Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1567–1574

FAL: A Forensics Aware Language for Secure Logging

Shams Zawoad
zawoad@cis.uab.edu
University of Alabama at Birmingham

Marjan Mernik
marjan.mernik@uni-mb.si
University of Maribor

Ragib Hasan
ragib@cis.uab.edu
University of Alabama at Birmingham

***Abstract*—Trustworthy system logs and application logs are crucial for digital forensics. Researchers have proposed different security mechanisms to ensure the integrity and confidentiality of logs. However, applying current secure logging schemes on heterogeneous formats of logs is tedious. Here, we propose FAL, a domain-specific language (DSL) through which we can apply a secure logging mechanism on any format of logs. Using FAL, we can define log structure, which represents the format of logs and ensures the security properties of a chosen secure logging scheme. This log structure can be later used by FAL to serve two purposes: it can be used to store system logs securely, and it will help application developers for secure application logging by generating required source code.**

***Keywords*-DSL, Secure Logging, Audit Trail, Digital Forensics**

I. Introduction

IN RECENT years, digital crime case has increased tremendously. An annual report of the Federal Bureau of Investigation (FBI) states that the size of average digital forensic case is growing 35% per year in the United States. From 2003 to 2007, it increased from 83 GB to 277 GB [1]. Various logs, e.g., network log, process log, file access logs, audit trail of application play vital role in a successful digital forensics investigation. System and application logs record crucial events, such as, user activity, program execution status, system resource usage, network usage, and data changes through which some important attacks can be identified, e.g., network intrusion, malicious software, unauthorized access to software, and many more. Log is also important to ensure the auditability of a system and auditability is a vital issue to make a system compliant with the regulatory acts, e.g., Sarbanes-Oxley (SOX) [2] or The Health Insurance Portability and Accountability Act (HIPAA) [3]. Keeping system audit trails and reviewing them in a consistent manner is recommended by NIST as one of the good principles and practices for securing computer systems [4].

While the necessity of logs and application audit trail are indisputable, the trustworthiness of this evidence will remain questionable if we do not take proper measures to secure them. In many real-world applications, sensitive information is kept in log files on an untrusted machine. As logs are crucial for identifying an attacker, attackers often attack the logging system to hide the trace of their presence in the attack or to frame an honest user. Very often, experienced attackers first attack the logging system [5], [6]. Malicious insider users colluding with the attacker can also tamper with logs. Moreover, forensics investigators can also alter evidence before presenting to court. To protect logs from these possible attacks, we must need a secure logging mechanism. Researchers have already proposed several secure logging schemes [7]–[9], which are designed to defend such attacks.

However, ensuring the privacy and integrity of the logs is costly given that it requires special knowledge and skill of developers. To implement a secure logging scheme, we need to give complete access of the logs to application developers. Providing full access of sensitive logs to developers definitely increases the attack surface. They can violate the privacy, sell sensitive business or personal information, and most importantly can keep a back door for future attack. Adding secure application audit trail can also be burdensome for developers, and increases the application development cost. On the other hand, system admins, who have access to network logs, process logs may not have sufficient knowledge for developing a securing logging scheme.

In this paper, we propose a DSL [10] to assist system admins and application developers for maintaining system logs and application audit trail securely, which is crucial for digital forensics investigation. A DSL is designed for a particular domain and has great advantages over general-purpose language for that specific domain. DSLs provide higher productivity by its greater expressive power, the ease of use, easier verification and optimization [10]–[12]. Using our proposed DSL *FAL*, system admins can define log structure and parse a log file according to the structure. They can also define the security parameters to preserve the integrity and confidentiality of logs. To accomplish this, they only need their domain knowledge related with system logs. Using FAL, a software security analyst can define the required audit trail structure and can generate code for a generic purpose language (GPL), e.g., Java, C# to store the audit logs securely.

Contribution. The contribution of this work is two-fold:

- We propose the first domain-specific language FAL, which can be used to ensure the security of system logs, and application audit logs.
- We show all the DSL development processes, which can be served as a guideline for future DSL development.

978-1-4673-4471-5/$25.00

II. Background and Motivation

In this section, we present the necessity of secure logging scheme, common approaches for secure logging, and how a DSL can help to mitigate some challenges of secure logging.

A. Secure Logging

As logs are crucial for digital forensics investigation, this is often become the target of attacker. There can be two types of attacks on logs:

- **Integrity**: Integrity of logs can be violated in three ways – an attacker can remove log information, can re-order the log entries, and can add fake logs. A malicious user can launch these attacks to hide the trace of illegal activities from forensics investigation, or to frame an honest user. Timing of an incident is crucial for forensics investigation. Hence, re-ordering the log entries can be important for an attacker, which can give him a chance to produce some alibi.
- **Confidentiality**: From various system logs and application logs, we can identify the activity of users as well as sensitive private information about the users. From the application logs of a business organization, we can also trace out very sensitive business information. This information has high value to attacker. Hence, attack on the confidentiality of logs can be highly beneficial to attacker.

The above attacks can come from different types of attackers:

- **External Attackers**: An external attacker can be a malicious user intending to attack users' privacy from the logs, or try to modify logs to hide the trace of any attack (e.g., network intrusion, malware, spyware). A dishonest forensic investigator can also be an external attacker, as the investigator can alter the logs before presenting to court.
- **Internal Attackers**: A more crucial attack can come from insider attackers colluding with malicious users. A dishonest insider can be a system admin, database admin, or application developer. As system admins have access to all system logs, they can always tamper with logs. Application logs and some of the system logs can be stored in database. In this case, threats can come from database admin. A malicious database admin can modify logs without leaving any trace of the modification. Application developers can modify application logs, or can create a backdoor to collect the application logs. Besides tampering the logs, these insiders can also attack on the privacy of users. They can collect and sell sensitive business and personal information derived from the logs.

To defend the confidentiality and integrity of logs, researchers have proposed several secure logging schemes [7]–[9], [13]. The commonalities among these secure logging schemes are: encrypting sensitive fields to protect the confidentiality, and maintain a hash-chain of the logs to protect the integrity of logs. Hash-chain maintains the chronological information of data. Hence, if any log is missing from the chain or if there is a reordering of the logs then this alteration can be detected from the hash-chain. Hash-chain of one log entry is calculated using the hash of its previous entry. In this way, it preserves the sequence information.

B. Motivation

Though there are some proven secure logging schemes, developing and maintaining a scheme is always challenging because of the following reasons:

1) The first problem is logs are in heterogeneous format. Unfortunately, there is no standard of logs format. Hence, two types of systems logs can look completely different. Moreover, same log can vary by operating systems. For example, format of a process log entry is different in MacOS and Debian.
2) To build a secure logging scheme, we need to permit the logging scheme developers to access the logs. Developers' accessibility to crucial log information certainly increases the attack surface. Earlier, we only need to trust system admins; adding developers in the loop adds an extra level of trust. The developer might place a back door to collect plain log information and can violate the privacy of users.
3) For application logging, application developers need to add secure application logging code for every scenario. Most of the cases, we need to log the database operations – Add, Update, Delete. Through these logs, we can get who has done some specific operations on a specific data. Writing code for all of the possible scenarios is burdensome for developers, and skipping one important logging method may turn out to be crucial.

To resolve the above challenges, we suggest that a well-defined DSL should help. For system logs, with the help of a DSL, we can shift the responsibility of developing a secure logging scheme from programmers to system admins. Because systems admins already have the domain knowledge about system logs, and with the help of a DSL, they can easily define the required security parameters. In this way, we can minimize one level of attack surface. The DSL should also deal with the heterogeneous formats of logs. Hence, we do not need to re-implement a scheme when the log format changes because of any system migration. For application logs, a DSL can generate required application logging code to ease the life of application developers. However, using proprietary encryption and hashing algorithm cannot be adopted by a DSL. Hence, our proposed DSL can only handle established encryption and hashing algorithms.

III. The domain-specific-language FAL

A. Domain Analysis

The very first step of designing a DSL is the detailed analysis and structuring of the application domain [14], which is provided by domain analysis. Output of domain analysis is a Domain Model, which gives us commonalities and variabilities, semantics of concepts, and dependencies between properties. Among various schemes of domain analysis, we choose FODA

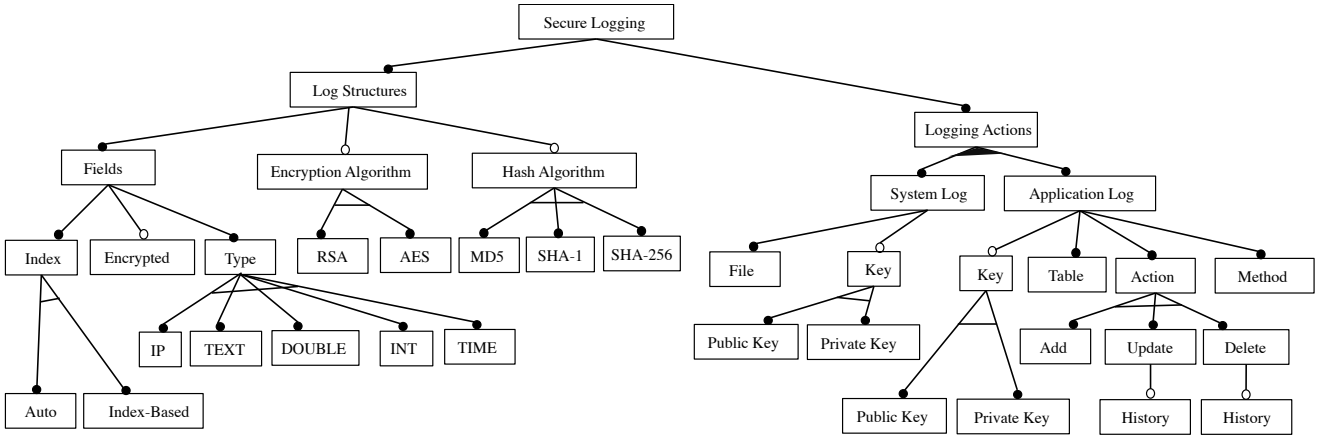

Fig. 1: The Feature Diagram of FAL

(Feature Oriented Domain Analysis). In FODA, the results of the domain analysis are obtained in a feature model [15]. One of the most prominent ways of describing feature model is by feature diagram (FD). The FD is represented as a tree with nodes as rectangles and arcs connecting the nodes. Nodes determine the features, while arcs determine the dependency between the features. The nodes can be mandatory or optional, which are denoted by closed dots, or open dots respectively. The FD of FAL is illustrated in Figure 1.

From Figure 1, it is clear that a secure logging scheme constitutes of log structure and logging action. Every log structure must have fields. Every field must have a type. According to the chosen secure logging scheme, a field can be encrypted or not. Fields may have an index attribute, which will be used to specify the location of a field in an input. The type of a field can be IP, Text, Double, Integer, or Time. Time can be auto-generated, i.e. current system time, or can be index-based. For index-based field, value will be extracted from input file or argument list according to the position defined by the index. For encryption, encryption algorithms, such as, RSA [16], AES [17] can be used. Some secure logging mechanisms use hashing, and hash-chain to ensure the integrity of the logs. Hence hashing algorithms, e.g., SHA-1[1], SHA-256[1], or MD5[2] can be used.

After defining a secure log structure, we need to use the structure for system or application logging. There can be two types of actions. First, for system logs, we need to parse the log files according to a pre-defined structure, and apply the security features while storing. Second, for application log, we need to generate GPL code. For system logs, we must have a file name, and we may have public or private key file. By encrypting with public key, we can ensure that only the private key owner can decrypt certain information. Private key is also needed to create a signature on certain data. Using the public key we can verify the signature. For application logging, we must have a table name, action, method, and may have public or private key file. Method is actually a method name of a GPL program, from where the action is called. An action can be adding a new record, update, or delete a record. For update and delete, we may want to save the history of previous records.

The FDs represent the common features, which always exist in a system (commonalities) and optional features, which may or may not exist in a system (variabilities). Some of the commonalities identified from the FD of FAL are *Fields*, *Type*, etc., and some variabilities are *Encryption Algorithm*, *Key*, etc. From FD, the variation points can be easily identified (optional, one-of and more-of features). After the domain analysis, we can gather the following information – terminology, concepts, and common and variable properties of concepts and their interdependencies.

B. The Abstract Syntax

After the domain analysis, the next step is to design the DSL, from which we will get syntax and semantics of the language. During the domain analysis using FODA, we identified several concepts in the application domain that needed to be mapped into DSL syntax and semantics. From the FD, we can notice the relationship between concepts/features in an application domain and non-terminals in a context-free grammar (CFG). Table I represents the mapping between application domain concepts and non-terminals in context-free grammars, which appears on the left hand side (LHS) and right-hand side (RHS) of CFG production.

Based on Table I, we define the abstract syntax of FAL, which is presented in Table II. The syntactic domains of variables are presented in Table III. A FAL program consists of Log structures LS, and logging actions LA. Log structure LS defines field description F and security parameter S. There can be one or more LS. The field description F specifies field type, id, index I, and encrypted status. There can be one or more fields in a log structure. Index I is either an integer number, or auto. Security parameter S defines encryption and hashing algorithm. Logging action LA can be either System logging action SLA or Application logging action ALA. There can be one or more logging actions. SLA specifies the system log file

[1]http://www.itl.nist.gov/fipspubs/fip180-1.htm
[2]http://tools.ietf.org/html/rfc1321

TABLE I: Translation of the application domain concepts to a context-free grammar

Application domain concepts	LHS non-terminal	RHS structure
Secure Logging	P	Description of Log structure, and logging action
Log Structure	LS	Description of fields and security parameters
Fields	F	Field id, type (IP, Text, Double, Integer, Time), indexing feature, encrypted (or not encrypted)
Index	I	Position of a field in input, or auto.
Security Parameters	S	Description of encryption and hashing algorithm
Logging Action	LA	Description of system logging, or application logging statement.
System logging	SLA	File name to be parsed to store securely, and encryption key.
Application log	ALA	Database operation, table id, GPL method name, encryption key, and history preservation option.

name and encryption key. ALA specifies the database action name, database table name, GPL method name, encryption key, and history preservation option.

TABLE II: Abstract syntax of FAL

P ::= LS LA
LS ::= lid F S |LS1; LS2
F ::= type fid I encrypted |type fid I |F1; F2
S ::= encAlg hashAlg |encAlg |hashAlg |ϵ
I ::= n |Auto
LA ::= SLA |ALA |LA1; LA2
SLA ::= file key |file
ALA ::= action tid key m withhistory |action tid m key |
action tid m history |action tid m

TABLE III: Syntactic Domains

P ∈ Pgm	LS ∈ LogStructure
F ∈ Field	LA ∈ LogAction
I ∈ Index	S ∈ SecAttrs
SLA ∈ SystemLog	ALA ∈ AppLog
n ∈ Num	file ∈ FileSpec
type ∈ {IP, Text, Double, Integer, Time}	fid ∈ FileIdentifier
tid ∈ TableIdentifier	m ∈ MethodName
action ∈ {Add,Update,Delete}	key ∈ KeyFileSpec
lid ∈ LogStructureIdentifier	encAlg ∈ {RSA,AES}
hashAlg ∈ {MD5, SHA-1,SHA-256}	

C. The Concrete Syntax

After defining the abstract syntax, we experimented with various forms of concrete syntaxes to see how various constructs might look. For example, a log structure with two field *fromip* and *user* can be defined using the concrete syntax as described in Listing 1.

Listing 1: FAL Log Structure

```
1: Define netlog {
2:     IP fromip Index 0 Encrypted;
3:     TEXT user Index 1;
4:     Use Encryption With RSA;
5:     Use Logchain With SHA_1;
6: };
```

Here, *fromip* field has data type IP, and *user* is of TEXT data type. The *Index* attribute represents the position of a field in the network log file. The *Encrypted* attribute states that the field will be encrypted according to the encryption algorithm defined in line 4. If there are multiple encrypted fields, all the fields will be encrypted using the same encryption algorithm. Line 5 adds the flexibility of choosing any hash function.

After defining a log structure a log action will be defined, which uses the pre-defined log structure. A concrete example of storing a network log file securely can be defined as follows (Listing 2):

Listing 2: FAL Logging Action

```
1: Watchfile network.log Using netlog
2: {
3:     Privatekey private.key;
4: }
```

The *Watchfile* statement uses the predefined 'netlog' structure to parse the 'network.log' file and provides the required encryption key to start the process of preserving logs securely.

Listing 3: FAL Program for System and Application Log

```
1: SampleProgram[
2:     Define netlog {
3:         IP fromip Index 0 Encrypted;
4:         TEXT user Index 1;
5:         Use Encryption With RSA;
6:         Use Logchain With SHA_1;
7:     }
8:     Define patientlog{
9:         TIME logtime Auto;
10:        TEXT user Index 0 Encrypted;
11:        INT refid Index 1;
12:        TEXT message Index 2 Encrypted;
13:        Use Logchain With SHA_256;
14:    }
15:    Watchfile network.log Using netlog {
16:        Privatekey private.key;
17:    }
18:    Watchtable Patient Using patientlog {
19:        Action Edit Withhistory;
20:        Method updatepatient;
21:        Publickey public.key;
22:    }
23: ]
```

When a language designer is satisfied with the look and feel of the language's syntax, and possible additional constraints from domain experts or language end-users are fulfilled, the concrete syntax can be finalized. In Listing 3, a complete example of FAL program for secured system and application logs is described. We finalized the concrete syntax on the basis of several example programs. Finalizing the concrete syntax process can be executed in parallel with defining language semantics. In Table IV, the FAL concrete syntax is given.

TABLE IV: The concrete syntax of FAL

```
Program := #CCStart [LOG_STRUCT LOG_ACTION]
LOG_STRUCTS := LG_STRUCTS
LG_STRUCTS := LG_STRUCTS LG_STRUCT |LG_STRUCT
LG_STRUCT:= Define #Id {DEF}
DEF := FIELDS SEC_ATTRS
FIELDS := FIELDS FIELD |FIELD
FIELD := #Type #Id IND_BASE ENC ;
IND_BASE := Index #Number |Auto
ENC := Encrypted |ϵ
SEC_ATTRS := SEC_ATTRS SEC_ATTR |ϵ
SEC_ATTR := Use SEC_STMT ;
SEC_STMT := ENC_STMT |HASH_STMT
ENC_STMT := Encryption With #EncAlgorithm
HASH_STMT := Logchain With #HashAlgorithm
LOG_ACTION := LG_ACTIONS
LG_ACTIONS := LG_ACTIONS LG_ACTION |LG_ACTION
LG_ACTION := SYS_ACT |APP_ACT
SYS_ACT := Watchfile #FileName Using #Id {ENC_KEY}
ENC_KEY := PUB_KEY |PRIV_KEY |ϵ
PUB_KEY := Publickey #FileName;
PRIV_KEY := Privatekey #FileName;
APP_ACT := Watchtable #CCStart Using #Id {PARAM}
PARAM := DB_ACTION GPL_MTHD ENC_KEY
DB_ACTION := Action ACT_NAME ;
ACT_NAME := Add |ACT_HSTRY
ACT_HSTRY := ACT_HSTRY_NAME HISTRY_STMT
ACT_HSTRY_NAME := Edit |Delete
HISTRY_STMT := Withhistory |ϵ
GPL_MTHD := Method #Id ;
```

D. Translational Semantics

The advantages of using formal description for semantics of DSL (e.g., attribute grammars, denotational semantics, operational semantics) have been previously discussed in [10], where an ability to find problems in semantics before a DSL is actually implemented was exposed. In this work, we used translational semantics, which is simpler to define then denotational and operational semantics, and it is often used for defining semantics of domain-specific modeling languages [18]. Due to space considerations, only the translational semantics for log structures is presented in Listing 4 (translational semantics for logging actions is omitted from this paper). For each non-terminal in CFG, (Table II) a translational function is defined, which maps syntactic domains (Table III) to their meanings – generated code in Java using a specialized API for secure logging. In particular, the meaning of non-terminal LS is defined by translational function TLS, which maps $LogStructure$ to $code$. Two different forms of LS exist (see abstract syntax in Table II). Hence, two translational functions TLS are defined (lines 4 and 5 in Listing 4). The first translational function TLS (line 4 in Listing 4) maps syntactic structure $lid\ F\ S$ into several Java statements: declaration of new object as an instance of class $LogStructure$, setting a name to the newly created object by calling $setName$ method, and additional Java statements, which will be generated by applying translational functions TF and TS on non-terminals F and S representing fields and security attributes, respectively. Whilst, the second translational function TLS (line 5 in Listing 4) define the meaning of sequence of log structures ($LS1; LS2$). The generated code for $LS1$ is simply concatenated with generated code for $LS2$ (line 5 in Listing 4). In similar manner, other translational functions are defined.

Listing 4: Translational Semantics

```
1:  TP : Pgm → Code
2:  TP⟦LS LA⟧ = TLS⟦LS⟧ + TLA⟦LA⟧
3:  TLS : LogStructure → Code
4:  TLS⟦lid F S⟧ = "LogStructure " + lid + " = new LogStructure();" +
    lid+".setName(" + lid + ");" + TF⟦F⟧ lid + TS⟦S⟧ lid
5:  TLS⟦LS1; LS2⟧ = TLS⟦LS1⟧ + TLS⟦LS2⟧
6:  TF : Field → lid → Code
7:  TF⟦type fid I encrypted⟧ = lid+".addField(FieldType." + type + "," +
    fid + "," + TI⟦I⟧ + ", true);"
8:  TF⟦type fid I⟧ = lid+".addField(FieldType." + type + "," + fid + "," +
    TI⟦I⟧ + ", false);"
9:  TF⟦F1; F2⟧ = TF⟦F1⟧ + TF⟦F2⟧
10: TI : Index → Code
11: TI⟦n⟧ = "true, " + n
12: TI⟦Auto⟧ = "false, INTEGER.MAX_VALUE"
13: TS : SecAttrs → lid → Code
14: TS ⟦encAlg hashAlg⟧ = lid + ".setEncryptionAlgorithm("+ encAlg
15:     + ");" + lid + ".setHashingAlgorithm("+ hashAlg + ");"
16: TS ⟦encAlg⟧ = lid + ".setEncryptionAlgorithm("+ encAlg +");"
17: TS ⟦hashAlg⟧ = lid + ".setHashingAlgorithm("+ hashAlg + ");"
18: TLA : LogAction → Code
19: ...
```

E. Implementation

Various implementation techniques to implement a DSL exist, such as preprocessing, embedding, compiler/interpreter, compiler generator, extensible compiler/interpreter, commercial off-the-shelf, and hybrid approaches [10]. Kosar et al. [19] suggested focusing end-user usability while implementing a DSL. One implementation approach can be good in terms of effort needed to implement a DSL. However, the same approach may not be suitable for end-users. End-users may need extra effort to rapidly write correct programs using the DSL. If only DSL implementation effort is taken into consideration, then the most efficient implementation technique is embedding. However, the embedding approach might have significant penalties when end-user effort is taken into account (e.g., DSL program size, closeness to original notation, debugging and error reporting). To minimize end-user effort, building a DSL compiler [19] is most often a good solution, but this process costs most from an implementation point of view. However, the implementation effort can be greatly reduced, but not as much as with embedding, especially if compiler generators (e.g., LISA [20], ANTLR [21], Silver [22]) are used.

To implement FAL, we depend on source-to-source transformation technique. To transform a FAL program into an intermediate Java program, we build a FAL compiler using LISA, which has proven itself useful in many other DSL projects [23], [24]. The intermediate program uses a pre-build Java API. The design of the API is illustrated in Figure 2.

Fields are represented by *Field* class. The *LogStructue* has a list of Field object, and the security attributes. The name field of *LogStructure* is used to map with the database table name. *LogAction* is an Abstract class with the abstract method *execute*, and it also has an instance of *LogStructure*. *FileWatcher* extends the *LogAction* class and implements the execute method. The execute method is responsible to parse a log file and store it to database with the help of LogStructure and Field. *TableWatcher*

also extends the *LogAction* class and implements the execute method, which generates application logging code for developer. The *SecurityUtil* class defines all the required encryption and hashing methods.

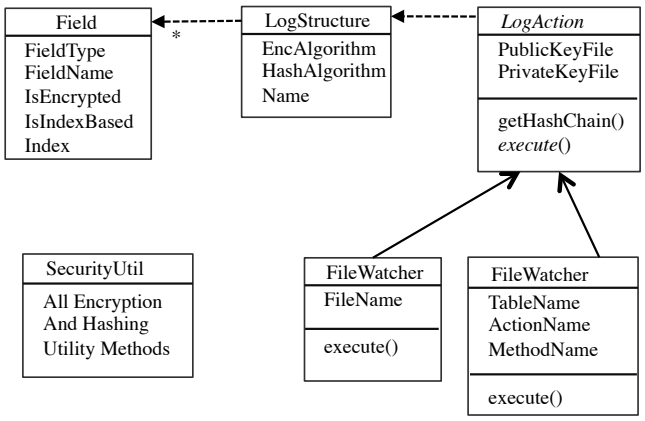

Fig. 2: Design of the API for FAL

After finalizing the Java API, we now know what the intermediate program will be. The FAL compiler will generate this intermediate program from a FAL program. To transform the FAL program to Java program correctly, we use the attribute grammar based approach and LISA specifications are based on attribute grammars [25], [26]. It is capable to generate the compiler from formal attribute grammar-based language specifications.

The first task to implement the compiler is to define the lexicon. Defining the lexicon in Lisa is straightforward. It is showed in Listing 5.

Listing 5: Lexical specification for FAL in LISA.

```
1:  lexicon {
2:      Number [0-9]+
3:      Id [a-z][a-z0-9_]*
4:      Type IP |TEXT |INT |TIME |DOUBLE
5:      EncAlgorithm RSA |AES
6:      HashAlgorithm MD5 |SHA_1 |SHA_256
7:      keywords Define |Use |Encryption |With |Logchain |Index |
8:          Auto |Encrypted |Watchfile |Using |Publickey |Privatekey |
9:          Watchtable |Action |Withhistory |Method |Parameter
10:     FileName [a-z][a-z0-9_]*.[a-z]*
11:     CCStart [A-Z][a-z0-9_]*
12:     ActionName Add |Edit |Delete
13:     Separator \; |\{ |\} |\, |\[ |\]
14:     ignore [ \0x09\0x0A\0x0D\]+
15: }
```

To write the attribute-based semantic rules, first, we need to identify the required attributes for proper semantic analysis. Listing 6 presents the attributes that we used. *code* is the main synthesized attribute that produces the targeted GPL program. *ivar* is an inherited attribute that is used to propagate the variable name down the parse tree. *envs* is a synthesized attribute and *envi* is an inherited attribute; both were needed to maintain a HashSet of already defined variables. *errorMsg* is a synthesized attribute required to report FAL error message to users. *ok* is a synthesized attribute that indicates whether a FAL program is correct or not. Finally, *PROGRAM.file* attribute is used to write the generated GPL program in a file.

Listing 6: Attributes for FAL in LISA.

```
1:  attributes String *.code;
2:      String *.ivar;
3:      String *.errorMsg;
4:      HashSet *.envs;
5:      HashSet *.envi;
6:      boolean *.ok;
7:      BufferedWriter PROGRAM.file;
```

An implementation of translational semantics (Listing 4) using LISA is a straightforward task. The implementation of translational function TF (Line 7 in Listing 4) is presented in Listing 7. Note, how closed both notations are.

Listing 7: Semantic Rules in LISA.

```
1:  rule field {
2:      FIELD ::= #Type #Id IND_BASE ENC \; compute {
3:          FIELD.code = FIELD.ivar + ".addField( FieldType." +
4:              #Type.value() + ",\"" + #Id.value()+"\", " +
5:              IND_BASE.code + "," + ENC.code+");";
6:      };
7:  }
8:  rule ind_base {
9:      IND_BASE ::= Index #Number compute {
10:         IND_BASE.code = "true,"+ #Number.value();
11:     }
12:     |Auto compute {
13:         IND_BASE.code = "false,Integer.MAX_VALUE";
14:     };
15: }
16: rule enc {
17:     ENC ::= Encrypted compute {
18:         ENC.code = "true";
19:     }
20:     |epsilon compute {
21:         ENC.code = "false";
22:     };
23: }
```

After compiling a FAL program, a required Java code is automatically generated, which uses previously defined APIs to store logs, and generate audit trail code for ensuring the integrity and confidentiality of the logs.

IV. Practical Experience

The goal of this section is to acquaint the reader with the practical experiences that were obtained by using FAL. We have therefore selected two case studies of FAL applications:

- Preserve snort log securely using FAL.
- Generate application logging code for a patient information update method in Java.

A. Preserve Snort log

Snort[3] is a free lightweight network intrusion detection system. The network logs generated by Snort plays vital role in network forensics. Hence, preserving the confidentiality and integrity of Snort logs is crucial from digital forensics perspective. Here is a sample Snort log:

```
11/19-13:43:43.222391 11.1.0.5:51215 ->
74.125.130.106:80  TCP TTL:64 TOS:0x0 ID:22101
```

[3]http://www.snort.org

```
IpLen:20 DgmLen:40 DF  ***A***F Seq: 0x3EA405D9
Ack: 0x89DE7D Win: 0x7210 TcpLen: 20''
```

This log tells that the machine with IP 11.1.0.5 performed an http request to machine 74.125.130.160 at time 11/19-13:43:43.222391. Hence, when a machine attacks another machine, we can identify the attacker machine IP from the snort log. Let's assume that a system admin decides to store the 'from IP', 'to IP', and time of network request securely. To protect the confidentiality of logs, among these three fields, the admin decides to encrypt 'from IP', and 'to IP' by the public key of law enforcement agencies using RSA algorithm. To protect the integrity of the logs, the system maintains hash-chain of the logs using SHA-256 hash function. The FAL program described in Listing 8 can be used to ensure all the properties.

Listing 8: FAL Program for Snort Log

```
SnortParser[
    Define snortlog {
        IP fromip Index 1 Encrypted;
        IP toip Index 3 Encrypted;
        Time logtime Index 0;
        Use Encryption With RSA;
        Use Logchain With SHA_256;
    };
    Watchfile snortnetwork.log Using snortlog {
        Publickey lawpublic.key;
    }
]
```

The above FAL program will generate Java code as follows (Listing 9):

Listing 9: Translated Java Code from FAL

```
LogStructure snortlog = new LogStructure();
snortlog.setName("snortlog");
snortlog.addField(FieldType.IP,"fromip",true,1,true);
snortlog.addField(FieldType.IP,"toip",true,2,true);
snortlog.addField(FieldType.TIME,"logtime",true,0,false);
snortlog.setEncryptionAlgorithm("RSA");
snortlog.setHashingAlgorithm("SHA_256");
FileWatcher snortlogFileWatcher = new FileWatcher();
snortlogFileWatcher.setLogStructure(snortlog);
snortlogFileWatcher.setFileName("snortnetwork.log");
snortlogFileWatcher.setPubicKeyFile("public.key");
snortlogFileWatcher.execute();
```

Executing the Java code (Listing 9) will parse the snort log file and store them with the security parameter. However, FAL users do not need to understand the underlying API or the intermediate Java code generated by FAL.

B. Application Logging

Application log is crucial for many applications including business and health care sector. The methods that directly communicate with database need to be logged. From these logs, later we can identify who added a new record, or who updated or deleted some record, etc. Application developer needs to integrate this logging feature with every method that updates database. FAL can generate the necessary logging code for application developer.

Here, we present a hypothetical scenario of a health care application, where we can use FAL for secure application logging. In the application, there is a Patient table, and we want to store logs whenever any update is operated on patient's record. The log will include the user name of the application, patient id is being updated, a description of the operation, and time of operation. The security analyst of the application decides to encrypt user name, and the operation description using AES encryption algorithm and SHA-1 hash function to maintain the hash-chain of logs. The FAL program described in Listing 10 can be used to generate necessary application logging code.

Listing 10: FAL Program for Application Logging

```
PatientAppLog [
    Define useraudit {
        TIME logtime Auto;
        TEXT username Index 0 Encrypted;
        INT refid Index 1;
        TEXT message Index 2 Encrypted;
        Use Encryption With AES;
        Use Logchain With SHA_1;
    };
    Watchtable Patient Using useraudit {
        Action Edit Withhistory;
        Method updatepatient;
        Privatekey serveraes.key;
    }
]
```

The translated Java code from FAL program (Listing 10) will generate the application logging method as described in Listing 11.

Listing 11: Generated Code For Application Logging

```
public void auditPatientEdit(String username, int refid, String message,
String logtime)
{
    try {
        String rowValue = username + refid + message + logtime;
        String currHashs = getHashChain("useraudit","id",rowValue,
            "SHA-1");
        String aesKey = HandleKey.readAESKey("serveraes.key");
        username = HandleKey.aesEncrypt(username +"", aesKey);
        message = HandleKey.aesEncrypt(message +"", aesKey);
        String query = "insert into useraudit( username, refid,
            message, logtime, tablename, actionname, methodname,
            logchain, withhistory) values("' + username
            + "'," + refid + ",'" + message + "','" + logtime +
            "','Patient','Edit','updatepatient','"+currHashs+"',true)";
        DBHandler dbHandler = new DBHandler();
        dbHandler.insertData(query);
    }catch (Exception e) { e.printStackTrace();}
}
```

V. Related Work

There has been a lot of prior research on secure logging, because of its vital role in digital forensics. Schneier et al. proposed a secure audit logging scheme, which can detect any modification of logs after a machine got compromised [27]. It also preserves the confidentiality of logs. Zawoad et al. proposes a secure logging scheme for cloud computing environment where the cloud service provider itself can be dishonest and can try to alter original logs [9]. Though there are no DSL for secure logging, there are some DSLs for providing access control facility on the audit logs or provenance

record and also for general-purpose access control. Ni et al. provided a XML-based access control language for general provenance model [28]. Using this language, users can define and evaluate access control policies on application audit logs. It also supports specifying policies to a particular record and its fields. Weissmann proposed ACS [29], an access control language for specifying access control policy, which especially resolves undecidability of granting or denying access, and incapability of editing control policy without changing the model. Ribeiro et al. provided SPL, an access control language for security policies with complex constraints [30]. SPL supports simultaneously multiple complex policies by resolving conflicts between two active policies. Beyond the permission / prohibition, they also showed how to express and implement the obligation concept.

VI. Conclusion and Future Work

For proper digital forensics investigation, maintaining the trustworthiness of logs is compulsory, and for this, we need a proper secure logging mechanism. To address the problem of secure logging mechanism, we have designed and implemented the domain-specific language FAL with the following benefits:

- Shifting the responsibility of developing a secure logging schemes from application programmers to security experts, which in turn increases trustworthiness.
- Required code to use specialized API for secure application logging is automatically generated. Hence, the effort and cost for developing secure logging scheme is lower.
- Heterogeneous formats of logs with any secure logging schemes can be easily handled.
- Detail understanding of specialized API for secure logging is not needed for FAL users.

We are working to add user specified delimiter feature with FAL. In future, we will add a user-friendly error reporting. Another important future feature is to incorporate timing option with system logging action. With this feature, users can define for how long they want to run system logging option. We will also work towards making FAL more robust so that it can generate audit-trailing code for all popular GPLs. Finally, FAL's design needs to be validated by end-users by performing usability studies and control experiments.

References

[1] FBI, "Annual report for fiscal year 2007," 2008 Regional Computer Forensics Laboratory Program, 2008, [Accessed July 5th, 2012].

[2] Congress of the United States, "Sarbanes-Oxley Act," http://thomas.loc.gov, 2002, [Accessed May 5th, 2013].

[3] U.S. Department of Health and Human Service, "Health information privacy," http://www.hhs.gov/ocr/privacy/, [Accessed May 5th, 2013].

[4] M. Swanson and B. Guttman, *Generally Accepted Principles and Practices for Securing Information Technology Systems*. National Institute of Standards and Technology (NIST), Technology Administration, US Department of Commerce, 1996.

[5] M. Bellare and B. Yee, "Forward-security in private-key cryptography," *Topics in Cryptology, CT-RSA 2003*, pp. 1–18, 2003.

[6] ——, "Forward integrity for secure audit logs," Technical report, Computer Science and Engineering Department, University of California at San Diego, Tech. Rep., 1997.

[7] D. Ma and G. Tsudik, "A new approach to secure logging," *Transaction of Storage (TOS)*, vol. 5, no. 1, pp. 2:1–2:21, Mar. 2009.

[8] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 159–176, May 1999.

[9] S. Zawoad, A. Dutta, and R. Hasan, "SecLaaS: Secure logging-as-a-service for cloud forensics," in *Proceedings of 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, May 2013.

[10] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[11] A. Van Deursen and P. Klint, "Little languages: Little maintenance?" *Journal of software maintenance*, vol. 10, pp. 75–92, 1998.

[12] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, C. D. Da, and R. P. Henriques, "Comparing general-purpose and domain-specific languages: An empirical study," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 247–264, 2010.

[13] R. Accorsi, "On the relationship of privacy and secure remote logging in dynamic systems," in *Security and Privacy in Dynamic Environments*. Springer US, 2006, vol. 201, pp. 329–339. [Online]. Available: http://dx.doi.org/10.1007/0-387-33406-8_28

[14] A. Van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 1–17, 2004.

[15] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[16] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[17] N. F. Pub, "197: Advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, pp. 441–0311, 2001.

[18] B. Bryant, J. Gray, M. Mernik, P. Clarke, R. France, and G. Karsai, "Challenges and directions in formalizing the semantics of modeling languages," *Computer Science and Information Systems*, vol. 8, no. 2, pp. 225–253, 2011.

[19] T. Kosar, P. A. Barrientos, M. Mernik *et al.*, "A preliminary study on various implementation approaches of domain-specific language," *Information and Software Technology*, vol. 50, no. 5, pp. 390–405, 2008.

[20] M. Mernik and V. Žumer, "Incremental programming language development," *Computer Languages, Systems & Structures*, vol. 31, no. 1, pp. 1–16, 2005.

[21] T. Parr, "The definitive ANTLR reference: Building domain-specific languages (pragmatic programmers)," *Pragmatic Bookshelf, May*, 2007.

[22] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: an extensible attribute grammar system," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 2, pp. 103–116, 2008.

[23] P. R. Henriques, M. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu, "Automatic generation of language-based tools using the LISA system," in *Software, IEE Proceedings-*, vol. 152, no. 2. IET, 2005, pp. 54–69.

[24] I. Fister, M. Mernik, and J. Brest, "Design and implementation of domain-specific language Easytime," *Computer Languages, Systems & Structures*, vol. 37, no. 4, pp. 151–167, 2011.

[25] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[26] J. Paakki, "Attribute grammar paradigms: a high-level methodology in language implementation," *ACM Computing Surveys (CSUR)*, vol. 27, no. 2, pp. 196–255, 1995.

[27] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 159–176, 1999.

[28] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han, "An access control language for a general provenance model," *Secure Data Management*, pp. 68–88, 2009.

[29] M. Weißmann, "Domain specific language for specifying access controls," Ph.D. dissertation, Georg Simon Ohm University of Applied Sciences, Nuernberg, Germany, 2007.

[30] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, "SPL: An access control language for security policies with complex constraints," in *Proceedings of the Network and Distributed System Security Symposium*, 2001, pp. 89–107.