposed by Huang et al. [2018] and Putz et al. [2019], only it is based on a blockchain to ensure integrity and authenticity. Lee et al. [2019] use ARM Trustzone to securely protect the keys used to compute signatures for syslog-ng.

Also based on syslog-ng, Blass and Marvedel [2020] present an advanced secure logging service that ensures forward integrity and confidentiality by encrypting each log with a unique one-time cryptographic key and using cryptographic authentication codes to protect integrity of the entire log archive. Log verification, however, must go through the log from start to end, which can be demanding when verifying large log files.

1.2 Contributions

The structural similarities of secure logging protocols observed by Freiling and Bajramovic [2018] lead to the question of whether all these protocols are merely instances of a more fundamental protocol idea. The aim of this article is to unify the domain of secure logging and integrate the central algorithmical ideas of previous secure logging protocols into a single meta protocol.

More specifically, we identify five properties of secure logging protocols that can be expressed using a single numerical parameter,

- *communication efficiency*: the number of log entries combined in a single message from logging device to the log. This basically determines how quickly events are logged.
- *authentication efficiency*: the number of log entries that are jointly authenticated. This determines the overhead incurred by the use of cryptography.
- *strength of forward integrity*: the maximum number of log entries that are affected by a successful key compromise. Basically this determines the overhead of key renewal within the protocol.
- *strength of truncation resistance*: the maximum number of contiguous log entries that an attacker can delete from the tail of the log. This parameter determines the communication between logging device and log even if no logging events occur.
- *verification efficiency*: the number of log entries that can be skipped during verification to speed up the checking of large log files, a method that we call *fast-forward log verification*. We are not aware of any previous work that has attempted to speed up log verification in this way.

We then develop a general **Log Authentication and Verification Algorithm (LAVA)** along these five properties that can be instantiated to result in the core of (almost) all previous secure logging protocols. It is a complete redesign of the preliminary and unpublished version of LAVA by Bajramovic et al. [2019] and considers the additional property of verification efficiency and provides a security analysis. Furthermore, given the new notion of verification efficiency, we show that there is a fundamental tradeoff between *full* and *fast* log verification. We argue later under which assumptions fast-forward log verification ensures the same security properties as full log verification.

To summarize, we make the following contributions in this article:

(1) We identify five parameters that can be used to express the main integrity properties of secure logging algorithms.
(2) We outline the meta protocol LAVA that can be instantiated using the five parameters to result a flexible array of secure logging protocols, including the core functionalities of the major logging protocols from the literature.
(3) We provide a security and efficiency analysis and show that LAVA satisfies the main security goals against realistic adversaries and allows constant factor speedup for the verification of large log files.
(4) We make LAVA available to the community as a Python module.

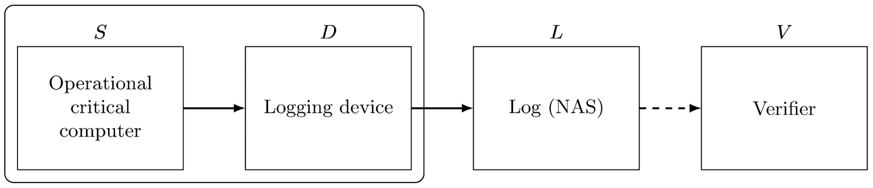

Fig. 1. Overall system model.

Similarly to Freiling and Bajramovic [2018], we focus on the main (integrity) properties and disregard confidentiality. For simplicity, we assume a fully trusted verifier.

1.3 Roadmap

The remainder of the article is organized as follows: We first describe our system and attacker model in Section 2. We then distill the main security parameters from previous protocols and formulate a parameterized interface by which the logging algorithm may be used in Section 3. Section 4 outlines the LAVA meta protocol followed by an example in Section 5. We present a security and efficiency analysis in Section 6. We give some details on the open source Python implementation of the protocol by the LAVA project team [2023] in Section 7. We conclude in Section 8.

2 SYSTEM AND ATTACK MODEL

The system model is shown in Figure 1. There is a single *operational critical computer* S whose actions are to be monitored. Directly attached to S is a *logging device* D such that when an event e happens on S, a callback is invoked on D signaling e. The code executed in the callback translates e into one or multiple *log entries* following an algorithm (of which the execution model is described below) that are subsequently sent (i.e., written) to the log.

The *log* L is a machine that, similarly to a network-attached storage device, can be used to read and write data to local storage. In our setting, the log merely receives a stream of log entries from D and writes then to its local storage. For simplicity, we assume that the log is organized as a file to which new entries are appended at the end.

We also assume a *verifier* V, i.e., a computer who is fully trusted and has access to the log. The purpose of the verifier is to perform audit checks on the log to ensure the security properties.

2.1 Algorithms and Execution Model

We formulate our algorithms in an event-based style similarly to the notation of Cachin et al. [2011]. As an example, consider Algorithm 1, which describes the basic control loop executed on the logging device and which is the basis for all further developed protocols. In general, an algorithm consists of a set of variables of different types that are initialized to certain values. The algorithm itself is expressed as a sequence of *actions* (*event handlers*) that also express the condition under which they should be executed.

The execution of the algorithm proceeds as follows: The initial values of the variables define the initial state of the algorithm. The execution then considers all actions in sequential order. For each action, the event condition is evaluated. If the condition evaluates to true, then the assignments associated with the event are executed. If the condition evaluates to false, then the dependent code is skipped. In any case, the execution then turns to the next action, and so on. If the execution reaches the end of all actions, then it simply starts from the beginning. Event conditions can be either simple Boolean expressions on the state of the algorithm or the occurrence of other events.

As an example, Algorithm 1 shows a possible "plain" logging algorithm with no security features enabled that can be run on D. It has a simple log entry buffer b as variable. If S triggers an event e, then the first (and only) action is executed and the log entry corresponding to e is written to the buffer. Subsequently, a message containing b is sent to L, and the buffer is emptied. The log itself is assumed to write the content of any message received from D to the (end of the) log.

ALGORITHM 1: Control Loop executed on D

```
variables
    $b$                                              ▹ text buffer for one log entry
end variables
actions
    upon event ⟨ $S$ triggers event $e$ ⟩ do
        $b \leftarrow$ log entry corresponding to $e$
        send $b$ to $L$                              ▹ add $b$ to the log
        $b \leftarrow empty$
    end event
end actions
```

We assume a clock that ticks in periods no faster than a minimal time interval (necessary to execute the event loop) and that S generates at most one event per tick, meaning that S cannot exceed a maximum event production rate. If event handlers are "fast" (i.e., do not contain infinite loops or otherwise slow computations), then we assume that the control loop can be executed once per tick and so all events produced by S can be handled by the algorithm.

2.2 Use of Cryptography

Since we focus on integrity properties of secure logging, we need ways to express the computation of a digital signature or a cryptographic message authentication codes. Although symmetric key cryptography could be used, too, for simplicity, we formulate algorithms using public key cryptography to generate digital signatures as follows. We assume a secure digital signature system consisting of the following three efficient algorithms:

- KGen() computes a public (pk) and private (sk) key pair.
- Sign(sk, m) computes the digital signature s on message m using key sk.
- Verify(pk, s, m) checks if the digital signature s is valid on message m using pk and returns either *true* or *false*.

We assume a secure signature algorithm to be existentially unforgeable against adaptive chosen-message attacks [Goldwasser et al. 1988]. To simplify the exposition, we call the secret key of the key pair (sk, pk) a *credential*. Similarly, the term *authenticator* is used to refer to a digital signature.

We assume a secure hash function as the efficient algorithm $H(m)$, which computes a distinctive fixed-size output for arbitrary input message m. We assume a secure hash function to be collision-resistant [Damgård 1988].

2.3 Attacker Model

Although security tools are used to protect the logging infrastructure, we assume that some parts of the system can be compromised by an attacker. Depending on the depth of infrastructure compromise, we distinguish two attacker models, strong and weak, that can be totally ordered in the sense that a strong attacker also includes the abilities of a weak one:

- A *weak attacker* has compromised (only) the Log, i.e., it can arbitrarily read and write data to L.
- A *strong attacker* has compromised both the Logging Device D and the Log L, i.e., it can read and write arbitrary data to L and can execute any functionality on D (apart from breaking the used cryptographic primitives).

Obviously, for strong attackers no security properties can be guaranteed after the system has been compromised. We therefore distinguish the time before and after an attack.

A system component that is not affected by the attacker performs correctly and therefore can be trusted. We assume the verifier to always operate correctly.

3 SECURITY PROPERTIES OF LOGGING PROTOCOLS

We now formulate five important properties of secure logging protocols that can be parameterized and expressed using a natural number.

3.1 Communication Efficiency

We begin with a way to express the *communication efficiency* of a secure logging protocol. As shown above, the logging device D buffers log entries and sends them to log L and so incurs communication resources. More sent log entries means higher resource usage that can be reduced if more than one log entry is sent per message. In our work, communication overhead is expressed using parameter b, where b is the number of log entries to be sent in one message (and therefore "*b*uffered" on the logging device). We assume b to be a natural number larger than 0. When $b = 1$, every log entry is sent as one message. The larger values of b reduce the communication overhead. Smaller values allow for "quicker" logging.

3.2 Authentication Efficiency

Authenticity of log entries refers to the fact that the verifier can validate the authenticator associated with the log entry. This verifies that the logging device created the log entry. As long as the logging device is not compromised, it creates only log entries when the corresponding event happened on S (see Algorithm 1). So even in the presence of a weak attacker (that can arbitrarily read/write to L), verification of a log entry implies that the corresponding event previously happened on S [Freiling and Bajramovic 2018]. So, overall, authenticity is achieved through adding authenticators (e.g., digital signatures) to log entries.

Similarly to communication efficiency, it is possible to append an authenticator to each log entry individually, but it is also possible to authenticate multiple log entries using a single authenticator, resulting in a different overhead. We define *authentication overhead* using a natural number a, where a is the number of jointly authenticated log entries. If $a = 0$, then no authenticators are added and we have no authenticity, so we disregard this case. If $a = 1$, then one authenticator is produced per log entry, leading to maximum overhead. The larger parameter a is, the better the ratio between size of log entries and size of the authenticator is, so with increasing values of a the authentication efficiency increases, too. However, for large values of a authenticators will be computed for large concatenated messages that might affect performance, so basically a is also a measure for the overhead incurred by the use of cryptography.

3.3 Forward Integrity

As noted above, perpetual integrity can only be achieved against weak attackers that cannot affect the logging device. This does not mean that all forms of integrity are lost in the presence of strong attackers, because an attacker needs to have access to the credentials to manipulate the entire log. If credentials are regularly updated, then "old" entries can be safe despite strong attackers. Integrity can therefore still be achieved in a finite sense [Bellare and Yee 1997].

More precisely, *forward integrity* means that at any point in time an attacker can only forge a bounded number of log entries in the past. Forward integrity is achieved through regular credential update and it is expressed as parameter c, where c is the number of log entries that a strong attacker at most can forge in the past. This is equivalent to number of log entries between credential updates. The larger parameter c is, the less frequent we update credentials. This is more efficient but also more dangerous, as it allows attackers to forge log entries until next verification. A value of $c = 1$ means that the credential is updated with every new log entry.

3.4 Truncation Resistance

Truncation attacks consist of removing a finite suffix of log entries from the log. Since a verifier does not necessarily know how many log entries were written at any point in time and every prefix of a log file is usually also a valid log file, such attacks are notoriously difficult to protect against. The approach developed by Ma and Tsudik [2009] continuously updates one central log entry and (necessarily) overwrites the previously stored value in that location. By reading this value (and hiding old versions thereof), a verifier can check the consistency with respect the number of stored log entries.

Protecting log entries using standard append-only logs (like we assume) is slightly more complicated, as the entire history of the log is always visible and no prior information can be hidden. The common approach is to generate (redundant) *metronome events* at regular time intervals that subsequently recorded in the log and are known to the verifier (e.g., every minute). Therefore, the blocking of such log entries or the truncation of the log can be detected after a certain time, i.e., when expected log entries are not present.

We formalize truncation resistance as a parameter d, where d correlates with the maximum number of log entries that the attacker can truncate from the log. This is equivalent to the amount of time (number of clock ticks) after which (at the latest) a dummy event is generated by S. When parameter d is equal to 1, the operational computer "invents" a dummy event every clock tick unless a regular event has happened. This effectively means that a strong attacker can truncate at most one log entry from the log and an attack is detectable immediately. We assume that d is at least 1. The higher the value of d, the less dummy events are generated leading to better efficiency but more possibilities for the attacker to truncate the log without this being immediately detectable. Note that d is measured in clock ticks and is therefore relative to the execution speed of the algorithm. It has no real unit of measurement and could be, e.g., 1 second, 1 hour, or 1 day.

3.5 Verification Efficiency

Depending on the protocol used for secure logging, there are many ways to check the integrity of the log. The standard way to do this for append-only logs is to rewind the logging process and recheck every entry starting from a trusted synchronization point at the start. For large log files, this process can be rather expensive and a good method for secure logging should provide a method with which this can be done faster.

The degree to which the speed of verification can be improved can be measured in many different ways. Here we use the simplest possible way, namely the factor by which the verification can be improved with respect to the number of log entries. We formalize this as a parameter e, which stands for the number of log entries that can be verified "in one step" (i.e., skipped) by the verifier when performing verification. If $e = 1$, then there would be no possibility of a fast forward through the log by the verifier. The larger e is, the higher the speedup for V.

Verification efficiency e is related to authentication efficiency a, but while a aims at improving the authentication effort for the logging device D, e improves the effort that needs to be invested by the verifier V. In practice, e should be several orders of magnitude larger than a. Despite these structural similarities, skipping log entries during verification naturally implies a risk of overlooking a manipulation (like a deleted or overwritten log entry). As we explain below, this is a fundamental limitation of fast-forward log verification. The precise conditions of correctness will be formulated in the security analysis.

Table 1. Summary of Parameters of LAVA

measure	param.	meaning
authentication efficiency	a	number of log entries per authenticator
communication efficiency	b	number of log entries sent in one message
forward integrity	c	number of log entries per new credential
truncation resistance	d	number of clock ticks per dummy event
verification efficiency	e	number of log entries that can be jointly skipped during verification

All have a minimum value of 1.

3.6 Summary

The five parameters are summarized in Table 1. In the next section, they will be used to "configure" our meta protocol LAVA.

4 LAVA

We now explain our generic log authentication and verification algorithm (Algorithm 2) by incrementally explaining code parts. The basic data structures are a counter i for the number of log entries processed so far, the current value of the global hash chain h, and an output queue Q to which items can be added on the logging device and which eventually is sent and written to the log (line 41) and reset to collect new data (line 43). The core control loop of LAVA is similar to the control loop sketched in Algorithm 1: If event e happens on S, then the logging device adds the log entry l corresponding to e to Q, extends the hash chain and increments the counter i (see lines 15–18).

4.1 Communication Efficiency

To enable communication efficiency, we collect several log entries in Q and send them as a "batch" in one message to the log (line 42). The parameter b determines when this is done and therefore b is part of the trigger condition for sending the content of Q (line 41). Note that i is incremented with every regular log entry (which is based on an event on S), but as we will see below, more elements can be added to the log with a single event (authenticators and credentials).

4.2 Authentication Protocol

The idea to ensure authenticity is to compute an authenticator on a sequence of log entries generated by the system using a credential. These authenticators are also added to the output queue Q and are part of the data that are sent to the log as just described.

More precisely, for every a log entries added to the Q, a digital signature is computed and then also added to Q (lines 20 and 21). The credential A used for creating the digital signature is initialized in line 10. It must be shared between logging device and verifier, i.e., both must initially have the same trustworthy credential.

4.3 Credential Update and Verification Protocol

After certain number log entries were authenticated, credentials need to be updated and verified to continue securing log entries and to achieve forward integrity. Authentication credential update is done for every c log entries. As explained above and analyzed in the security analysis, the drawback of higher value of c is that the attacker can forge more log entries in the past.

The corresponding code is shown in lines 23–27. Whenever c log entries have been added to Q, a signing limit is reached and a credential update is performed in line 24. Since we use asymmetric cryptography, the initial credential is a public/private key pair and the new credential is a new (random) public/private key pair, the new value overwriting the old value (line 27). To validate the new credential, the new public key is also added to Q

and signed by the old credential. To verify a log entry or block of log entries, the verifier consequently needs the complete set of previous log entries and has to verify the sequence of signatures using the initial credential A as the starting point. If the signature check succeeds, then the log entry is accepted.

4.4 Truncation Resistance Protocol

To prevent truncation attacks, we use metronome events. This is shown in lines 15 and 16. Truncation resistance is measured in the number of clock ticks d defining truncation epochs. When d time has passed and no log entry from S reaches D, an artificial event is generated by S and a corresponding "dummy" log entry is added to Q. Metronome log entries [Holt 2006] are distinctive log entries generated periodically to show that the log is continuously receiving and accepting new log entries. If an attacker alters or truncates the log entry immediately prior to the time the entry is generated, then she also alters and truncates any metronome log entries arriving after that entry. Furthermore, the attacker must then stop any further recording of log entries as entries will not be successfully verified and alteration or truncation will be detected. If any metronome log entry is not present, then the latest verified entry specifies the first period at which the log entry is altered or truncated.

On the one hand, generating more metronome log entries in shorter time reduces the chance of an attacker to truncate many log entries from the log undetectable. On the other hand, less metronome log entries save bandwidth but extend the time in which an attacker can successfully perform a truncation attack. There are no restrictions on the timeout period d apart that it must not be 0.

For simplicity, we left number of log entries unlimited and so the sequence number i can also serve as a unique identifier for any entry in the log. If this is not required, then the value of i can be reset to 0 whenever buffers are sent to the log.

4.5 Enabling Fast Forward Verification

In contrast to the previous features that were already part of previous secure logging protocols, we now explain fast-forward log verification that to the best of our knowledge has not been part of any prior protocols.

The idea of efficient log verification is to add another chain of credentials E and signatures similar to what is used for authenticating log entries. Furthermore, we do not only sign the current hash chain value h (line 31) but also the credential A used for standard log authentication using E (line 33). This creates two (partly redundant) chains of signed public keys, one using A with covers a log entries and one using E that covers e log entries (see Figure 2). Due to E signing A, we can then use the chain E to "fast forward" through the chain of signatures using A. Obviously, the larger e is over a, the faster this mechanism works.

To also ensure forward integrity for signatures created with E, we regularly renew E as well (lines 36–39). As we argue later, the security guarantees by short-term credentials A are still maintained, but now additionally large steps using credentials E allow fast forward.

5 EXAMPLE

We now describe an example that illustrates the workings of the protocol. Consider Figure 3 where time flows from left to right and the log items that are triggered in the input queue of the logging device are shown at the top ($l_0, l_1, \ldots$). The initial credentials A and E are shown on the left, and the resulting contents of the output queue are shown at the bottom. Black numbers attached to values indicate the resulting position in the output queue for illustrative purposes. Arrows indicate flow of information.

The contents of the output queue are sent in batches of $b = 10$ entries to the log L.

In the example, the number of log entries per authenticator is set to $a = 2$. This means that after every two log entries an authenticator is computed using credential A and added to the output queue.

The number of log entries per new credential is set to $c = 4$, meaning that after every four log entries a new credential A is created, signed by the old credential and written to the log.

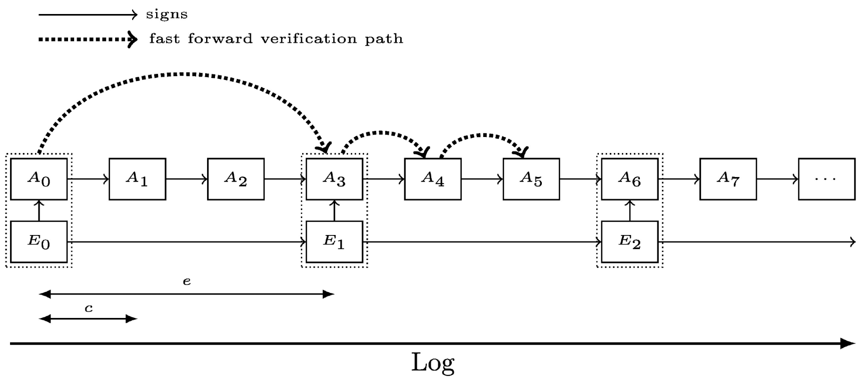

Fig. 2. Enablng fast-forward log verification.

Similarly, after $e = 9$ log entries, a new credential E is created, written to the log together with signatures using the old credential and the necessary variables to perform the fast forward. The old credential E is also used to sign the current public key of A to connect both signature chains during fast-forwards verification.

The contents of the output queue resemble the results of running the algorithm: When events happen on S, they are sent to D. An event can either be a regular event or the result of a timeout after d clock ticks, in which case a metronome log entry is sent. Both types of events generate a log entry l. For each l, the hash chain h is updated, i is incremented, and l is put to the output queue Q.

After two consecutive log entries, the authentication protocol is triggered. Hence, the current hash value h is signed using the secret key of A_0. The result Z is put to the output queue Q as well. After additional two log entries, the authentication protocol is triggered again ($i = 4$). Within this step, the key update protocol is triggered as well. As seen in Figure 3, a new credential A_1 and the respective signature $S_{A,1}$ is calculated using the previous (and initial) private key of A_0. The results are put into Q. The next two additional log entry blocks, the particular two additional authenticators Z_2 and Z_3, the next credential update A_2, and the respective signature $S_{A,2}$ are put to Q in the described order shown in Figure 3.

When the next log entry l_8 is processed, the hash value h is upgraded, i is incremented, and the update of E is triggered, resulting in a starting point for fast forward within the log file in the verification process. The following entries are put to Q:

(1) current h and respective signature S_{Eh}
(2) currently used $A.pk$ and respective signature S_{EA}
(3) current i and respective signature S_{Ei}
(4) new credentials E and respective signature S_E (signed with previous key)

After this, the last shown entry l_9 is added to Q, together with another authenticator Z_4, because the authentication protocol is triggered. Additionally, the buffer protocol is triggered as well. Therefore, the content of Q is sent to the log and Q is emptied.

6 VERIFICATION ALGORITHM AND SECURITY ANALYSIS

6.1 Verification Algorithm

Algorithm 3 shows the verification process performed by the verifier. Since we assume that the logging device continuously generates log entries, the verification algorithm is a non-terminating program that throws an exception once it observes a violation of integrity (indicated by return statements).

The initialization process is similar to LAVA itself. To verify, only the public key of $E_0.ltpk$ is needed. Fast forward is performed in a terminating loop in lines 11 to 17. Here the long-term credential signature chain is

ALGORITHM 2: Log Authentication and Verification Algorithm (LAVA).

```
parameters
    $a > 0, b > 0, e > 0$    ▹ measure of authentication, communication and verification efficiency
    $c > 0$    ▹ measure of forward integrity
    $d > 0$    ▹ metronome event interval
end parameters
variables
    $i \leftarrow 0$    ▹ natural sequence number of events and metronome events
    $h \leftarrow 0$    ▹ initial hash
    $Q$    ▹ queue for output data, initially empty
    $A$    ▹ initial key pair for forward integrity
    $E$    ▹ initial key pair for verification efficiency
end variables
actions
    upon event ⟨ $S$ triggers event $e$ or metronome event timeout $d$ is triggered ⟩ do
        $l \leftarrow$ log entry corresponding to $e$ or metronome event
        $Q$.put($l$)    ▹ add entry to output queue
        $h \leftarrow H(h||l)$    ▹ compute hash chain with hash function $H$
        $i \leftarrow i + 1$    ▹ increase sequence number
    end event
    upon event ⟨ $i$ mod $a = 0$ ⟩ do
        $Q$.put(Sign($A.sk, h$)    ▹ add authenticator for hash chain (termed $Z$ in text) to queue
    end event
    upon event ⟨ $i$ mod $c = 0$ ⟩ do
        $A' \leftarrow KGen$    ▹ compute new credential
        $Q$.put($A'.pk$)    ▹ add new public key to output queue
        $Q$.put(Sign($A.sk, A'.pk$)    ▹ add signature of new public key using previous private key to queue
        $A \leftarrow A'$    ▹ overwrite old credential
    end event
    upon event ⟨ $i$ mod $e = 0$ ⟩ do
        $Q$.put($h$)    ▹ add hash to output queue
        $Q$.put(Sign($E.sk, h$))    ▹ add authenticator to output queue
        $Q$.put($A.pk$)    ▹ add public key $A$ to queue
        $Q$.put(Sign($E.sk, A.pk$))    ▹ add signature of public key to output queue
        $Q$.put($i$)    ▹ add $i$ to output queue
        $Q$.put(Sign($E.sk, i$))    ▹ add signature of $i$ output queue
        $E' \leftarrow KGen$    ▹ compute new credential $E'$
        $Q$.put($E'.pk$)    ▹ add new public key to output queue
        $Q$.put(Sign($E.sk, E'.pk$))    ▹ add signature of new public key to output queue
        $E \leftarrow E'$    ▹ overwrite credential $E$
    end event
    upon event ⟨ $i$ mod $b = 0$ ⟩ do
        send content of $Q$ to $L$    ▹ send log messages to $L$ and empty queue
        $Q \leftarrow$ empty
    end event
end actions
```

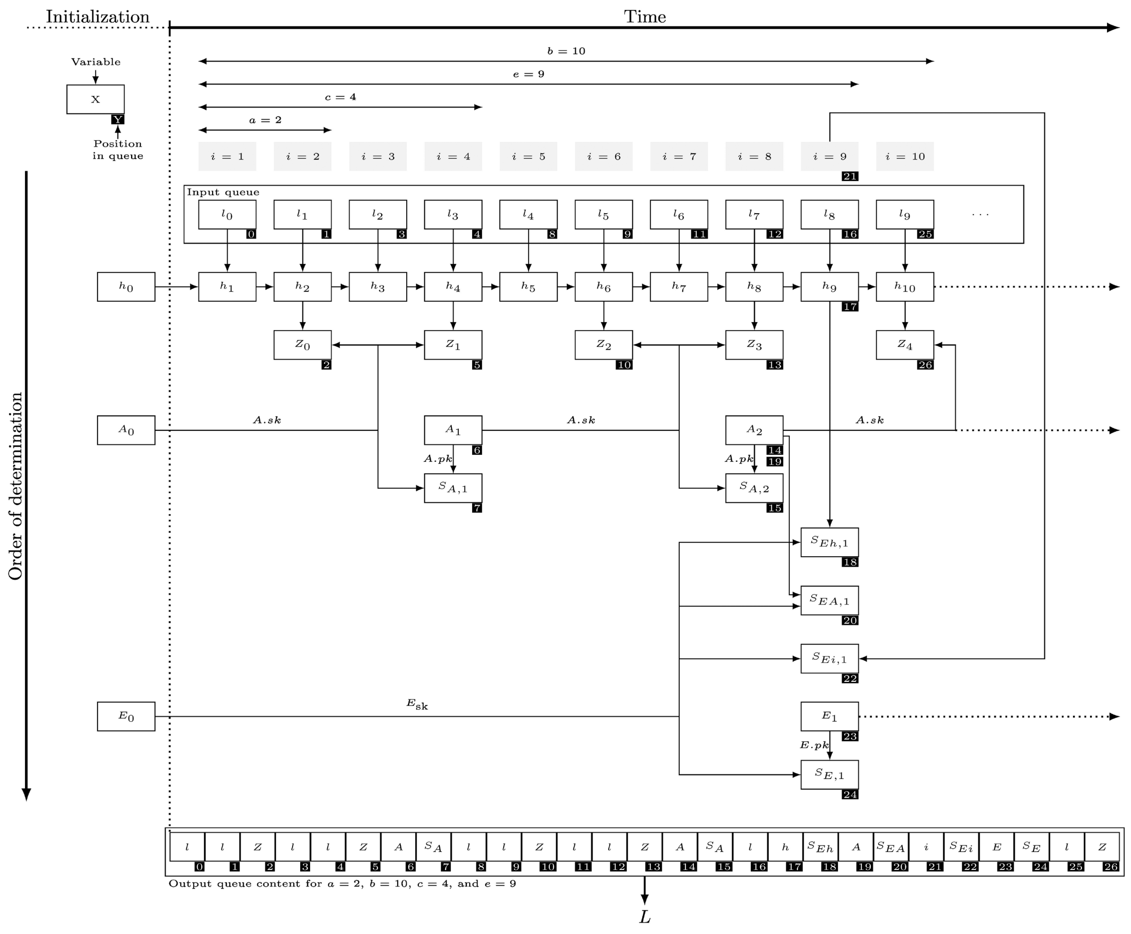

Fig. 3. Example parameter settings where $a = 2$, $b = 10$, $c = 4$, and $e = 9$.

checked. If any signature is not validated successfully, then the verification fails. After the long-term credential chain verification succeeds, the signature on the first *short-term* credential needs to be verified.

The main verification algorithm starts in line 24 as an infinite loop. In case of a normal log entry, the hash value h is updated, and its value is checked whenever an authenticator is expected (lines 26–30). In case a new public key for $A.pk$ is expected (line 31), the key is extracted and verified. If the verification fails, then the verifier returns false.

Checking the metronome events is performed using a timeout on the maximum delay that it may take for the next log entry to be available (line 40). In the worst case, i.e., if the adversary blocks all future log messages, then one metronome event every d ticks must happen, and after b ticks the next entry reaches the log. On the timeout $b \cdot d$ clock ticks, we add an additional value Δ, which is an upper bound on the time it takes for the logging device to send a packet of log entries to the log.

6.2 Security Analysis

If the log is very long, then verification of the log will also take a very long time. Therefore, we introduced the notion of verification efficiency where checking of e log entries can be skipped to "fast forward" through the

ALGORITHM 3: Verification Process returns false if log is not ok, takes as input a value k (number of log entry batches of size e to skip during fast forward).

```
parameters
    $a, b, c, d, e$                                   ▷ same parameters as used in LAVA
end parameters
variables
    $h \leftarrow 0$                                  ▷ initial hash
    $pk$                                              ▷ initial credential $A.pk$ for forward integrity as used in LAVA
    $ltpk \leftarrow \log.E_0.ltpk$                   ▷ initial credential $E.pk$ for fast forward as used in LAVA
end variables
actions
    upon event ⟨ on input $k$ ⟩ do
        for $i = 1$ to $\lfloor k/e \rfloor$ do       ▷ perform fast forward verification
            if Verify($ltpk, \log.E_i.\sigma, \log.E_i.ltpk$) then   ▷ verify signature for next long-term public key
                $ltpk \leftarrow \log.E_i.ltpk$        ▷ update long-term public key
            else
                return False                          ▷ credential authentication failure at position $i \cdot e$
            end if
        end for
        if Verify($ltpk, \log.A_{\lfloor k/e \rfloor \cdot e}.\sigma_{\text{E-to-A}}, \log.A_{\lfloor k/e \rfloor \cdot e}.pk$) then   ▷ verify signature for first credential
            $pk \leftarrow \log.A_{\lfloor k/e \rfloor \cdot e}.pk$   ▷ set first credential
            $h \leftarrow \log.E_i.hash$               ▷ get first value of hash chain
        else
            return False                              ▷ credential authentication failure at position $i \cdot e = \lfloor k/e \rfloor \cdot e$
        end if
        for $i = \lfloor k/e \rfloor \cdot e + 1$ to $\infty$ do   ▷ perform regular (non-terminating) verification
            $h \leftarrow H(h \| \log_i.entry)$        ▷ compute next element of hash chain based on next log entry
            if $i \mod a = 0$ then                    ▷ verify next authenticator
                if $\neg$Verify($pk, \log_i.\sigma, h$) then
                    return False                      ▷ verification failure of log entry $i$
                end if
            end if
            if $i \mod c = 0$ then                    ▷ verify next credential
                if Verify($pk, \log.A_i.\sigma, \log.A_i.pk$) then   ▷ verify signature of credential from log
                    $pk \leftarrow \log.A_i.pk$        ▷ update $A.pk$
                else
                    return False                      ▷ credential authentication failure at position $i$
                end if
            end if
        end for
    end event
    upon event ⟨ timeout after waiting longer than $b \cdot d + \Delta$ clock ticks for next log entry ⟩ do
        return False                                  ▷ verification timeout after position $i$
    end event
end actions
```

log. Unfortunately, as mentioned above, if log entries are skipped, then manipulations in skipped log entries will not be detected. We therefore need a special notion of verification security if we want to also have verification efficiency.

Definition 6.1 (Partial Verification Security). If a finite prefix of log entries is skipped, then the verification algorithm rejects if and only if the remaining suffix of the log is manipulated.

Proposition 6.2. *LAVA and its verification algorithm satisfy* Partial Verification Security *for weak attackers.*

Proof. The proof proceeds in two steps: First, we show by *correctness* of the underlying signature scheme that, if verification rejects, then the log must have been manipulated. Second, we show by *security* of the underlying signature scheme that every manipulation of the verified suffix leads to rejection of the verification algorithm.

Our verification procedure first verifies the fast-forwarding chain of successive verification keys and then, after fast forwarding, continuously proceeds with verifying the signature–log-entry pairs and the signatures of subsequent keys in the verification key chain of the verified suffix. This process perfectly inverts the original creation of this suffix's part of the log and, therefore, can only fail if (a) this part (or the fast-forwarding chain) was changed in the meantime or if (b) an honestly created signature is rejected by underlying verification algorithm. Since the latter would break correctness of the signature scheme, rejection of the verification algorithm implies that the log was indeed manipulated. This includes the case that the stream of log messages eventually stops (i.e., the adversary truncates the log and blocks any further write accesses by L to the log), which leads to a timeout that triggers rejection.

To show that manipulations of the verified log suffix always lead to rejection, note that the attacker never sees a signing key. For the purpose of a proof by contradiction, assume that the attacker is indeed able to undetectably manipulate at least one log entry in this suffix—again, this also includes the case that the adversary truncates the log or blocks logging. The reason that this manipulation is not detectable could either be that (a) the signature for this particular log entry was successfully forged or that (b) one of the signatures in the fast-forwarding or normal chain of verification keys was successfully forged. The latter would mean that the attacker planted an own adversarial, intermediate verification key in the chain. In either case, this means that the attacker crafted a valid signature for a signing key that it never saw. This forged signature breaks the unforgeability of the underlying signature scheme. Since we use a *secure* signature scheme, this contradiction shows that the assumption above (i.e., the attacker can undetectably manipulate a log entry) is false, which proves Proposition 6.4. □

Using the above result, it is easy to prove that LAVA and its verification algorithm also satisfy the following security properties regarding the correct verification of the *entire* log file from the beginning.

Definition 6.3 (Perpetual Verification Security). The verification algorithm rejects if and only if the log was manipulated.

Proposition 6.4. *LAVA and its verification algorithm satisfy* Perpetual Verification Security *for weak attackers.*

Instead of proving Proposition 6.4 directly, it will be evident that Proposition 6.4 is implied by the following generalized proof of Proposition 6.2. For this, we strengthen the attacker capabilities by considering *strong* attackers instead of weak ones. Obviously, for strong attackers partial verification security cannot be guaranteed, because a strong attacker at some point in time has access to the secret signing keys of E and A and from that point onwards can impersonate the logging device to replace, add, or remove log entries arbitrarily. Yet, if credentials are regularly updated, then the attacker is restricted in how many log entries she can manipulate "backwards" (toward the beginning of the log).

Proposition 6.5. *For strong attackers, LAVA and the verification algorithm satisfies* Perpetual *and* Partial Verification Security *until the last credential update before the time of attack.*

Proof. Consider the moment in which the attacker mounts the attack and corrupts the signing keys that are stored in the logging device's memory. We call all singing keys that were used until then but were *not stored* in the memory at that moment anymore (because they were replaced and updated already by newer signing keys) *safe*, and we call all keys that *are contained* in the memory when the attack begins *corrupted*.

We cannot hope for any guarantees of signatures created with corrupted signing keys. However, for safe signing keys, we are in the same situation as we were for Partial Verification Security under *weak* attackers: The attacker never sees these safe signing keys. As a result, any signature that verifies successfully for a safe signing key but that was not honestly created by the logging device breaks the unforgeability of the underlying signature scheme. This means that all signatures of verification keys in the fast-forwarding and normal chain as well as all signatures of log entries that verify under safe signing keys protect their *Perpetual* and *Partial Verification Security*. □

Since in practice, the time of attack is often not precisely known, the significance of Proposition 6.5 may seem limited, because acceptance by the verification algorithm does not necessarily mean that no manipulations have been performed. But in such cases, at least forward integrity is ensured—that means only a limited amount of log entries might be undetectably changed. Without updating the key material, the log is entirely meaningless in the presence of strong attackers.

7 IMPLEMENTATION

We developed an example implementation of LAVA by developing a LAVA class in Python that we make available to the community as open source software [LAVA project team 2023]. The implementation uses JSON objects to encode the type and value of a data entry in the log (see Figure 4). The log in this figure is based on the parameter settings of the example described above in Section 5.

To initialize the Python object, the parameters a, b, c, d, and e have to be set. The implementation uses a shared queue object for incoming log entries that the process of S generates. Additionally, the hash sum is initialized by hashing an empty string and i is set to 0. To private/public key pairs A and E have to be submitted to ensure forward integrity and verification efficiency. The buffer of the logger is implemented using another (not shared) queue object.

Initially, the LAVA class can append the public keys A and E together with the respective signatures. This achieves an easy transfer to the validation process, because only the public keys are necessary to perform this task. The output log shown in Figure 4 is an output of the developed implementation. The JSON objects are directly put into the output queue using the corresponding type key and a formatted string as data. The class checks frequently the input queue for new entries. If the queue is empty, then a counter is used to insert metronome log entries according to parameter d. In a real-world example, this metronome log entries are inserted into the queue by the process of S.

In Figure 4, the initial public keys A and E are set (line 1 and 4), each followed by the determined signatures (S_A and S_B). The log starts with one normal log entry, in this example a temperature T (line 5). The following log entry (line 6) is the first metronome log entry within this log. One can see that subsequent metronome log entries (e.g., in lines 16 and 17) have a difference of about 0.2 s. That is in accord to a parameter setting $d = 0.2$. All together, the log shows the 31 first entries. This corresponds to one buffer size b, consisting of 10 log entries and the determined entries to implement authentication, forward integrity, and verification efficiency.

If the input queue is not empty, then the hash is updated and the natural sequence number i is incremented. After each insertion, parameter i is evaluated based on the parameters a, c, and e in the stated order. For example, the event to determine an authenticator is based on the condition $i \bmod a = 0$. If a condition is true, then the respective method for the event is run. The determined values, e.g., a new public key A followed by the respective signature based on the previous key is put into the buffer queue. If the buffer condition $i \bmod b = 0$ is reached, then two targets are possible. On the one hand, the entries of the output queue can be written into a text file. This

```
{"type": "A", "data": "680583405989582946646l418[...]"}
{"type": "S_A", "data": "17c66503a6c03297d6c1df002[...]"}
{"type": "E", "data": "826902886765441751812946 8[...]"}
{"type": "S_E", "data": "73f88d5f09fa33c1bc4260d9c[...]"}
{"type": "l", "data": "T = 277.82 K"}
{"type": "l", "data": "2023-01-15 20:25:49.417568"}
{"type": "Z", "data": "4df014a8222b529e353391c29[...]"}
{"type": "l", "data": "2023-01-15 20:25:49.622204"}
{"type": "l", "data": "T = 277.99 K"}
{"type": "Z", "data": "25c0b12107ca4ad4561ae64c9[...]"}
{"type": "A", "data": "1075259300486095788402182[...]"}
{"type": "S_A", "data": "597afa02ceb00ebda8063d488[...]"}
{"type": "l", "data": "T = 274.34 K"}
{"type": "l", "data": "2023-01-15 20:25:49.822307"}
{"type": "Z", "data": "2dcca81b550a2d17cc77e82c2[...]"}
{"type": "l", "data": "2023-01-15 20:25:50.026775"}
{"type": "l", "data": "2023-01-15 20:25:50.227239"}
{"type": "Z", "data": "25f54f157e0e133ce8b5b9210[...]"}
{"type": "A", "data": "7201599955275506684111206[...]"}
{"type": "S_A", "data": "64b6e35311c334c8c641e6be7[...]"}
{"type": "l", "data": "T = 276.60 K"}
{"type": "h_E", "data": "030e07017da42ebdf6435f764[...]"}
{"type": "S_Eh", "data": "4322e6ba29ae277761b521e4d[...]"}
{"type": "A_E", "data": "7201599955275506684111206[...]"}
{"type": "S_EA", "data": "3a1d9c10709c8351a96ee3f6e[...]"}
{"type": "i_E", "data": "9"}
{"type": "S_Ei", "data": "87f5797238faf575d8fa157f9[...]"}
{"type": "E", "data": "6921231169284316845000998[...]"}
{"type": "S_E", "data": "8b749211172592e6a87f1eee9[...]"}
{"type": "l", "data": "2023-01-15 20:25:50.736327"}
{"type": "Z", "data": "0882ffd4f1993c5cd87f95f92[...]"}
                              ⋮
```

Fig. 4. Example log where $a = 2$, $b = 10$, $c = 4$, and $e = 9$.

is convenient if the log is not verified in real time. On the other hand, the entries can be put into an additional queue object that can be reached by the verification process. In this case, the produced log entries and the particular LAVA related messages can be verified in real time.

8 CONCLUSION

The initial motivation behind LAVA were the structural similarities of the published secure logging protocols [Freiling and Bajramovic 2018]. LAVA can therefore be regarded as a meta protocol that can be instantiated using the five parameters to result in (almost) all the major logging protocol. By setting any of the parameters a, b, c, d, and e to a very large value (∞) the specific functionality governed by that parameter is "switched off."

Structurally, LAVA is probably most similar to Holt's logcrypt system [Holt 2006], which is based on asymmetric cryptography. It is unclear whether LAVA can also be instantiated with symmetric cryptography. But LAVA allows us to extend the idea of logcrypt with communication efficiency. Similarly, the procotols by Schneier and Kelsey [1999] and Bellare and Yee [2003] do not feature truncation prevention that, however, is part of LAVA.

In future work, we plan to integrate secure logging of ICS security events into SIEM systems and thereby improve logging solutions implemented in existing OT landscapes within a bigger enterprise organization [Bajramovic 2019].

ACKNOWLEDGMENTS

We thank Stefan Kiltz for his support and guidance.

REFERENCES

Rafael Accorsi. 2008. *Automated Counterexample-driven Audits of Authentic System Records*. Ph.D. Dissertation. Universität Freiburg.

Rafael Accorsi. 2009. Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In *Proceedings of the 5th International Conference on IT Security Incident Management and IT Forensics (IMF'09)*. IEEE, 94–110.

Rafael Accorsi. 2010. BBox: A distributed secure log architecture. In *Proceedings of the 7th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI'10)*. Springer Berlin Heidelberg, 109–124. https://doi.org/10.1007/978-3-642-22633-5_8

Edita Bajramovic. 2019. *Secure Logging in Operational Instrumentation and Control Systems*. Ph.D. Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg.

Edita Bajramovic, Marius Frinken, and Felix Freiling. 2019. *LAVA: Log Authentication and Verification Algorithm*. Technical Report CS-2019-01. Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg. https://doi.org/10.25593/issn.2191-5008/CS-2019-01

Mihir Bellare and Bennet Yee. 1997. *Forward Integrity for Secure Audit Logs*. Volume 184, Technical Report. Computer Science and Engineering Department, University of California at San Diego.

Mihir Bellare and Bennet Yee. 2003. *Forward-security in Private-key Cryptography*. In *Topics in Cryptology.CT-RSA 2003: The Cryptographers. Track at the RSA Conference 2003 San Francisco, CA, USA, April 13.17, 2003 Proceedings*. Springer Berlin Heidelberg, 1–18. https://doi.org/10.1007/3-540-36563-X_1

Erik-Oliver Blass and Stephan Marvedel. 2020. Secure logging with syslog-ng: Tamper evidence and confidentiality. In *Proceedings of the Free and Open source Software Developers' European Meeting (FOSDEM'20)*.

Erik-Oliver Blass and Guevara Noubir. 2017. Secure logging with crash tolerance. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS'17)*. IEEE, 1–10.

Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer. Science & Business Media. https://doi.org/10.1007/978-3-642-15260-3

Ivan Bjerre Damgård. 1988. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, Berlin Heidelberg, 203–216.

Felix Freiling and Edita Bajramovic. 2018. Principles of secure logging for safekeeping digital evidence. In *Proceedings of the 11th International Conference on IT Security Incident Management & IT Forensics*, Harald Baier, Christian Keil, Klaus-Peter Kossakowski, and Holger Morgenstern (Eds.). IEEE, Los Alamitos, CA, 65–75.

Rainer Gerhards. 2009. *The Syslog Protocol*. Internet Request for Comments RFC 5424. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5424

Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing* 17, 2 (1988), 281–308.

Jason E. Holt. 2006. Logcrypt: Forward security and public verification for secure audit logs. In *Proceedings of the Australasian Workshops on Grid Computing and e-Research - Volume 54 (ACSW Frontiers'06)*. Australian Computer Society, Inc., Darlinghurst, Australia, 203–211.

Jiansen Huang, Hui Li, and Jiyang Zhang. 2018. Blockchain based log system. In *Proceedings of the IEEE International Conference on Big Data (Big Data'18)*. IEEE, 3033–3038.

John Kelsey, Jon Callas, and Alexander Clemm. 2010. *Signed Syslog Messages*. Internet Request for Comments RFC 5848. Internet Engineering Task Force.

LAVA project team. 2023. LAVA Code. Retrieved from https://faui1-gitlab.cs.fau.de/felix.freiling/lava-code

Seungho Lee, Wonsuk Choi, Hyo Jin Jo, and Dong Hoon Lee. 2019. How to securely record logs based on ARM trustzone. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*, Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang (Eds.). ACM, 664–666. https://doi.org/10.1145/3321705.3331001

Chris M. Lonvick. 2001. The BSD Syslog Protocol. RFC 3164. https://doi.org/10.17487/RFC3164

Di Ma and Gene Tsudik. 2009. A new approach to secure logging. *ACM Transactions on Storage (TOS)* 5, 1 (2009), 1–21.

Joachim Metz. 2016. Windows XML Event log (EVTX) Format. Retrieved from https://github.com/libyal/libevtx/blob/master/documentation/Windows%20XML%20Event%20Log%20(EVTX).asciidoc

Benedikt Putz, Florian Menges, and Günther Pernul. 2019. A secure and auditable logging infrastructure based on a permissioned blockchain. *Computers & Security* 87 (2019), 101602. https://doi.org/10.1016/j.cose.2019.101602

Bruce Schneier and John Kelsey. 1999. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)* 2, 2 (1999), 159–176.

Andreas Schuster. 2007. Introducing the microsoft vista event log file format. *Digital Investigation* 4 (2007), 65–72.

Trusted Computing Group. 2005. *TPM Main Specification*. Main Specification Version 1.2 rev. 85. Trusted Computing Group.

Received 2 June 2023; accepted 26 June 2023