Computer Science and Information Systems 0(0):1–6 DOI: N/A

Towards Building a Forensics Aware Language for Secure Logging

Shams Zawoad[1], Marjan Mernik[2], and Ragib Hasan[1]

[1] University of Alabama at Birmingham
Birmingham, AL-354209, USA
{zawoad,ragib}@cis.uab.edu
[2] University of Maribor
Maribor, Slovenia
marjan.mernik@uni-mb.si

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 Forensics Aware Language (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 later be 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 the required source code.

Keywords: DSL, Secure Logging, Audit Trail, Digital Forensics.

1. Introduction

In recent years, the number of digital crime cases has increased tremendously. An annual report of the Federal Bureau of Investigation (FBI) states that the size of the 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 [9]. Various logs, e.g., network log, process log, file access log, audit trail of applications, play a 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. Logs are also important to ensure the auditability of a system, which is crucial in making a system compliant with various regulatory acts, such as, the Sarbanes-Oxley Act (SOX) [7] or the Health Insurance Portability and Accountability Act (HIPAA) [36]. Keeping system audit trails and reviewing them in a consistent manner is recommended by the National Institute of Standards and Technology (NIST) as one of the good principles and practices for securing computer systems [35].

While the necessity of logs and application audit trails are indisputable, the trustworthiness of this evidence remains 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 attackers, they often attack

the logging system to hide the trace of their presence or to frame an honest user. Very often, experienced attackers first attack the logging system [2, 3]. Malicious insiders users, colluding with attackers can also tamper with logs. Moreover, forensics investigators can also alter evidence before it is presented in a court of law. To protect logs from these possible attacks, we need a secure logging mechanism. Researchers have already proposed several secure logging schemes [1, 2, 21, 32, 41], 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 on developers' side. To implement a secure logging scheme, application developers need complete access to the logs. However, providing developers with full access to sensitive logs definitely increases the attack surface. This opportunity enables the malicious developers to violate the privacy, acquire and sell sensitive business or personal information, and most importantly can keep a back door for future attack. Adding secure application audit trails can also be burdensome for developers. It also increases the application development cost. On the other hand, system administrators, who have access to network logs or process logs, may not have sufficient knowledge for developing a secure logging scheme.

In this paper, we propose Forensics Aware Language (FAL) – a domain-specific language (DSL) [23] to assist system administrators and application developers for maintaining system logs and application audit trails securely, which is crucial for digital forensics investigations. A DSL is designed for a particular domain and has great advantages over general-purpose languages for that specific domain. A DSL provides higher productivity by its greater expressive power, the ease of use, easier verification, and optimization [19,23,37]. Based on 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 general-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.

This paper is an extension of [42]. In this paper, we augmented the scheme presented in [42] by providing the complete translational semantics of FAL. We also made FAL more robust by providing a new feature. Previously, the delimiter to parse a system log file was fixed. In the new version, we added the user provided delimiter feature, by modifying all the DSL development steps described in [42] (such as abstract syntax, syntactic domain, grammar, translational semantics, and implementation). We also describe the life cycle for DSL development that we followed during the development of FAL.

The rest of the paper is organized as follows. Section 2 describes the background of secure logging and the motivation of developing a DSL to solve some of the challenges of secure logging. Section 3 discusses the life cycle for DSL development. In Section 4, we describe the development and implementation of FAL. Section 5 describes two practical

applications of FAL in two different scenarios. Section 6 discusses the related work in secure logging and usage of DSL in security domain. Finally, we conclude in Section 7.

2. Background and Motivation

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

2.1. Secure Logging

As logs are crucial for digital forensics investigation, attackers often target logs to destroy the evidence. 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, re-order the log entries, and add fake logs. A malicious user can launch these attacks to hide the trace of his 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**: Activity of users, as well as, sensitive private information about the users can be identified from various system logs and application logs. 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 malicious investigators 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 the 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 [1, 21, 32, 41]. 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 chronological information.

2.2. Motivation

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

1. One of the problems in developing a universal secure logging scheme is that logs exist in heterogeneous format. Unfortunately, there is no standard format of logs. 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 introduces an extra level of trust. Developers 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 identify who has executed some specific operations on a specific data. Writing code for all possible scenarios is burdensome for developers. On the other hand, skipping one important logging method may turn out to be crucial.

We believe that a well-defined DSL would be able to resolve the above challenges. 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. As system admins already have the domain knowledge about system logs, they can easily define the required security parameters with the help of a DSL. In this way, we can reduce one level of attack surface.

Since one of the main challenges of integrating a secure logging scheme is that logs are in heterogeneous formats, a DSL should also deal with this issue. A secure logging scheme that is already integrated for one log format, need to be changed for another log format. Instead of using a GPL, if we use a DSL which can cope up with heterogeneous formats of logs, the amount of code that has to be changed can be highly reduced. Moreover, 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 reduce the application development cost.

To integrate a secure logging scheme, knowledge about existing encryption and hashing algorithms should also be integrated with a DSL. For FAL, this knowledge is embedded with the specialized Application Programming Interface (API) (details in Section 4.5). However, if we want to use a proprietary encryption or hashing algorithm, we need to upgrade the DSL to provide the knowledge of that encryption or hashing algorithm. For example, FAL supports the common hashing algorithms: MD5, SHA-1, SHA-256, and SHA-512. To use the SHA-1024 hashing algorithm, we need to upgrade FAL and the API

to integrate the SHA-1024 hashing algorithm. Hence, our proposed DSL can only handle established encryption and hashing algorithms.

3. DSL Development Methodology

A DSL life cycle comprises of the following phases: decision, domain analysis, DSL design, DSL implementation, DSL testing, DSL deployment, and DSL maintenance [5, 23]. During the decision phase several criteria need to be evaluated and contrasted to find out whether the development of a new DSL is a solution to our problem. In this respect, decision patterns [23] might be helpful as they indicate those situations of the past, where the introduction of a DSL into a process had been successful. If the decision about implementing a DSL is found to be positive during the initial phase, then the next stage is a DSL development, which is a topic of this Section. It is comprised of the following phases: domain analysis, DSL design, and DSL implementation. These phases are crucial during a DSL life cycle and appropriate methodology is needed to do it correctly. Many DSLs have been developed from scratch by informally performing a particular phase (domain analysis, DSL design, DSL implementation), certain parts of a phase (e.g., semantic part of a DSL design), or even all the phases. There are several problems with the 'from scratch' approach. The more notable problems are: often an unsatisfactory DSL is developed and several costly re-development iterations are needed, difficult maintenance, and that DSL evolution is hard. For example, often problems that should have been identified within early phases only become visible during later phases. Hence, such an informal approach to DSL development is not recommended. In this section, a particular formal DSL development methodology is described. Namely, domain analysis, DSL design, and DSL implementation are not narrow processes and various formalisms can be applied.

The task of domain analysis is to select and define the domain of focus, collect appropriate domain information, and integrate them into a coherent domain model that represents concepts within a domain and relationships within the domain concepts. Here several existing domain analysis methodologies can be used. In particular, our recommendation is to use Feature-Oriented Domain Analysis (FODA) [15] since common and variable properties of a domain are easy to identify in feature diagrams (i.e., variation points). In fact, the list of variations indicates precisely which information is required for specifying an instance within a system. This information must be either directly specified within programs written in a DSL or be derivable from them. On the other hand, the commonalities are used for defining the execution model (through a set of common operations) and the primitives of the language. The outputs from domain analysis are: terminology, concepts, commonalities, and variations. These are easily identified from FODA feature diagrams [34] and should be used as inputs into the next phase – DSL design.

Designing a language involves defining the constructs within the language (syntax) and giving semantics to the language. Both sub-phases, syntax and semantics, can be managed informally or formally. The advantages of formal syntax and semantic specification of programming languages are well-known: the structure and meaning of a program is precisely and unambiguously defined, and it offers a unique possibility for the automatic generation of compilers or interpreters. Programming languages that have been designed using one of the various formal methods for syntax and semantic definitions have better syntax and semantics, lesser number of exceptions, and easier learning curve. Moreover,

researchers have recognized the possibility that many other language-based tools could be generated from formal language specifications. Therefore, many language implementation systems not only automatically generate a compiler/interpreter but also complete language-based environments including editors, type checkers, debuggers, various analyzers, and animators [13]. The following formal methods have been used for DSL syntax definition: BNF, FDL [14], metamodels, DTD, and XML Schema. The powers of all these formal methods for syntax definition are the same. Hence, transformations between different syntax descriptions are more or less easy to achieve. In our DSL development methodology, we opted for BNF since many language implementation systems (i.e., compiler generators [6,8,12,24]) use variants of BNF. Semantic formalisms are usually based on abstract syntax instead of concrete syntax. Hence, both forms need to be developed as concrete syntax is later required when parsing. Whilst different syntax formalisms are equivalent, the situation is quite different for the semantics, where approaches such as attribute grammars, axiomatic semantics, operational semantics, denotational semantics, and translational semantics are complementary, and used by different stakeholders. For example, attribute grammars are used by compiler writers, whilst axiomatic and denotational semantics are used by language designers to prove various language properties without concentrating on particular implementation. On the other hand, operational semantics define the meaning of the language through configuration changes and is closer to the implementation on virtual machines. Another distinction amongst different semantic formalisms is whether they are able to describe the static and/or dynamic semantics of a language. In our DSL development methodology, we used the translational semantics for code generation.

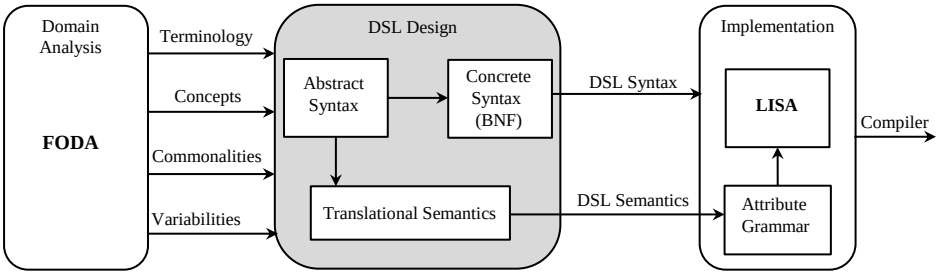

Fig. 1: FAL Development Life Cycle

Finally, after DSL has been designed, it is time for its implementation. Different approaches for DSL development have been introduced in [23], such as interpreter, compiler/application generator, embedding, preprocessing, extensible compiler/interpreter, Commercial Off-The-Shelf (COTS), and the hybrid approach. Clearly we want to select an approach that requires the least effort during implementation and offers the greatest efficacy to the end-user [17]. In our approach to DSL development, the formal specifications during design phase constitute an important part. Of course, it is harder to design a DSL formally than informally. This pays off during the DSL implementation phase, where a complete compiler/interpreter can be automatically generated. This is achieved in our case by mapping translational semantics to the language implementation system LISA [24], which is

based on attribute grammars [16,27]. Code generation using translational semantics is easy to implement in attribute grammars.

The whole process of our methodology for DSL development is presented in Figure 1. Section 4 shows how our DSL development methodology has been used for the development of FAL.

4. The Domain-Specific-Language FAL

4.1. Domain Analysis

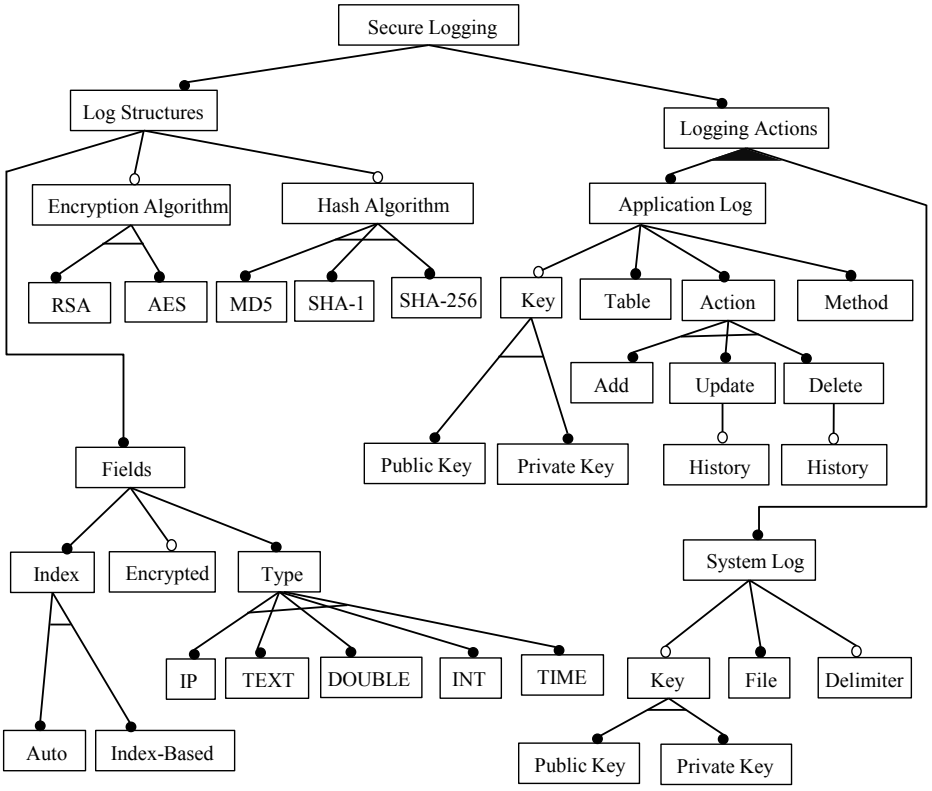

Fig. 2: The Feature Diagram of FAL

The very first step of designing a DSL is the detailed analysis and structuring of the application domain [38], 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. In FODA, the results of the domain analysis are obtained in a feature model [33]. One of the most prominent ways of describing a feature model is 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. Nodes can be mandatory or optional, which are denoted by closed dots and open dots respectively. The FD of FAL is illustrated in Figure 2.

From Figure 2, 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 can be used to specify the position 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, various encryption algorithms, such as, RSA [31], AES [29] can be used. Some secure logging mechanisms use hashing and hash-chain to ensure the integrity of 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 system log files according to a predefined 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 and we can verify that signature using the public key. 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 deletion, we may want to save the history of previous records.

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 with their interdependencies.

4.2. 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 identify the relationship between concepts/features in an application domain and non-terminals in a context-free grammar (CFG). Table 1 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.

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

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, delimiter used in parsing, and the encryption key.
Application log	ALA	Database operation, table id, GPL method name, encryption key, and history preservation option.
System Log Encryption Key	SLPK	Public key or private key encryption file for encrypting system log.
Application Log Encryption Key	ALPK	Public key or private key encryption file for encrypting application log.

Table 1: Translation of the application domain concepts to a context-free grammar

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 ::= slaid file SLPK |slaid file SLPK delimiter
ALA ::= alaid action tid mid withhistory ALPK |alaid action tid mid ALPK
SLPK ::= pubKey |privKey |ϵ
ALPK ::= pubKey |privKey |ϵ

Table 2: Abstract syntax of FAL

Based on Table 1, we define the abstract syntax of FAL, which is presented in Table 2. The syntactic domains of variables are presented in Table 3. 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 descriptor 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*. A field that has *auto* as the index, indicates that the value of the field is not extracted from a certain position of a given log file (for system log) or does not bind with a position of function parameters (for application log). The value of this field is generated from intermediate code, such as current time. Security parameter S defines encryption and hashing algorithm. Logging action LA can be either

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	mid ∈ MethodName
lid ∈ LogStructureIdentifier	action ∈ {Add,Update,Delete}
hashAlg ∈ {MD5, SHA-1,SHA-256}	encAlg ∈ {RSA,AES}
SLPK ∈ SysLogEncryptionFile	ALPK ∈ AppLogEncryptionFile
slaid ∈ SystemLogActionIdentifier	alaid ∈ AppLogActionIdentifier
pubKey ∈ PublicKeyFileSpec	privKey ∈ PrivateKeyFileSpec
delimiter ∈ ASCII Character Sequence	

Table 3: Syntactic Domains

System logging action SLA or Application logging action ALA. There can be one or more logging actions. SLA specifies the system log file name, delimiter to be used in parsing, and encryption key. ALA specifies the database action name, database table name, GPL method name, encryption key, and history preservation option. SLPK and ALPK specify the public key/private key for system logging and application logging respectively.

4.3. The Concrete Syntax

After defining the abstract syntax, we experimented with different forms of concrete syntaxes to see how various constructs might look. For example, a log structure with two fields *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, we define a logging action, 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

```
Watchfile network.log Using netlog
{
    Privatekey private.key;
    Delimiter ";";
}
```

The *Watchfile* statement uses the previously defined 'netlog' structure to parse the 'network.log' file and uses the private.key, a private key encryption file to encrypt the *fromip* field defined in Listing 1.

Listing 3: FAL Program for System and Application Log

```
SampleProgram[
    Define netlog {
        IP fromip Index 0 Encrypted;
        TEXT user Index 1;
        Use Encryption With RSA;
        Use Logchain With SHA_1;
    }
    Define patientlog{
        TIME logtime Auto;
        TEXT user Index 0 Encrypted;
        INT refid Index 1;
        TEXT message Index 2 Encrypted;
        Use Logchain With SHA_256;
    }
    Watchfile network.log Using netlog {
        Privatekey private.key;
        Delimiter ";";
    }
    Watchtable Patient Using patientlog {
        Action Edit Withhistory;
        Method updatepatient;
        Publickey public.key;
    }
]
```

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 4, we provide the concrete syntax 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 DELIM}
ENC_KEY := PUB_KEY |PRIV_KEY |ϵ
PUB_KEY := Publickey #FileName;
PRIV_KEY := Privatekey #FileName;
DELIM := Delimiter #UserDelimiter; |ϵ
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 ;

Table 4: The concrete syntax of FAL

4.4. 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 [23]. The authors of [23] discussed the ability to find problems in semantics before a DSL is actually implemented. In this work, we used translational semantics, which is simpler to define compared to denotational and operational semantics, and it is often used for defining semantics of domain-specific modeling languages [4]. Listing 4 provides the complete translational semantics of FAL. For each non-terminal in CFG (Table 2), a translational function is defined, which maps syntactic domains (Table 3) to their meanings – Java code that uses a specialized API for secure logging. For example, the meaning of non-terminal LS is defined by translational function TLS, which takes $LogStructure$ as input and return two components: first one is $code$ and the second one is lid (object id of the LogStructure class). Two different forms of LS exist (see abstract syntax in Table 2). Hence, two translational functions TLS are defined (lines 4 and 5 in Listing

Listing 4: Translational Semantics

1: TP : Pgm → Code
2: TP⟦$LS\ LA$⟧ = (TLS⟦LS⟧) ↓ 1 + TLA⟦LA⟧ (TLS⟦LS⟧) ↓ 2
3: TLS : LogStructure → Code × lid
4: TLS⟦$lid\ F\ S$⟧ = (“LogStructure ” + lid + “ = new LogStructure();” + lid+“.setName(” + lid + “);” + TF⟦F⟧ lid + TS⟦S⟧ lid, lid)
5: TLS⟦$LS1; LS2$⟧ = ((TLS⟦$LS1$⟧)↓ 1 + (TLS⟦$LS2$⟧)↓ 1, (TLS⟦$LS1$⟧) ↓ 2)
6: TF : Field → lid → Code
7: TF⟦$type\ fid\ I\ encrypted$⟧ lid = lid+“.addField(FieldType.” + type + “,” + fid + “,” + TI⟦I⟧ + “, true);”
8: TF⟦$type\ fid\ I$⟧ lid = lid+“.addField(FieldType.” + type + “,” + fid + “,” + TI⟦I⟧ + “, false);”
9: TF⟦$F1; F2$⟧ lid = TF⟦$F1$⟧ lid + TF⟦$F2$⟧ lid
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 = lid + “.setEncryptionAlgorithm(”+ encAlg + “);” +
15: lid + “.setHashingAlgorithm(”+ hashAlg + “);”
16: TS⟦$encAlg$⟧ lid = lid + “.setEncryptionAlgorithm(”+ encAlg +“);”
17: TS⟦$hashAlg$⟧ lid = lid + “.setHashingAlgorithm(”+ hashAlg + “);”
18: TLA : LogAction → lid → Code
19: TLA⟦SLA⟧ lid = TSLA⟦SLA⟧ lid
20: TLA⟦ALA⟧ lid = TALA⟦ALA⟧ lid
21: TLA⟦$LA1; LA2$⟧ lid = TLA⟦$LA1$⟧ lid + TLA⟦$LA2$⟧ lid
22: TSLA : SystemLog → lid → Code
23: TSLA⟦$slaid\ file\ SLPK$⟧ lid = “FileWatcher ” + slaid + “ = new FileWatcher();” + slaid + “.setLogStructure(” + lid + “);” + slaid + “.setFileName(” + file + “);” + TSLPK⟦$SLPK$⟧ slaid + slaid + “.setDelimiter(\“ \”);”
24: TSLA⟦$slaid\ file\ SLPK\ delimiter$⟧ lid = “FileWatcher ” + slaid + “ = new FileWatcher();” + slaid + “.setLogStructure(” + lid + “);” + slaid + “.setFileName(” + file + “);” + TSLPK⟦$SLPK$⟧ slaid + slaid + “.setDelimiter(“ + delimiter +”);”
25: TSLPK : SysLogEncryptionFile → slaid → Code
26: TSLPK⟦$pubKey$⟧ slaid = slaid + “.setPublicKeyFile(” + pubKey + “);”
27: TSLPK⟦$privKey$⟧ slaid = slaid + “.setPrivateKeyFile(” + privKey + “);”
28: TALA : AppLog → lid → Code
29: TALA⟦$alaid\ action\ tid\ mid\ withhistory\ ALPK$⟧ lid = “TableWatcher ” + alaid + “ = new TableWatcher();” + alaid + “.setLogStructure(” + lid + “);” + alaid + “.setAction(” + action + “);” + alaid + “.setTable(” + tid + “);” + alaid + “.setMethod(” + mid + “);” + alaid + “.setMaintainHistory(true);” + TALPK⟦$ALPK$⟧ alaid
30: TALA⟦$alaid\ action\ tid\ mid\ ALPK$⟧ lid = “TableWatcher ” + alaid + “ = new TableWatcher();” + alaid + “.setLogStructure(” + lid + “);” + alaid + “.setAction(” + action + “);” + alaid + “.setTable(” + tid + “);” + alaid + “.setMethod(” + mid + “);” + alaid + “.setMaintainHistory(false);” + TALPK⟦$ALPK$⟧ alaid
31: TALPK : AppLogEncryptionFile → alaid → Code
32: TALPK⟦$pubKey$⟧ alaid = alaid + “.setPublicKeyFile(” + pubKey + “);”
33: TALPK⟦$privKey$⟧ alaid = alaid + “.setPrivateKeyFile(” + privKey + “);”

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. The additional statements will be generated by applying translational functions TF and TS on non-terminals F and S, where F and S represent fields and security attributes respectively. This function also returns the lid as the second parameter. Whilst, the second translational function TLS (line 5 in Listing 4) defines 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.

4.5. 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 [23]. Kosar et al. [17] 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 that 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-users' effort, building a DSL compiler [17] 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 [25], ANTLR [28], Silver [39]) 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 its usefulness in many other DSL projects [10, 11, 13, 20, 22]. The intermediate program uses a pre-build Java API.

Design of the Java API is illustrated in Figure 3. Fields are represented by *Field* class. The *LogStructure* 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 into the 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.

After finalizing the Java API, we now know what the intermediate program will be. For example, the API provides *addField(Enum FieldType, String fieldName, boolean isEncrypted, int index, boolean isIndexBased)* method to add a new field. For using a specific encryption and hashing algorithm, the intermediate program can use *setEncryptionAlgorithm(String algoName)* and *setHashingAlgorithm(String algoName)* methods provided by the API. The FAL compiler will generate this intermediate program from a FAL program. To transform the FAL program to Java program correctly, we use the

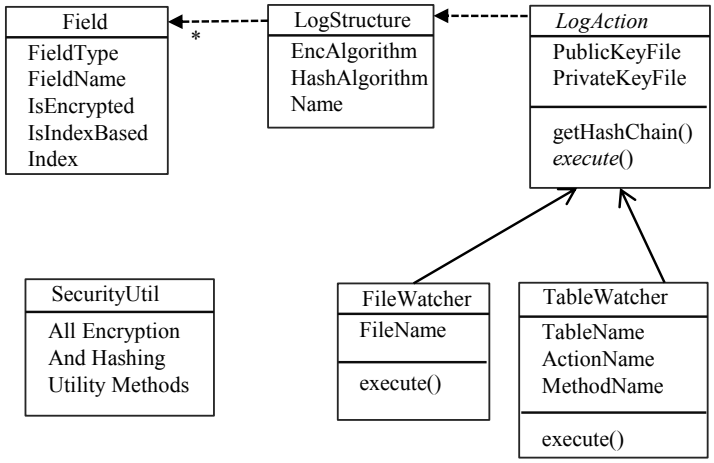

Fig. 3: Design of the API for FAL

attribute grammar-based approach as LISA specifications are based on attribute grammars [16, 27]. 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

```
lexicon {
    Number [0-9]+
    Id [a-z][a-z0-9_]*
    Type IP |TEXT |INT |TIME |DOUBLE
    EncAlgorithm RSA |AES
    HashAlgorithm MD5 |SHA_1 |SHA_256
    keywords Define |Use |Encryption |With |Logchain |Index |
        Auto |Encrypted |Watchfile |Using |Publickey |Privatekey |
        Watchtable |Action |Withhistory |Method |Parameter
    FileName [a-z][a-z0-9_]*.[a-z]*
    UserDelim "[\0x20 - \0x7E]+"
    CCStart [A-Z][a-z0-9_]*
    ActionName Add |Edit |Delete
    Separator \; |\{ |\} |\, |\[ |\]
    ignore [ \0x09\0x0A\0x0D\]+
}
```

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 are 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 (Lines 7 and 8 in Listing 4) is presented in Listing 7. Note, how closed both notations are.

After compiling a FAL program, the required Java code will be automatically generated. The generated code utilizes predefined APIs to store logs, and generate audit trail code for ensuring the integrity and confidentiality of the logs.

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: }
```

5. 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.

5.1. Preserve Snort log

Snort[3] is a free lightweight network intrusion detection system. The network logs generated by Snort play a 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 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 these properties.

Listing 8: FAL Program for Snort Log

```
 1: SnortParser[
 2:     Define snortlog {
 3:         IP fromip Index 1 Encrypted;
 4:         IP toip Index 3 Encrypted;
 5:         Time logtime Index 0;
 6:         Use Encryption With RSA;
 7:         Use Logchain With SHA_256;
 8:     };
 9:     Watchfile snortnetwork.log Using snortlog {
10:         Publickey lawpublic.key;
11:     }
12: ]
```

The above FAL program will generate the Java code provided in Listing 9.

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

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.

5.2. 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 the person who has modified (add/update/delete) any record. Application developer needs to integrate this logging feature with every method that updates database. FAL can generate the necessary logging code for application developer.

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;
    }
]
```

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. For such application, a log entry should include the user name, who executed an operation, 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.

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

```
1:  public void auditPatientEdit(String username, int refid, String message, String logtime)
2:  {
3:      try {
4:          String rowValue = username + refid + message + logtime;
5:          String currHashs = getHashChain("useraudit","id",rowValue,
6:              "SHA-1");
7:          String aesKey = SecurityUtil.readAESKey("serveraes.key");
8:          username = SecurityUtil.aesEncrypt(username +"", aesKey);
9:          message = SecurityUtil.aesEncrypt(message +"", aesKey);
10:         String query = "insert into useraudit( username, refid,
11:             message, logtime, tablename, actionname, methodname,
12:             logchain, withhistory) values(''' + username
13:             + "'," + refid + ",'" + message + "','" + logtime +
14:             "','Patient','Edit','updatepatient','"+currHashs+"',true)";
15:         DBHandler dbHandler = new DBHandler();
16:         dbHandler.insertData(query);
17:     } catch (Exception e) { e.printStackTrace();}
18: }
```

6. Related Work

As logging information is one of the prime needs in forensic investigation, several researchers have explored this problem across multiple dimensions. There have been number of cryptographic approaches to address security for audit logs that are generated and stored on local logging servers [2, 3, 32]. Bellare et al. provided a solution for secure logging, where the encryption/decryption key of a logging server has been compromised but the attacker cannot read or modify the previously encrypted logs [2, 3]. Schneier et al. proposed a secure audit logging scheme, where the log information are stored in an untrusted machine [32]. The proposed cryptographic scheme ensures that after an attack, the attacker can acquire little or no information and cannot alter the sensitive log information without being detected. In their scheme, they used public key and private key based encryption, message authentication code, and hashing. According to Schneier's scheme, a logging machine U opening a new audit log first establishes a shared secret key A_0 with a trusted remote server T. After each audit entry is generated, the current secret key A_i is evolved into A_{i+1} through a one-way function. Log entries are linked using a hash chain.

Secure logging in cloud computing environment, where users can run virtual machine (VM) on cloud infrastructure requires special attention due to the inherent nature of clouds.

Zawoad et al. proposed a secure logging scheme, SecLaaS for cloud computing environment [41]. While proposing SecLaaS, they considered the cloud service provider as dishonest who can collude with an attacker to tamper with the original logs. Alteration of original logs can hide the trace of malicious behavior of the attacker and impede the forensics investigation process. They used public/private key-based encryption and hash-chain scheme to ensure the privacy and integrity of cloud VM logs. The schemes stated earlier are against post-compromise insertion, alteration, deletion, and reordering pre-compromise of log entries.

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 [26]. The language supports the specification of both actor preferences and organizational access control policies. 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. However, in this paper, the authors did not provide the language development process. Ribeiro et al. provided SPL, an access control language for security policies with complex constraints [30]. SPL supports simultaneous 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. This paper also did not provide the details of language development process

Weissmann proposed ACS (Access Control Sets), an access control language to solve the access control problem of UMLsec[4] and aspect-oriented programming [40]. The proposed language particularly tries to solve the problem of undecidability in granting or denying a privilege, incapability of changing access controls without changing the model, incapability of delegating access control specifications, and inflexibility of UML to define relations other than logical. Domain analysis of the language was executed informally, and the author provided BNF grammar for the language. The language is based on mathematical concepts of sets, hence the semantics of ACS closely follow that of set theory. This language can be used in a business application to define all the policies of the business organization, which can make both writing and modifying access control specifications easy by reducing the human interaction with the security code .

7. 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 reduced.
- Heterogeneous formats of logs with any secure logging schemes can be easily handled.

[4] http://www4.in.tum.de/%CB%9Cumlsec/

– Detail understanding of specialized API for secure logging is not needed for FAL users.

One important feature that we are planning to incorporate with FAL is a timing option with system logging action. With this feature, users can define when they want to start the system logging and for how long they want to run the system logging option. Currently, FAL does not have user friendly error reporting feature, which we will integrate in future. For example, if a FAL user uses same index value for two fields, or uses an encryption algorithm that is not available with FAL, these problems should be detected at compile time and appropriate messages will be shown to user. For now, FAL generates audit-trailing code for Java. We will also work towards making FAL more robust so that it can generate audit-trailing code for other popular GPLs such as C++, C#, Python, Ruby, etc. To accomplish this goal, we need to develop the current Java API for other GPLs. Finally, FAL's design needs to be validated by end-users by performing usability studies and control experiments [18].

References

1. Accorsi, R.: On the relationship of privacy and secure remote logging in dynamic systems. In: Security and Privacy in Dynamic Environments, vol. 201, pp. 329–339. Springer US (2006), http://dx.doi.org/10.1007/0-387-33406-8_28
2. Bellare, M., Yee, B.: Forward integrity for secure audit logs. Tech. rep., Technical report, Computer Science and Engineering Department, University of California at San Diego (1997)
3. Bellare, M., Yee, B.: Forward-security in private-key cryptography. Topics in Cryptology, CT-RSA 2003 pp. 1–18 (2003)
4. Bryant, B., Gray, J., Mernik, M., Clarke, P., France, R., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. Computer Science and Information Systems 8(2), 225–253 (2011)
5. Čeh, I., Črepinšek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Computer Science and Information Systems 8(2), 317–342 (2011)
6. Cervelle, J., Forax, R., Roussel, G.: A simple implementation of grammar libraries. Computer Science and Information Systems 4(2), 65–77 (2007)
7. Congress of the United States: Sarbanes-Oxley Act. http://thomas.loc.gov (2002), [Accessed May 5th, 2013]
8. Cordy, J.R., Halpern-Hamu, C.D., Promislow, E.: Txl: A rapid prototyping system for programming language dialects. Computer Languages 16(1), 97–107 (1991)
9. FBI: Annual report for fiscal year 2007. 2008 Regional Computer Forensics Laboratory Program (2008), [Accessed July 5th, 2012]
10. Fister, I.J., Fister, I., Mernik, M., Brest, J.: Design and implementation of domain-specific language Easytime. Computer Languages, Systems & Structures 37(4), 151–167 (2011)
11. Fister, I.J., Kosar, T., Fister, I., Mernik, M.: EasyTime++: A case study of incremental domain-specific language development. Information Technology and Control 42(1), 77–85 (2013)
12. Hedin, G., Magnusson, E.: JastAdd: An aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
13. Henriques, P.R., Pereira, M.V., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic generation of language-based tools using the LISA system. Software, IEE Proceedings - 152(2), 54–69 (2005)
14. de Jonge, M., Visser, J.: Grammars as feature diagrams. In: ICSR7 Workshop on Generative Programming. pp. 23–24 (2002)

15. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
16. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory 2(2), 127–145 (1968)
17. Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 50(5), 390–405 (2008)
18. Kosar, T., Mernik, M., Carver., J.: Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empirical Software Engineering 7(3), 276–304 (2012)
19. Kosar, T., Oliveira, N., Mernik, M., Pereira, M.V., Črepinšek, M., Cruz, D. da., Henriques, P.R.: Comparing general-purpose and domain-specific languages: An empirical study. Computer Science and Information Systems 7(2), 247–264 (2010)
20. Lukovič, I., Pereira, M.V., Oliveira, N., Cruz, D. da., Henriques, P.R.: A DSL for PIM specifications: Design and attribute grammar based implementation. Computer Science and Information Systems 8(2), 379–403 (2011)
21. Ma, D., Tsudik, G.: A new approach to secure logging. Transaction of Storage (TOS) 5(1), 2:1–2:21 (Mar 2009)
22. Mernik, M.: An object-oriented approach to language compositions for software language engineering. Journal of Systems and Software 86(9), 2451–2464 (2013)
23. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) 37(4), 316–344 (2005)
24. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: Lisa: An interactive environment for programming language development. In: Compiler Construction. pp. 1–4. Springer (2002)
25. Mernik, M., Žumer, V.: Incremental programming language development. Computer Languages, Systems & Structures 31(1), 1–16 (2005)
26. Ni, Q., Xu, S., Bertino, E., Sandhu, R., Han, W.: An access control language for a general provenance model. Secure Data Management pp. 68–88 (2009)
27. Paakki, J.: Attribute grammar paradigms: a high-level methodology in language implementation. ACM Computing Surveys (CSUR) 27(2), 196–255 (1995)
28. Parr, T.: The definitive ANTLR reference: Building domain-specific languages (pragmatic programmers). Pragmatic Bookshelf, May (2007)
29. Pub, N.F.: 197: Advanced encryption standard (AES). Federal Information Processing Standards Publication 197, 441–0311 (2001)
30. Ribeiro, C., Zuquete, A., Ferreira, P., Guedes, P.: SPL: An access control language for security policies with complex constraints. In: Proceedings of the Network and Distributed System Security Symposium. pp. 89–107 (2001)
31. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
32. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Transactions on Information and System Security (TISSEC) 2(2), 159–176 (May 1999)
33. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)
34. Štuikys, V., Damaševicius, R.: Measuring complexity of domain models represented by feature diagrams. Information Technology and Control 38(3), 179–187 (2009)
35. Swanson, M., Guttman, B.: Generally Accepted Principles and Practices for Securing Information Technology Systems. National Institute of Standards and Technology (NIST), Technology Administration, US Department of Commerce (1996)
36. U.S. Department of Health and Human Service: Health information privacy. `http://www.hhs.gov/ocr/privacy/`, [Accessed May 5th, 2013]

37. Van Deursen, A., Klint, P.: Little languages: Little maintenance? Journal of software maintenance 10, 75–92 (1998)
38. Van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10(1), 1–17 (2004)
39. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Electronic Notes in Theoretical Computer Science 203(2), 103–116 (2008)
40. Weißmann, M.: Domain Specific Language for Specifying Access Controls. Ph.D. thesis, Georg Simon Ohm University of Applied Sciences, Nuernberg, Germany (2007)
41. Zawoad, S., Dutta, A., Hasan, R.: SecLaaS: Secure logging-as-a-service for cloud forensics. In: Proceedings of 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS) (May 2013)
42. Zawoad, S., Mernik, M., Hasan, R.: FAL: A forensics aware language for secure logging. In: Proceedings of the 2013 Federated Conference on Computer Science and Information Systems. pp. 1579–1586 (2013)