Usable Privacy-aware Logging for Unstructured Log Entries

Christof Rath
Institute of Applied Information Processing and Communications
Graz University of Technology
Graz, Austria
Email: christof.rath@iaik.tugraz.at

***Abstract*—Log files are a basic building block of computer systems. They typically contain sensitive data, for example, information about the internal structure of a service and its users. Additionally, log records are usually unstructured in the sense that sensitive data will not occur in every entry and not always occur at defined positions within a record. To mitigate the threat of illicit access to log files, we propose a flexible framework for the creation of privacy-preserving log records. A crucial step is the annotation of sensitive data, by using arbitrary labels, during the development of a system. These labels are mapped to redaction filters to form a redaction policy. Thus, we can create two parallel log streams. One log stream contains fully redacted log entries. It, hence, does not contain any sensitive information and is intended for everyday use. The second stream contains the original entires. Here, confidentiality must be ensured. Our framework fosters *privacy by default* principles and can support selective disclosure of relevant data. We developed an implementation of our solution for `logback`, one of the major logging frameworks in Java, and successfully evaluated its applicability.**

Keywords-**Privacy-aware Logging; Privacy-preserving Auditing; Redacted Logfiles; Logging Framework; Usable Privacy**

I. Introduction

The paradigm shift towards distributed computing and especially cloud computing of recent years leads to an increased demand on verbose log data. First, this data is required for debugging purposes. This is owed to the increased complexity of distributed systems and the limited possibilities to remotely debug applications. Furthermore, it is required as evidence in the case of disputes between the acting parties, as a proof of the correct functionality of a system, or to increase transparency of data processing. The EU directive 1995/46/EC on Data Protection, for instance, recognizes transparency as a key privacy principle [1]. Although required, the accumulation of log data contradicts fundamental privacy and data protection goals: *data minimization* and a *defined purpose* for the processing of data. Often, log data is accumulated as preventive measure, which in most cases would not have been necessary. Since the actual use is unknown, it has to be collected in such quantities that hopefully all conceivable woes can be mitigated.

At the same time as the demand on verbose log data increases, also potential threats regarding data breach grow. In almost all cases do log files contain sensitive data. These are, for example, information about the internal structure of a service, process flows, or data about related services. But in many cases it also contains information about the users of a service. Even if a log file may not directly contain sensitive data of a user, like a password or social security number, it might still be possible to use these records for profiling purposes.

This lead to the design of a *privacy-aware logging framework for unstructured log entries*. It fosters *privacy by design* by annotating sensitive data during the development of a system and follows *privacy by default* principles. With a so-called redaction policy, it is possible to generate two different log streams. A privacy-friendly log stream, which does not contain any sensitive data. It can, thus, be stored unencrypted and is intended for everyday use, for example, for debugging purposes. The second log stream contains the original information. For this stream confidentiality has to be ensured. A possible use case could be to encrypt this log under the public key of a data protection agency. In case of disputes, it requires the consent, that is, the involvement of the data protection agency to decrypt the log files. Our framework supports the integration of new redaction mechanisms and log file encodings to support advanced and upcoming cryptographic schemes.

II. Related Work

One of the first works regarding secure logging is from 1998 by Schneier and Kelsey [2]. They proposed cryptographic support to create secure log files on untrusted machines. In their scheme all log entries are encrypted using a unique symmetric encryption key. A cryptographic hash function is used to derive a new encryption key for each log entry. Furthermore, a hash chain as well as message authentication codes (MAC) are used to ensure the integrity of the log file. The main shortcoming of this scheme is the missing search capability, when used for debugging purposes. To search for a specific incidence, the complete, or at least a substantial part of the log file has to be decrypted. Once the relevant log entries are found it is, however, possible to selectively communicate the corresponding keys to allow access only to those entries.

In 2004, Waters et al. extended the concept of secure logging by a searchable encryption scheme [3]. They proposed two different schemes for the encryption of keywords: Their first approach is based on symmetric encryption. To read the log entries that contain a certain keyword, the auditor has to obtain a search capability for that keyword from an audit escrow agent, a trusted third party. A disadvantage of this scheme is a shared secret key between the servers that create the log entries and the audit escrow agent. If that key gets compromised, an adversary cannot only create new log entires but can also read all previous entries for which keywords are known or can be guessed. To mitigate this threat, Waters et al. proposed an asymmetric version of their scheme. They use an identity-based encryption scheme to encrypt an ephemeral key by using the set of keywords as public keys. To decrypt an entry the auditor has to obtain the private key that corresponds to a certain keyword, which only the audit escrow agent is able to compute. However, the drawback of this solution is the number of computationally expensive pairing and exponentiation operations.

With their work from 2008, Wouters et al. published a secure and privacy-friendly logging framework for eGovernment services [4]. Using their scheme, a citizen can reconstruct a trail of log events in a privacy-friendly manner. In this context, that means that only the authorized subject, that is, the citizen, can link the different log entries related to one specific process. Furthermore, their scheme allows logging servers to show that they have behaved in accordance to a certain logging policy.

In 2013, Pulls et al. published an article about distributed privacy-preserving transparency logging [5]. In their work, they present a cryptographic scheme that enables data processors to inform users about the actual data processing that takes place on their personal data. Additionally, they claim to be the first to formalize the required security and privacy properties. Using their scheme, it is possible that multiple data controllers process data disclosed by a user. Each of the data processors can log the usage of the data to an arbitrary number of log servers. The user then can query those log servers to retrieve a complete trail of the usage of the data.

In previous work, the protection of privacy goals was always achieved by encrypting the entire log entry. While this obviously fulfils the goal, it also poses two drawbacks that we are tackling in this work. First, the usage of fully encrypted logs imposes a threat on its availability. Second, not all data in a log file is privacy critical. As a result of full encryption, the complete log file has to be decrypted even to trace some incidence that might in the end not be related to sensitive data. Additionally, this also renders use cases infeasible where the private key is deposited at an external entity, like a data protection agency.

In the later works by Wouters et al. and Pulls et al. access to the log entries was limited to the data subject. This achieves the highest possible level of privacy. However, we believe that this condition is too restrictive for a practical use. The reasons to collect log data can be split into three basic categories: debugging, auditing and transparency. But these categories are not necessarily independent. Data collected for auditing may be useful for debugging, albeit probably not as verbose as if collected specifically for debugging. Likewise, data collected for transparency can certainly be part of an audit to prove the lawful usage of personal data. Thus, if a data processor has no access to the transparency logs, as in the works by Wouters et al. and Pulls et al., the same information has to be collected again as part of an audit log. More important, using different logging facilities for different log purposes increase the complexity of a system and add an additional source of failure. We, therefore, propose a flexible and extendable framework that can be used for arbitrary log purposes by specifying different policies for particular requirements.

III. Requirements

The conducted survey on existing privacy-preserving logging schemes has identified a lack of dynamically adaptable and practical solutions that can easily be integrated in existing software and new solutions. To overcome this issue, we propose a privacy-aware logging framework that can easily be adapted to specific requirements. We have designed the proposed solution, which will be introduced in Section IV in detail, according to a set of requirements. These requirements have been extracted from an analysis of existing solutions and from published evaluations of these solutions such as [3], [5]. The derived requirements are discussed in the following in more detail.

R1: Forward Integrity
For any secure logging framework it is paramount that the integrity of logged data is ensured. It must be guaranteed, that previous entries cannot be deleted or modified and that new entries can only be appended at the end of the log file. Even though such modifications can usually not be prevented, it must always be possible to detect such changes.

R2: Confidentiality
To provide a secure and privacy-aware logging scheme it is necessary that only a limited group of persons can read sensitive data. This group should include the data subject and eligible internal and external auditors.

R3: Selective Disclosure
For debugging, or the cause of an audit, the complete information of a log file is not always necessary. For example, one might look at the actions of a particular user, then the identities and actions of other users do not need to be disclosed. Thus, it should be possible to create derived, authentic audit logs that contain only the necessary information without compromising integrity (Req. 1).

R4: Fully Redacted Logs
As a means of a first debug/support vector, it should be possible to create derived logs that do not contain any sensitive data. Since no sensitive data is contained, these logs can be stored and distributed more freely, for example, to the developers of a system.

R5: Privacy by Design
A practical solution should foster *privacy by design* principles [6]. That is, using this solution should enable the development of systems that adhere to these principles already in the design and development phase.

R6: Privacy by Default

While *privacy by design* covers the design and development of a system, *privacy by default* deals with the runtime of such systems. A system using our scheme should by default not leak any sensitiv information. Even if the logging framework is not configured to support our solution and falls back to some default configuration must no sensitive information be leaked.

IV. Proposed Solution

In recent years, generic logging frameworks evolved as independent projects and libraries. These frameworks allow the segregation of program logic and logging-related tasks, for example, which log entry has to be stored to what medium, or the filtering according to a certain log level. Later, a new abstraction layer, *logging facades*, was introduced to further decouple the software from the underlying logging framework.

A. Basic Functionality

The basic building blocks of a logging framework usually consist of a set of loggers and a set of appenders, as shown in Figure 1. The logger objects provide the interface to issue new logging incidences and the appenders take care of persisting or presenting the resulting log message according to their implementation. Certain types of appender additionally use encoders to convert a log event e to its binary representation. The creation of the log message is sometimes computationally

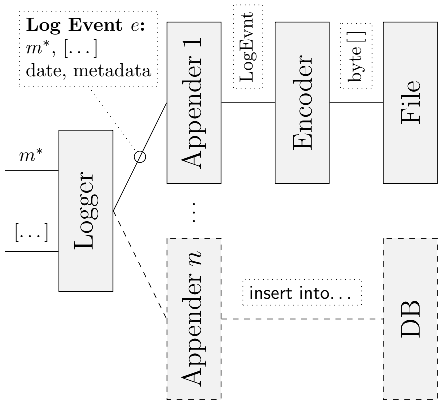
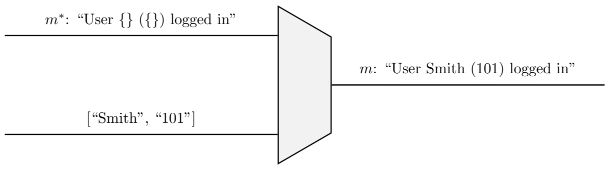


Fig. 1. Building Blocks of a Logger Framework

expensive, for instance, if it requires marshalling of large XML structures, or database queries. Typically, the level of verbosity also reflects the length and complexity of a log message; however, it is less likely that messages with a high level of verbosity will be presented or persisted at all in a productive environment. Consequently, it became common practice to check if a certain log level is enabled before a log message m is created:

Modern logging frameworks, thus, support a substitution mechanism, visualized in Figure 2. The log message consists of some fixed strings s_x and can be augmented by placeholders p, for example, "{}", yielding a log message template m^*.

$$m^* := s_0||p||s_1||p\dots$$



m^*: "User {} ({}) logged in"
["Smith", "101"]
m: "User Smith (101) logged in"

Fig. 2. Basic Logging Functionality

Additionally, a set of object $[o_0, o_1, \dots, o_n]$ can be passed to the `log(...)` function or its derivatives:

```
ComplexObject co = new ComplexObject();
LOG.debug("Log this {}", co);
```

Only if the `debug` level is enabled in the current configuration, the placeholders will be replaced by the string representation of the corresponding elements in the provided object array, resulting again in the log message m:

$$m := s_0||\,\mathrm{str}(o_0)||s_1||\,\mathrm{str}(o_1)\dots$$

In the above equation, $\mathrm{str}(\dots)$ denotes the function that converts an object to its string representation.

B. Annotation of Privacy-relevant Data

We extended this placeholder concept to annotate sensitive data. For that, each placeholder gets a label, for example, {`userID`}. We will show later how these labels are connected to, so-called, redactors to allow for a policy-based configuration. By labeling the placeholders we reach multiple goals:

- Specifying already during development which data is sensitive supports the requirement *privacy by design* (Req. 5).
- Since this is a non-standard syntax we already fulfil the requirement *privacy by default* (Req. 6). For instance, given the following code:

  ```
  LOG.debug("User {userID} has logged
  in", user.getUserID());
  ```

  The underlying logging framework does not recognize {`userID`} as a placeholder. Thus, if the logging framework is not configured to support our architecture, the resulting log message will be: `"User {userID} has logged in"` and the provided object `user.getUserID()` will be ignored.
- Finally, by annotating the data, that is, giving the placeholders a semantic, we can provide context-specific substitutions. The user ID, for example, might be replaced by a pseudonym for allowing the identification of related entries without revealing the identity of the user. Other attributes might be irrelevant for the everyday use of the log file and, hence, blanked out entirely.

We want to note that there is one flaw in our *privacy by default* solution. When mixing labeled and unlabeled placeholders, private data might be disclosed unintentionally: Consider a case where the message template contains first labeled and then unlabeled placeholders, for example, `"User {userID} has logged in {}"`. If this template is processed by a logger context that has not been configured to support our framework, unintentional information leakage will occur. In this situation the labeled placeholders are not recognized and will be ignored. The unlabeled placeholders will, consequently, be replaced by the first values of the object array. For example, given the message template from above and the object array `["101", "successfully"]` would result in the log message `"User {userID} has logged in 101"` instead of `"User ***** has logged in successfully"`. We propose two solutions for that: First, one can ensure that unlabeled placeholders are used only before labeled placeholders in all message templates. While this works, we would not recommend this approach, especially when working in large development teams, since it is hard to ensure that this requirement is fulfilled over a long period of time. Hence, our recommendation is to eliminate all unlabeled placeholders and instead explicitly label a placeholder to be substituted by *non-critical* data, for example, `"User {userID} has logged in {non-critical}"`. This label can then be connected to a 'pass-through' redactor. This solution has the practical advantage that unlabeled placeholders can be spotted easily even in a very large code base and if desired automatically replaced by a default label and, thus, fulfill Req. 6.

C. Redactors

In our architecture, we use redactors to replace a private information with its redacted representation. A redactor gets as input some private data d and outputs a redacted value d'.

$$d' := \mathrm{R}(d)$$

Trivially, the output of such a filter might be just a series of asterisk characters. Other examples are pure random characters,

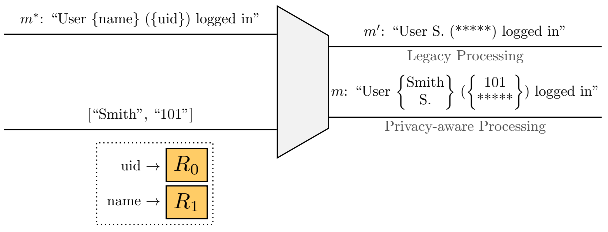

m^*: "User {name} ({uid}) logged in"
m': "User S. (*****) logged in"
Legacy Processing
m: "User $\begin{Bmatrix}\text{Smith}\\\text{S.}\end{Bmatrix}$ ($\begin{Bmatrix}101\\{*}{*}{*}{*}{*}\end{Bmatrix}$) logged in"
Privacy-aware Processing
["Smith", "101"]
uid → R_0
name → R_1

Fig. 3. Privacy-aware Logging Functionality

or the output of a random oracle.

By mapping the labels of the placeholders to redactors a desired redaction policy can be defined. This leads to the privacy-aware logger functionality shown in Figure 3.

V. Implementation

We implemented our framework in Java to demonstrate the applicability of our solution. The source code is available on github [7]. In this section we will first describe our implementation and then present the results of our performance tests using different redactor and encoder configurations.

The implementation of our solution is integrated in the logging framework `logback` [8], which is a native implementation of the simple logging framework for Java (`slf4j`). The integration of our solution happens via our own implementation of the `LoggerEvent` class. To instantiate the `PrivacyAwareLoggerEvent` either a proxy-appender can be used or the global filter mechanism of modern logging frameworks. The latter has the advantage to provide the modified logger event to all configured appenders and for lack of space we will only describe this approach.

A. Privacy-aware Turbo Filter

In the `logback` terminology global filters are called *turbo filters*. The privacy-aware turbo filter (PATF) implementation

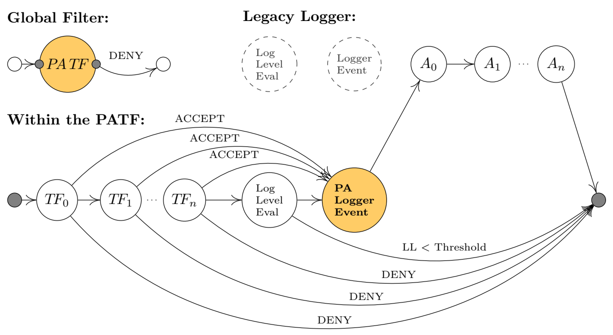


Fig. 4. Processing of Logging Incidences using the PATF. TF_x is a global filter instance (called turbo filter in the terminology of `logback`) and A_x an appender of the current logger. The highlighted circles show objects where the redactor mapping is used.

mimics the behaviour of the legacy logger. The necessary steps are: Execution of a *nested* turbo filter list ($TF_0 \dots TF_n$), evaluation of the log level, instantiation of a privacy-aware logging event object e' and processing of the list of appenders ($A_0 \dots A_n$) of the calling logger instance. The only difference between the legacy logger and our filter is that we instantiate a *privacy-aware logger event* object. The reason for the nested list of turbo filters is as follows: The return value of each filter may break the filter chain by returning either `ACCEPT` or `DENY`, as shown in Figure 4. Hence, if the PATF is not the first filter in the chain and any previous filter returned `ACCEPT` or `DENY`, the PATF does not get executed. The PATF, on the other hand, already calls the appenders of the issuing logger instance, as illustrated in Figure 4. To avoid doubled log entries for the same logging incidence, it is necessary that the further execution of the logging incidence is aborted by returning `DENY`. Consequently, this disables all filters that might have been registered after the PATF. Our solution tackles this by providing a nested list of turbo filters. When configured correctly, our filter is the only turbo filter registered directly at the logger context. This ensures that the PATF gets always executed and that the legacy program flow is aborted after the execution of the PATF, shown in Figure 4. All other turbo filters shall be registered at the PATF. With this setup, the exact same behaviour as in the legacy code is guaranteed, only now using a privacy-aware logger event object.

The rest of the framework components consists of the alternative proxy-appender, the `PrivacyAwareLoggerEvent` class and the interface of the redactors. The member function `getFormattedMessage()`, which performs the actual conversion of the message template m^* and the provided object array $[o_0 \dots o_n]$ into the formatted message m, has been overwritten to return the redacted version m' of m and a new method has been defined to return m. Consequently, only components which expect privacy-aware logger events have access to the fully disclosed data.

B. Redactors

The most trivial redactor is an identity redactor. It passes the input unmodified to the output.

Next, we implemented a blinding redactor. The output of the blinding redactor depends on its configuration. This can be a static string, for example, `*****`, or the label of the placeholder (`[userID]`). By specifying the algorithm of a cryptographic hash function, a message digest can be used to offer pseudonymization.

A variant of the blinding redactor is the anonymizing redactor. Here, not only the sanitized value gets redacted but also the original value, effectively destroying all links to the original data, even in the privacy-critical data path.

Finally, we developed two encryption-based redactors. The output of these redactors is the Base64 encoded cipher text of the input.

The main advantage of these redactors is the possibility to selectively disclose (Req. 3) any single value.

C. Encryption Encoders

So far we have discussed two means to fulfil the confidentiality requirement (Req. 2). First, we discussed the weak notion of solving it on a organizational level, that is, by persisting the data on a trusted device. Then we presented a cryptographically secure solution by using the encrypting redactors. Additionally it is possible to encrypt the complete log stream. For that we developed two different *wrapping* encoders. These encoder use a nested sub-encoder for the actual encoding of the logging entry but provide a cipher output stream to transparently encrypt the data. Our initial implementation uses the Cryptographic Message Syntax (CMS) scheme to encrypt the log records. Multiple recipients can be configured that are then able to decrypt the complete log file. However, our tests showed that the resulting log files may be corrupted if the output stream is not closed correctly before termination. This is a great risk for a logging system since exactly situations of unexpected program terminations are subject to consult the log files. Additionally, we have implemented a low-level implementation that uses a configurable symmetric cipher algorithm to encrypt the log data. A random secret key and initialization vector is generated, if not provided via the configuration. It is possible to provide an RSA certificate to instantiate a KEM-DEM scheme. With this encoder only the last entry may be lost if the program is terminated without closing the cipher output stream. With our implementation it is furthermore possible to set a second output stream to separate the key stream and the log data stream.

VI. Evaluation

Using our implementation, we conducted a performance evaluation. For that we measured the execution time of a test program using different logging configurations. Each test run issued 40,000 logging incidences in four parallel threads. All calls have been made using the same message template with four placeholders and always the same parameters. For each configuration two executions with five test runs each have been performed to mitigate the influence of the virtual machine, other running processes and the operating system. The median value of all execution times of a certain configuration was used in the following table. The integration of our framework was done using the privacy-aware turbo filter. All tests have been performed on an Apple MacBook Air with a 1,7 GHz Intel Core i7 processor (dual-core), 8GB RAM and a 250GB SSD.

A. Baseline Tests

To compare our results to the status quo, we first conducted a baseline test using only the standard `logback` components. Additionally, we used the same configuration also to test the performance impact of our digest converter. The converter was configured to print Base64 encoded values of a chain of hashes, that is, the hash value of the previous record is part of the input of the current message digest.

TABLE I
EXECUTION TIMES OF THE BASELINE TESTS

	Exec Time [ms]		
Test	Med.	Mean	σ
Baseline	697	771	222
SHA-256 Digest	785	924	302

In Table I, it can be seen, that the digest converter added roughly 13% overhead.

B. Redactor Tests

After the baseline test, we tested our five redactor implementations separately. We configured a single file-appender, which used a pattern layout encoder to generate the log records. The layout we used only printed the relative time, the name of the thread and the log message. We also wanted to evaluate the impact of our redactors when used multiple times per log record. Thus, we ran our test once by redacting only two parameters (Table II), and then again by redacting all four parameters (Table III).

TABLE II
EXECUTION TIMES OF REDACTORS (2 PARAMETERS)

	Exec Time [ms]		
Redactor	Med.	Mean	σ
Identity	649	702	138
Blind	576	617	116
Anonymize	736	851	278
AES Encrypt	721	797	180

TABLE III
EXECUTION TIMES OF REDACTORS (4 PARAMETERS)

Redactor	Exec Time [ms] Med.	Mean	σ
Identity	596	645	113
Blind	582	628	106
Anonymize	825	849	170
AES Encrypt	875	927	236

An interesting result of these test set can be seen in the first rows of Tables II and III: The identity redactor showed consistently better results when processing all four parameters. The blinding and anonymizing redactors can be configured in several ways, which will influence the execution time. We decided to use a static text ($*****$) for the blinding redactor and a SHA–1 hash-based anonymization. The AES encryption redactor was configured as KEM-DEM instance, that is, only an encryption certificate was configured. An ephemeral symmetric key and initialization vector was generated and RSA encrypted during the initialization.

C. Encoder Tests

Finally, we also tested our encoder implementations. For these test runs we used always the same appender configuration, which was a single file-appender, a pattern layout encoder and the four parameters redacted by the identity, blinding, anonymizing and AES-encryption redactors. For the first test run we used only the pattern layout encoder, as the baseline result. Subsequently, the pattern layout encoder was additionally wrapped by one of the *privacy-aware* wrapping encoders.

TABLE IV
EXECUTION TIMES OF PRIVACY-AWARE ENCODERS

Encoder	Exec Time [ms] Med.	Mean	σ
Plaintext	919	989	203
AES Encrypt	1,153	1,168	179
CMS Encrypt	1,036	1,221	355

The results in Table IV show that the encryption of the output requires at least 25% more time. However, the crypto library we used performed the AES encryption in software only. A significant performance gain can be achieved by using the main processors AES instruction set or by using dedicated crypto hardware like an HSM.

VII. CONCLUSION

In this article, we proposed, presented and discussed a practical solution to generate unstructured privacy-preserving log entries. Based on a set of relevant requirements, we have developed an appropriate framework for the proposed solution. We have then carried our proposed solution over to a concrete implementation as an extension of the popular Java logging framework `logback` [8]. Our architecture uses a new syntax to annotate privacy-critical data, thus, supporting *privacy by design* (Req. 5) and *privacy by default* (Req. 6) as a key principles. With the support of global filters, a feature already provided by `logback` and other logging frameworks, it is very easy to integrate our framework in existing projects. The requirements Req. 2 (confidentiality) and Req. 3 (selective disclosure) can be fulfilled either on a per-label basis by using encrypting redactors or on file-basis by using encrypting encoders. The source code of our implementation is publicly available on github.com [7].

Even though our framework is ready for productive use, there are still some open issues that are regarded as future work. The output of the fully disclosed log stream is currently only implemented for stream-based appenders, like console or file appenders. The means to process the fully disclosed records by alternative appenders, e.g., the syslog appender, is currently missing. However, our preliminary research shows that the framework is flexible enough to support also these use cases. Our solution shows, already in its current form, substantial improvements over the state of the art regarding privacy-aware logging of unstructured log entries. Especially the creation of fully redacted log files, alongside the original log data, offers a tremendous advantage over the state of the art, for the confidentiality of sensitive data in log files, for everyday use. Finally, this work is not intended as a direct replacement of related work but should also be seen as enabler to integrate sophisticated logging schemes into modern logging frameworks.

REFERENCES

[1] European Parliament, "Directive 95/46/EC," in *Official Journal of the European Communities*. European Commision, 1995, vol. L 281/31.

[2] B. Schneier and J. Kelsey, "Cryptographic Support for Secure Logs on Untrusted Machines," *Idea*, pp. 53–62, 1998.

[3] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, "Building an Encrypted and Searchable Audit Log," in *In The 11th Annual Network and Distributed System Security Symposium*, 2004.

[4] K. Wouters, K. Simoens, D. Lathouwers, and B. Preneel, "Secure and privacy-friendly logging for eGovernment services," in *ARES 2008 - 3rd International Conference on Availability, Security, and Reliability, Proceedings*, 2008, pp. 1091–1096.

[5] T. Pulls, R. Peeters, and K. Wouters, "Distributed privacy-preserving transparency logging," *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society - WPES '13*, no. 1, pp. 83–94, 2013.

[6] P. Schaar, "Privacy by Design," *Identity in the Information Society*, vol. 3, no. 2, pp. 267–274, Aug. 2010.

[7] Privacy-aware Logging Framework Source Code. visited on: 2016-03-18. [Online]. Available: https://github.com/nobecutan/privacy-aware-logging

[8] Logback Project. visited on: 2016-03-18. [Online]. Available: http://logback.qos.ch