A High-Level Aspect-Oriented Based Framework for Software Security Hardening

Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi *

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{mourad,ma_laver,debbabi}@ciise.concordia.ca

Abstract. In this paper, we present an aspect-oriented approach and propose a high-level language called *SHL* (Security Hardening Language) for the systematic security hardening of software. The primary contribution of this proposition is providing the software architects with the capabilities to perform security hardening by applying well-defined solutions and without the need to have expertise in the security solution domain. At the same time, the security hardening is applied in an organized and systematic way in order not to alter the original functionalities of the software. This is done by providing an abstraction over the actions required to improve the security of a program and adopting aspect-oriented programming to build and develop the solutions. *SHL* allows the developers to describe and specify the security hardening plans and patterns needed to harden systematically security into open source software. It is a minimalist language built on top of the current aspect-oriented technologies that are based on advice-poincut model and can also be used in conjunction with them. We explore the viability and relevance of our proposition by applying it into several security hardening case studies and presenting their experimental results.

Key words: Software Security Hardening, Aspect-Oriented Programming (AOP), Security Hardening Patterns, Security Hardening Plans, Open Source Software, Weaving

1 Introduction

Security is taking an increasingly predominant role in today's computing world. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the

* This research is the result of a fruitful collaboration between CSL (Computer Security Laboratory) of Concordia University, DRDC (Defence Research and Development Canada) Valcartier and Bell Canada under the NSERC DND Research Partnership Program.

box, even on programs that were not designed with security in mind. The challenge is even greater when legacy systems must be adapted to networked/web environments, while they are not originally designed to fit into such high-risk environments. Tools and guidelines have been available for developers for few years already, but their practical adoption is limited so far. Software maintainers must face the challenge to improve program security and are often underequipped to do so. In some cases, little can be done to improve the situation, especially for Commercial-Off-The-Shelf (COTS) software products that are no longer supported, or for in-house programs for which their source code is lost. However, whenever the source code is available, as it is the case for Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided.

As a result, integrating security into software is becoming a very challenging and interesting domain of research. Very few concepts and approaches emerged in the literature to help and guide developers to integrate security into software. The most prominent proposals could be classified into: Security patterns, secure coding and security injection using aspect oriented programming. In this context, the main intent of our research is to create methods and solutions to integrate systematically security models and components into open source software. More recently, several proposals have been advanced for code injection, via an aspect oriented computational style, into source code for the purpose of improving its security. This concept seems to be the most relevant to integrate security into open source software. Our proposition, introduced in [23], is based on aspect-oriented programming (AOP) and inspired by the best and most relevant methods and methodologies available in the literature, in addition to elaborating valuable techniques that permit us to provide a framework for systematic security hardening.

The main components of our approach are the security hardening plans and patterns that provide an abstraction over the actions required to improve the security of a program. They should be specified and developed using an abstract, programming language independent and aspect-oriented (AO) based language. The current AO languages, however, lack many features needed for systematic security hardening. They are programming language dependent and could not be used to write and specify such high level plans and patterns, from which the need to elaborate a language built on top of them to provide the missing features. In this context, we also propose a language called *SHL* for security hardening plans and patterns specification. It allows the developer to specify high level security hardening plans that leverage priori defined security hardening patterns. These patterns, which are also developed using *SHL*, describe the steps and actions required for hardening, including detailed information on how and where to inject the security code.

This paper provides our new contributions in developing our security hardening framework. The experimental results presented together with the security hardening plans and patterns, which are elaborated using *SHL*, explore the efficiency and relevance of our approach. The remainder of this paper is organized

as follows. In Section 2, we review the contributions in the field of security patterns, secure programming and practices and AOP for securing software. Afterwards, in Section 3, we summarize our approach for systematic security hardening. Then, in Section 4, we present the syntax and semantics of *SHL*. After that, in Section 5, we illustrate the useability of the approach and *SHL* language into case studies for different security issues and problems. Finally, we offer concluding remarks in Section 6.

2 Background and Related Work

Our approach constitutes an organized framework that provides a language and methodologies for the improvement of security at all levels of the software systems. As such, we present in the sequel an overview of the current literature on the approaches that may be useful for integrating security into software, and thus guide us in developing our security hardening framework.

2.1 Security Patterns

A security pattern describes a particular recurring security problem that arises in a specific context and presents a well-proven generic scheme for a security solution [26]. Such patterns are based on the security engineering concept, which aims at considering security early during the development life cycle of software. Many of them are available in order to guide software engineers in designing their security models and securing their applications at the architecture and design phase. When it comes to security hardening, these proposed security patterns are not relevant. The reason is that we are dealing with already developed applications and software that are, in many cases, deployed. However, their method of presenting and explaining the solutions seems interesting. We are inspired by their method to present our security hardening solutions under the form of security hardening patterns. Moreover, the published security patterns may help security experts while developing the security solutions and then elaborating their corresponding hardening patterns. In the sequel, we summarize briefly in the following the content of the related publications.

In [38], Yoder and Barcalow introduced a 7-pattern catalog. In fact, their proposed patterns were not meant to be a comprehensive set of security patterns, rather just as starting point towards a collection of patterns that can help developers to address security issues when developing applications. Kienzle et al. [19, 20] have created a 29-pattern security pattern repository, which categorized security patterns as either structural or procedural patterns. Structural patterns are implementable patterns in an application whereas procedural patterns are patterns that were aimed to improve the development process of security-critical software. The presented patterns were implementations of specific web application security policies. Romanosky [25] introduced another set

of design patterns. The discussion however has focused on architectural and procedural guidelines more than security patterns.

Brown and Fernandez [14] introduced a single security pattern, the authenticator, which described a general mechanism for providing identification and authentication to a server from a client. Although authentication is a very important feature of secure systems, the pattern, as was described, was limited to distributed object systems. Fernandez and Warrier extended this pattern recently in [15], although it remains similarly limited. Braga et al. [6] also investigated security-related patterns specialized for cryptographic operations. They showed how cryptographic transformation over messages could be structured as a composite of instantiations of the cryptographic meta-pattern. The Open Group [3] has possibly introduced the most mature design patterns so far. Their catalog proposes 13 patterns, and is based on architectural framework standards such as the ISO/IEC 10181 family. The most recent work in this domain is from Schumacher et al. [27]. They offered a list of forty-six patterns applied in different fields of software security, although most of them are rewriting of previously proposed patterns.

2.2 Secure Coding

The secure coding approach presents either safe programming techniques, or a list of programming errors together with their corresponding solutions. For instance, several publications compiled common errors and vulnerabilities in code production languages such as C/C++. Although their intent is to instruct developers to avoid these errors while implementing their new software, such proposals may also help to correct the errors causing vulnerabilities into already developed software. However, these secure coding practices are always applied manually in the code and our aim is actually to elaborate a systematic, and even preferably automatic approach for security hardening. On the other hand, such practices may be helpful for security experts while developing the security solutions and then elaborating their corresponding hardening patterns. In this context, we summarize briefly the related publications.

On the topic of secure programming of C programs, developers are offered a good selection of useful and highly relevant material. One of the newest and most useful additions is from Seacord [28], which offers in-depth explanations on the nature of all known low-level security vulnerabilities in C and C++. Another common reference is from Howard and Leblanc [16], and includes all the basic security problems and solutions, as well as code fragments of functions allowing to safely implement certain operations (such as safe memory wiping). The authors also describe high-level security issues, threat modeling, access control, etc. Slides from Bishop, in addition to his book [1] , provide a comprehensive view on information assurance, as well as security vulnerabilities in C. In addition, he provides some hints and practices to solve some existing security issues. Wheeler [36] offers the widest-reaching book on system security available online. He covers operating system security, safe temporary files, cryptography,

multiple operating platforms, spam, etc. We consider his solutions relevant to the problem of insecure temporary files.

2.3 Aspect-Oriented Programming and Security

The injection of security components into application using AOP is a relatively new programming paradigm that provides a more advanced modularization mechanism on top of the traditional object-oriented programming. AOP appears to be a promising paradigm for software security. Aspects allow to precisely and selectively define and integrate security objects, methods and events within applications, which make them interesting solutions for many security issues.

AOP / Advice-Poincut Model Aspect-Oriented Programming is a family of approaches that allow to integrate different concerns into useable software in a manner that is not possible using the classical object-oriented decomposition. The foundation of AOP is the principle of "Separation of Concerns", where issues that affect and crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed such as AspectJ [18], AspectC [8], AspectC++ [33], AspectC# [21] and the AOP version addressed for Smalltalk programming language [5]. The approach adopted by most of them is called the Pointcut-Advice model. The join points, pointcuts and advices constitute its main elements.

To develop under this paradigm, one must first determine what code needs to be injected into the basic model. This code describes the behavior of the issues that affect and crosscut the application. Each atomic unit of code to be injected is called an advice. Then, it is necessary to formulate where to inject the advice into the program. This is done by the use of pointcut expressions, which its matching criteria restricts the set of a program's join points for which the advice will be injected. A join point is an identifiable execution point in the application's code and the pointcut constitutes the constructor that designates a set of join points. The pointcut expressions typically allow to match on function calls and executions, on the control flow ulterior to a given join point, on the membership in a class, etc. At the heart of this model, is the concept of an aspect, which embodies all these elements. Examples of implemented aspects are presented in Section 5. Finally, the aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

AOP Languages There are many AOP languages that have been developed. However, these languages are used for code implementation and programming language dependent. Thus, they cannot be used to specify abstract security hardening plans and patterns, which is a requirement in our proposition. We distinguish from them AspectJ [18] built on top of the Java programming language, AspectC [8] built on top of the C programming language, AspectC++

[33] built on top of the C++ programming language, AspectC# [21] built on top of the C Sharp programming language and the AOP version addressed for Smalltalk programming language [5]. AspectJ and AspectC++ are dominant propositions in the field of AOP. Tribe [7] offers an approach based on virtual class families. This approach is not relevant for our needs, as languages such as C do not support classes. The AWED language [24] was developed for distributed applications. It also support sequences. TOSCANA [13] is a toolkit for kernel-level aop programming. It allows to modify the kernel in memory in the objective to perform autonomic computing. Their language is quite simple, but is restricted to C. The Arachne system [12], part of the OBASCO project is providing an interesting aspect-oriented language for which the sequence of events is encoded very simply in the aspect. Their approach is C-centric and works on the in-memory binary process.

AOP Approaches for Security Injection Few contributions that discuss and explore the relevance of AOP and AOP languages for integrating security code into applications have been published recently [4, 10, 17, 30, 37]. These research initiatives presented mainly as case studies, however, focus on exploring the usefulness of AOP for securing software by security experts who know exactly where each piece of code should be manually injected and/or proposing AOP languages for security. None of them proposed an approach or methodology for systematic security hardening with features similar to our proposition. We present in the following an overview on these contributions.

Cigital labs proposed an AOP language called CSAW [30, 31, 29, 35], which is a small superset of C programming language. Their work is mostly dedicated to improve the security of C programs. They presented typical aspects that defend against specific types of attacks and address local problems such as buffer overflow and data logging. These aspects were divided in the low-level and high-level categories. The low-level aspects target the problems of exploiting the environmental variables such as attacks against `Setuid` programs, the problems of format strings and variable verification that cause the buffer overflow attacks, the problems of confidentiality and communication encryption, etc. Their high level aspects address the problems of event ordering, signal race condition and type safety.

De Win et al. in [10, 9, 11, 34] discussed two aspect oriented approaches and explored their use in integrating security aspects within applications. In their first approach, the interception, they explored the need to secure all the interactions with the applications that cannot be trusted and they provided additional security measures for sensitive interactions. They used a coarse-grained alternative mechanism for interception that consists of putting an interceptor at the border of the application, where interactions are checked and approved. Their proposition is achieved by changing the software that is responsible of the external communication of the applications. Their second approach, the weaving-based AOSD, is based on a weaving process that takes two or more separate views of an application and merge them together into a single artifact

as if they are developed together. They used in this approach the Advice and Joinpoints concepts to specify the behavior code to be merged in the application and the location where this code should be injected. To validate their approach, they developed some aspects using AspecJ to enforce access control and modularize the audit and access control features of an FTP server.

In [4], Ron Bodkin surveyed the security requirements for enterprize applications and described examples of security crosscutting concerns. His main focus was on authentication and authorization. He discussed use cases and scenarios for these two security issues and he explored how their security rules could be implemented using AspectJ. He also outlined several of the problems and opportunities in applying aspects to secure web applications that are written in Java.

Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang et al. [17] introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. It is based on the Java Security packages JCE and JAAS. To make their aspects reusable, they left to the programmer the responsibility to specify and implement the pointcut. This approach is a useful first step, but requires the developer to be a security expert who knows exactly where each piece of code should be injected. Moreover, its goal is to prove the feasibility of reusing and integrating pre-built aspects.

Shlowikowski and Ziekinski discussed in [32] some security solutions based on J2EE and JBoss application server, Java Authentication and Authorization service API (JAAS) and Resource Access Decision Facility (RAD). These solutions are implemented in AspectJ. They explored in their paper how the code of the aforementioned security technologies could be injected and weaved in the original application.

3 Security Hardening Approach

This section illustrates a summary of our whole approach for systematic security hardening and also explores the need and usefulness of *SHL* to achieve our objectives. We defined in [22] software security hardening as *any process, methodology, product or combination thereof that is used to add security functionalities and/or remove vulnerabilities or prevent their exploitation in existing software*. Security hardening practices are usually applied manually by injecting security code into the software [2, 16, 28, 36]. This task entails that the software architects have deep knowledge of the inner working of the software code, which is not available all the time. In this context, we elaborated an approach based on aspect orientation to perform security hardening in a systematic way. The approach architecture is illustrated in Figure 1.

Each component participates by playing a role and/or providing functionalities in order to have a complete security hardening process. The developer is the person responsible of writing plans by deriving them from the security

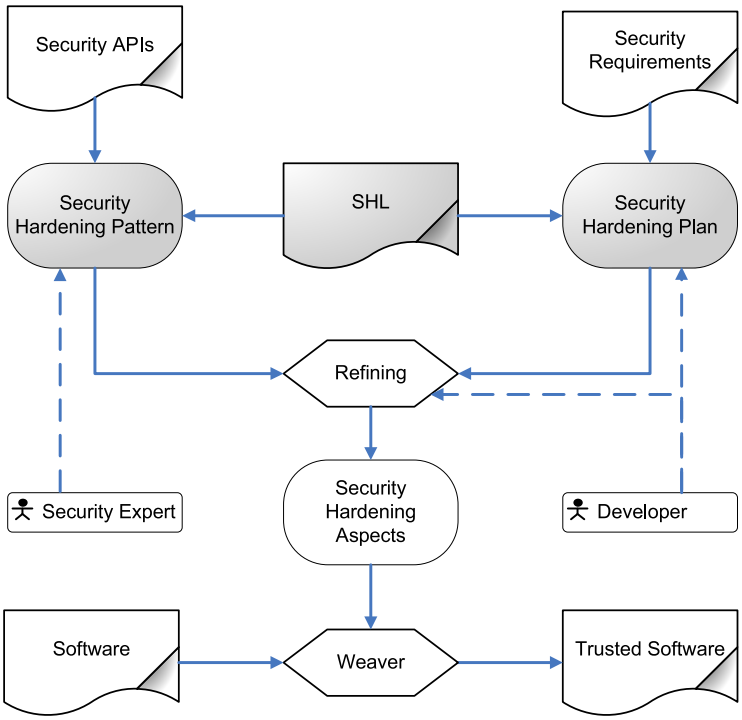

Fig. 1. Framework Architecture

requirements. These plans contains the abstract actions required for security hardening and uses the security hardening patterns that are developed by security experts and provided in a catalog. The security APIs constitute the building blocks used by the patterns to achieve the desired solutions. The *SHL* language is used to define and specify the security hardening plans and patterns.

The primary objective of this approach is to allow the developers to perform security hardening of open source software by applying well-defined solutions and without the need to have expertise in the security solution domain. At the same time, the security hardening should be applied in an organized and systematic way in order not to alter the original functionalities of the software. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build and develop the solutions. The developers are able to specify the hardening plans that use and instantiate the security hardening patterns using the proposed language *SHL*. We define security hardening patterns as well-defined solutions to known security problems, together with detailed information on how and where to inject each component of the solution into the application. The combination of hardening plans and patterns constitutes the concrete security hardening solutions.

The abstraction of the hardening plans is bridged by concrete steps defined in the hardening patterns using also *SHL*. This dedicated language, together with a well-defined template that instantiates the patterns with the plan's given parameters, allow to specify the precise steps to be performed for the hardening, taking into consideration technological issues such as platforms, libraries and languages. We built *SHL* on top of the current AOP languages because we believe, after a deep investigation on the nature of security hardening practices and the experimental results we got, that aspect orientation is the most natural and appealing approach to reach our goal.

Once the security hardening solutions are built, the refinement of the solutions into aspects or low level code can be performed using a tool or by programmers that do not need to have any security expertise. Afterwards, an AOP weaver (e.g. AspectJ, AspectC++) can be executed to harden the aspects into the original source code, which can now be inspected for correctness. As a result, the approach constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them, and allows the software engineers to use these solutions to harden open source software by specifying and developing high level security hardening plans. We illustrated the feasibility of the whole approach by elaborating several security hardening solutions that are dealing with security requirements such as securing connections, adding authorization, encrypting some information in the memory and remedying low level security vulnerabilities.

4 *SHL* Description

Our proposed language, *SHL*, allows the description and specification of security hardening patterns and plans that are used to harden systematically security into the code. It is a minimalist language built on top of the current AOP technologies that are based on advice-pointcut model. It can also be used in conjunction with them since the solutions elaborated in *SHL* can be refined into a selected AOP language (e.g. AspectC++) as illustrated in Section 5. We developed part of *SHL* with notations and expressions close to those of the current AOP languages but with all the abstraction needed to specify the security hardening plans and patterns. These notations and expressions are programming language independent and without referring to low-level implementation details. The following are the main features provided by *SHL*:

- Automatic code manipulation such as code addition, substitution, deletion, etc.
- Specification of particular code join points where security code would be injected.
- Modification of the code after the development life cycle since we are dealing with already existing open source software.

- Modification of the code in an organized way and without altering its functional attributes.
- Description and specification of security.
- Dedicated to describe and specify reusable security hardening patterns and plans.
- Parameterized language to allow the instantiation of the security hardening patterns through the security hardening plans.
- Programming language independent.
- Highly expressive and easy to use by security non experts.
- Intermediary abstractness between English and programming languages.
- Easily convertible to available AOP languages (e.g. AspectJ and AspectC++).

4.1 Grammar and Structure

In this section, we present the syntactic constructs and their semantics in *SHL*. Table 1 illustrates the `BNF` grammar of *SHL*. The language that we arrived at can be used for both plans and patterns specification, with a specific template structure for each of them. Examples of security hardening plans and patterns are elaborated using *SHL* and presented in Section 5.

Hardening Plan Structure A hardening plan starts always with the keyword `Plan`, followed by the plan's name and then the plan's code that starts and ends respectively by the keywords `BeginPlan` and `EndPlan`. Regarding the plan's code, it is composed of one or many pattern instantiations that allow to specify the name of the pattern and its parameters, in addition to the location where it should be applied. Each pattern instantiation starts with the keyword `PatternName` followed by a name, then the keyword `Parameters` followed by a list of parameters and finally by the keyword `Where` followed by the module name where the pattern should be applied (e.g. file name).

Hardening Pattern Structure A hardening pattern starts with the keyword `Pattern`, followed by the pattern's name, then the keyword `Parameters` followed by the matching criteria and finally the pattern's code that starts and ends respectively by the keywords `BeginPattern` and `EndPattern`. The matching criteria are composed of one or many parameters that could help in distinguishing the patterns with similar name and allow the pattern instantiation. The pattern code is based on AOP and composed of one or many `Location_Behavior` constructs. Each one of them constitutes the location identifier and the insertion point where the behavior code should be injected, the optional primitives that may be needed in applying the solution and the behavior code itself. A detailed explanation of the components of the pattern's code will be illustrated in Section 4.2.

4.2 Semantics

In this Section, we present the semantics of the important syntactic constructs in *SHL* language.

Start	::=	*SH_ Plan*	
	|	*SH_ Pattern*	
SH_ Plan	::=	`Plan`	*Plan_ Name*
		SH_ Plan_ Code	
Plan_ Name	::=	*Identifier*	
SH_ Plan_ Code	::=	`BeginPlan`	
		*Pattern_ Instantiation**	
		`EndPlan`	
Pattern_ Instantiation	::=	`PatternName`	*Pattern_ Name*
		(`Parameters`	*Pattern_ Parameter**)?
		`Where`	*Module_ Identification+*
Pattern_ Name	::=	*Identifier*	
Pattern_ Parameter	::=	*Parameter_ Name* =	*Parameter_ Value*
Parameter_ Name	::=	*Identifier*	
Parameter_ Value	::=	*Identifier*	
Module_ Identification	::=	*Identifier*	
SH_ Pattern	::=	`Pattern`	*Pattern_ Name*
		Matching_ Criteria?	
		SH_ Pattern_ Code	
Matching_ Criteria	::=	`Parameters`	*Pattern_ Parameter+*
SH_ Pattern_ Code	::=	`BeginPattern`	
		*Location_ Behavior**	
		`EndPattern`	
Location_ Behavior	::=	*Behavior_ Insertion_ Point+*	*Location_ Identifier+*
		Primitive?*	
		Behavior_ Code	
Behavior_ Insertion_ Point	::=	`Before`	
	|	`After`	
	|	`Replace`	
Location_ Identifier	::=	`FunctionCall <Signature>`	
	|	`FunctionExecution <Signature>`	
	|	`WithinFunction <Signature>`	
	|	`CFlow <Location_Identifier>`	
	|	`GAflow <Location_Identifier>`	
	|	`GDFlow <Location_Identifier>`	
	|	...	
Signature	::=	*Identifier*	
Primitive	::=	`ExportParameter <Identifier>`	
	|	`ImportParameter <Identifier>`	
	|	...	
Behavior_ Code	::=	`BeginBehavior`	
		Code_ Statement	
		`EndBehavior`	

Table 1. Grammar of *SHL*

Pattern_ Instantiation Specifies the name of the pattern that should be used in the plan and all the parameters needed for the pattern. The name and parameters are used as matching criteria to identify the selected pattern. The module where the pattern should be applied is also specified in the `Pattern_Instantiation`. This module can be the whole application, file name, function name, etc.

Matching_ Criteria Is a list of parameters added to the name of the pattern in order to identify the pattern. These parameters may also be needed for the solutions specified into the pattern.

Location_ Behavior Is based on the advice-pointcut model of AOP. It is the abstract representation of an aspect in the solution part of a pattern. A pattern may include one or many `Location_Behavior`. Each `Location_Behavior` is composed of the `Behavior_Insertion_Point`, `Location_Identifier`, one or many `Primitive` and `Behavior_Code`.

Behavior_ Insertion_ Point Specifies the point of code insertion after identifying the location. The `Behavior_Insertion_Point` can have the following three values: `Before`, `After` or `Replace`. The `Replace` means remove the code at the identified location and replace it with the new code, while the `Before or After` means keep the old code at the identified location and insert the new code before or after it respectively.

Location_ Identifier Identifies the joint point or series of joint points in the program where the changes specified in the `Behavior_Code` should be applied. The list of constructs used in the `Location_Identifier` is not yet complete and left for future extensions. Depending on the need of the security hardening solutions, a developer can define his own constructs. However, these constructs should have their equivalent in the current AOP technologies or should be implemented into the weaver used. In the sequel, we illustrate the semantics of some important constructs used for identifying locations:

`FunctionCall <Signature>` Provides all the join points where a function matching the signature specified is called.

`FunctionExecution <Signature>` Provides all the join points referring to the implementation of a function matching the signature specified.

`WithinFunction <Signature>` Filters all the join points that are within the functions matching the signature specified.

`CFlow <Location_Identifier>` Captures the join points occurring in the dynamic execution context of the join points specified in the input `Location_Identifier`.

`GAflow <Location_Identifier>` Operates on the control flow graph (CFG) of a program. Its input is a set of join points defined as a `Location_Identifier` and its output is a single join point. It returns the closest ancestor join point to the join points of interest that is on all their runtime paths. In other

words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. This output is the closest common ancestor that constitutes (1) the closet common parent node of all the nodes specified in the input set (2) and through which passes all the possible paths that reach them.

`GDFlow <Location_Identifier>` Operates on the CFG of a program. Its input is a set of join points defined as a `Location_Identifier` and its output is a single join point. It returns the closest child join point that can be reached by all paths starting from the join points of interest. In other words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. This output (1) is a common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes.

The `Location_Identifier` constructs can be composed with algebraic operators to build up other ones as follows:

`Location_Identifier && Location_Identifier` Returns the intersection of the join points specified in the two constructs.

`Location_Identifier || Location_Identifier` Returns the union of the join points specified in the two constructs.

`! Location_Identifier` Excludes the join points specified in the construct.

Primitive Is an optional functionality that allows to specify the variables that should be passed between two `Location_Identifier` constructs. The following are the constructs responsible of passing the parameters:

`ExportParameter <Identifier>` Defined at the origin `Location_Identifier`. It allows to specify a set of variables and make them available to be exported.

`Importparameter <Identifier>` Defined at the destination `Location_Identifier`. It allows to specify a set of variables and import them from the origin `Location_Identifier` where the `ExportParameter` has been defined.

Behavior_ Code May contain code written in any language, programming language, or even English as instructions to follow, depending on the abstraction level of the pattern. The choice of the language and syntax is left to the security hardening pattern developer. However, the code provided should be abstract and at the same time clear enough to allow a developer to refine it into low level code without the need to high security expertise. Example of such code behavior is presented in Listing 2.

4.3 Compiler and Tool

We implemented the `BNF` specification of *SHL* using ANTLR V3 Beta 6 and its associated ANTLRWorks development environment. The generated Java code

allows to parse hardening plans and patterns and verify the correctness of their syntax. We built on top of it a compiler that uses the information provided by the parser to build first its data structure, then reacts upon the provided values in order to run the hardening plans and compile and run the specified pattern and its corresponding aspect. Moreover, we integrated this compiler into a development graphical user interface for security hardening. The resulting system provides the user with graphical facilities to develop, compile, debug and run security hardening plans and patterns. It allows also to visualize the software to be hardened and all the compilation and integration activities performed during the hardening. Figure 2 shows a screenshot of this system where we can see a plan running and a pattern compiling together with the software to be hardened. The compilation process is divided into many phases that are performed consequently and systematically. The success of one phase leads automatically to execute the next one. In the sequel, we present and explain these phases.

Plan Compilation This phase constitutes of parsing the plan, verifying its syntax correctness and building the data structure required for the other compilation phases. Any error during the execution of this phase stops the whole compilation process and provide the developer with information to correct. This applies also on all the other phases.

Pattern Compilation and Matching A search engine has been developed to find the pattern that matches the pattern instantiations requested in the hardening plan (i.e. pattern name and parameters). A naming convention composed of the pattern name and parameters has been adopted to differentiate between the patterns with same name but different parameters. For instance, a pattern for authentication for Java will be named *AuthenticationJava.SHL*, while another one for C++ will be named *AuthenticationCPP.SHL*. Once the pattern matching the criteria is found, another check on the name and parameters specified inside the pattern is applied in order to ensure that the matching is correct and there is no error in the naming procedure. This includes automatically parsing and compiling the pattern contents to check the correctness of its syntax, verify the matching result and build the data structure required for the running process.

Aspect Matching Once the pattern is compiled successfully, a search engine similar to the aforementioned one is used to find the aspect corresponding to the matched pattern. However, the additional verification performed in pattern matching is not required here because the aspect will have exactly the same name of the pattern but with different extension depending on the weaver adopted.

Plan Running and Weaving Plan running is the last phase of the compilation process. Once the corresponding aspect is matched, the execution command is constructed based on the information provided in the data structure, which is built during the previous compilation phases. Afterwards, the aspect is weaved

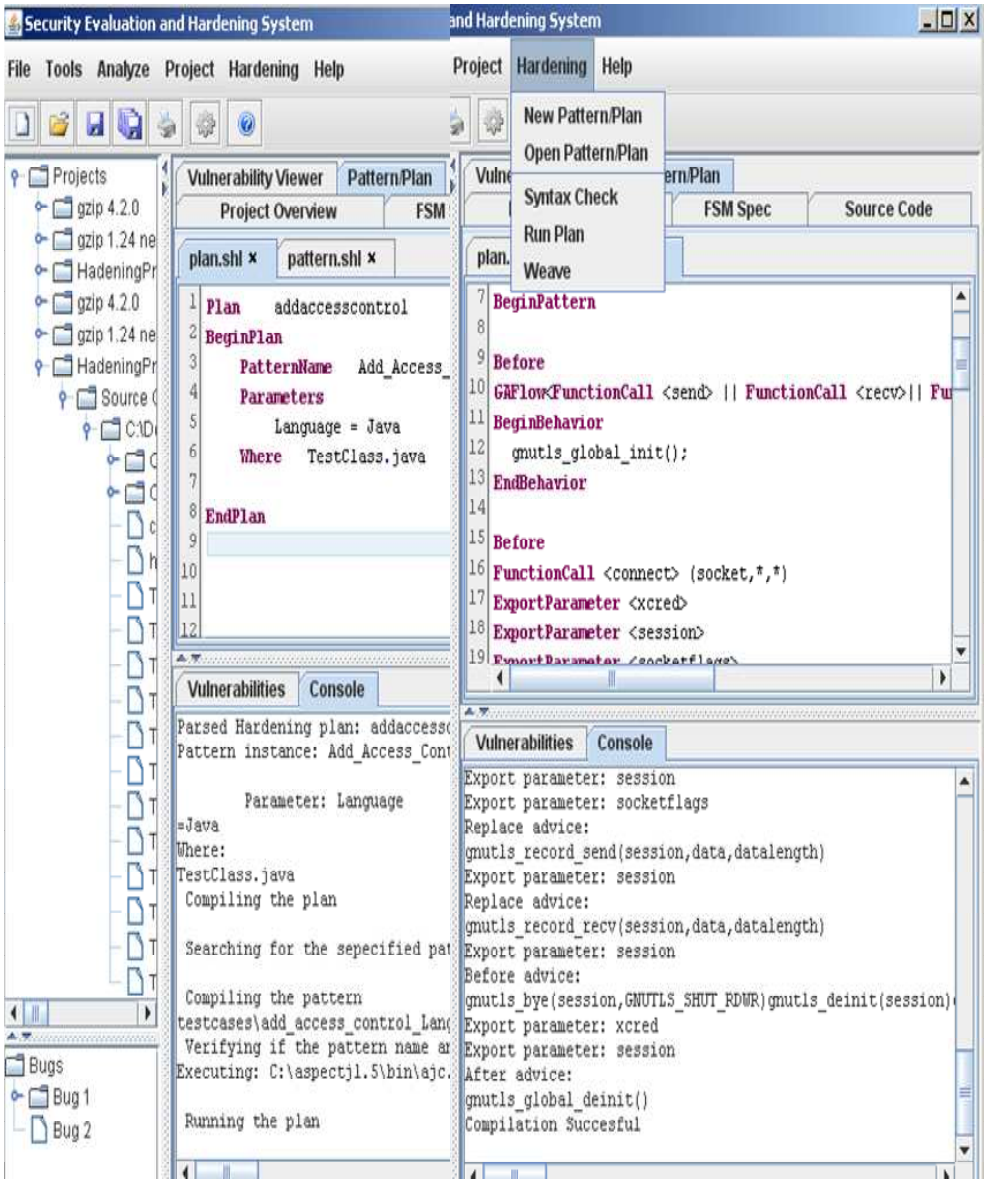

Fig. 2. Screenshot of the Security Hardening System

with the specified application or module and the resulted hardened software is produced. If the security hardening is applied on one or more modules of a bigger software, this module should be integrated in the original software that requires to be re-built for the hardening to take place.

5 Security Hardening Case Studies Using *SHL*

We demonstrated the feasibility of our approach and language for systematic security hardening by developing case studies that deal with security requirements such as securing connections, adding authorization, encrypting some information in the memory and remedying low level security vulnerabilities and applying them to developed and selected applications. During the course of our study, we developed plans, patterns, aspects in AspectC++ and AspectJ, utility functions in C and Java and example code that implement security hardening of the cases described previously. We will show some of our findings here.

5.1 Securing Connection of Client Applications

An important issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission or session hijacking. In this section, we illustrate our elaborated solutions for securing the connections of client applications by following the approach's methodology and using the proposed *SHL* language. In this context, we developed our own client application and selected an open source software called APT to secure their connections using GnuTLS/SSL library. Our application, which is a client implemented in C, allows to connect and exchange data with a selected server, typically an HTTP request. We implemented this program multiple times, with different internal structure, in order to ensure the flexibility and correctness of our hardening solution and cover as much as possible the implementation scenarios used in the current client applications.

Regarding APT, it is an automated package downloader and manager for the Debian Linux distribution. It is written in C++ and is composed of more than 23 000 source lines of code (based on version 0.5.28, generated using David A. Wheeler's 'SLOCCount'). It obtains packages via local file storage, FTP, HTTP, etc. We have decided to add HTTPS support to these two applications. In the sequel, we are going to present the hardening plans, pattern and aspect elaborated to secure the connections of APT and the aforementioned application.

***SHL* Hardening Plan** In Listing 1, we include an example of effective security hardening plans specified in *SHL* for securing the connection of the two aforementioned applications. In this case, the only difference between the two plans are the names and the modules where the patterns should be applied (i.e. the files' names specified after `Where`).

Listing 1. *SHL* Hardening Plans for Securing Connection

```
Plan     APT_Secure_Connection_Plan
BeginPlan
    PatternName    Secure_Connection_Pattern
    Parameters
            Language = C/C++
            API      = GNUTLS
            Peer     = Client
            Protocol = SSL
    Where    http.cc connect.cc
EndPlan

Plan     Own_Secure_Connection_Plan
BeginPlan
    PatternName    Secure_Connection_Pattern
    Parameters
            Language = C/C++
            API      = GNUTLS
            Peer     = Client
            Protocol = SSL
    Where    ourapplication1.c ourapplication2.c
EndPlan
```

***SHL* Hardening Pattern** Listing 2 presents the pattern elaborated in *SHL* for securing the connection of the two aforementioned application using GnuTLS/SSL. The code of the functions used in the `Code_Behavior` parts of the pattern is illustrated in Listing 3. It is expressed in C++ because our applications are implemented in this programming language. However, other syntax and programming languages can also be used depending on the abstraction required and the implementation language of the application to harden. To generalize our solution and make it applicable on wider range of applications, we assume that not all the connections are secured, since many programs have different local interprocess communications via sockets. In this case, all the functions responsible of sending receiving data on the secure channels are replaced by the ones providing TLS. On the other hand, the other functions that operate on the non-secure channels are kept untouched. Moreover, we suppose that the connection processes and the functions that send and receive the data are implemented in different components. This required additional effort to develop additional components that distinguish between the functions that operate on secure and non secure channels and export parameters between different places in the applications.

Listing 2. *SHL* Hardening Pattern for Securing Connection

```
Pattern Secure_Connection_Pattern
Parameters
     Language = C/C++
     API      = GNUTLS
     Peer     = Client
     Protocol = SSL
BeginPattern

Before
FunctionExecution <main> //Starting Point
BeginBehavior
    // Initialize the TLS library
```

```
    InitializeTLSLibrary;
EndBehavior

Before
FunctionCall <connect> //TCP Connection
ExportParameter <xcred>
ExportParameter <session>
BeginBehavior
    // Initialize the TLS session resources
    InitializeTLSSession;
EndBehavior

After
FunctionCall <connect>
ImportParameter <session>
BeginBehavior
    // Add the TLS handshake
    AddTLSHandshake;
EndBehavior

Replace
FunctionCall <send>
ImportParameter <session>
BeginBehavior
    // Change the send functions using that
    // socket by the TLS send functions of the
    // used API when using a secured socket
    SSLSend;
EndBehavior

Replace
FunctionCall <receive>
ImportParameter <session>
BeginBehavior
    // Change the receive functions using that
    // socket by the TLS receive functions of
    // the used API when using a secured socket
    SSLReceive;
EndBehavior

Before
FunctionCall <close> //Socket close
ImportParameter <xcred>
ImportParameter <session>
BeginBehavior
    // Cut the TLS connection
    CloseAndDealocateTLSSession;
EndBehavior

After
FunctionExecution <main>
BeginBehavior
    // Deinitialize the TLS library
    DeinitializeTLSLibrary;
EndBehavior

EndPattern
```

Listing 3. Functions Used in Secure Connection Pattern

```
InitializeTLSLibrary
  gnutls_global_init();

InitializeTLSSession
  gnutls_init (session, GNUTLS_CLIENT);
  gnutls_set_default_priority (session);
```

```
  gnutls_certificate_type_set_priority (session, cert_type_priority);
  gnutls_certificate_allocate_credentials(xcred);
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

AddTLSHandshake
  gnutls_transport_set_ptr(session, socket);
  gnutls_handshake (session);

SSLSend
  gnutls_record_send(session, data, datalength);

SSLReceive
  gnutls_record_recv(session, data, datalength);

CloseAndDealocateTLSSession
  gnutls_bye(session, GNUTLS_SHUT_RDWR);
  gnutls_deinit(session);
  gnutls_certificate_free_credentials(xcred);

DeinitializeTLSLibrary
  gnutls_global_deinit();
```

Hardening Aspect We refined and implemented (using AspectC++) in Listing 4 the corresponding aspect of the pattern presented in Listing 2. The reader will notice the appearance of `hardening_sockinfo_t`. These are the data structure and functions that we developed to distinguish between secure and non secure channels and export the parameter between the application's components at runtime (since the primitives `ImportParamter` and `ExportParameter` are not yet deployed into the weavers). We found that one major problem was the passing of parameters between functions that initialize the connection and those that use it for sending and receiving data. In order to avoid using shared memory directly, we opted for a hash table that uses the socket number as a key to store and retrieve all the needed information (in our own defined data structure). One additional information that we store is whether the socket is secured or not. In this manner, all calls to a `send()` and `recv()` are modified for a runtime check that uses the proper sending/receiving function.

Listing 4. Excerpt of Aspect for Securing Connections

```
aspect SecureConnection {

advice execution ("% main(...)") : around () {
    /*Initialization of the API*/
    /*...*/
    tjp->proceed();
    /*De-initialization of the API*/
    /*...*/
    *tjp->result() = 0;
}

advice call("% connect(...)") : around () {
    //variables declared
    hardening_sockinfo_t socketInfo;
    const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP,
          0};

    //initialize TLS session info
    gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
    /*...*/
```

```
    //Connect
    tjp->proceed();
    if(*tjp->result()<0) {return;}

    //Save the needed parameters and the information that distinguishes
        between secure and non-secure channels
    socketInfo.isSecure = true;
    socketInfo.socketDescriptor = *(int*)tjp->arg(0);
    hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);

    //TLS handshake
    gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr) (*(
        int*)tjp->arg(0)));
    *tjp->result() = gnutls_handshake (socketInfo.session);
}

//replacing send() by gnutls_record_send() on a secured socket
advice call("% send(...)") : around () {
    //Retrieve the needed parameters and the information that distinguishes
        between secure and non-secure channels
    hardening_sockinfo_t socketInfo;
    socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

    //Check if the channel, on which the send function operates, is secured
        or not
    if (socketInfo.isSecure)
       *(tjp->result())  = gnutls_record_send(socketInfo.session, *(char**)
           tjp->arg(1), *(int *)tjp->arg(2));
    else
       tjp->proceed();
}
};
```

Experimental Results In order to validate the hardened applications, we used the Debian apache-ssl package, an HTTP server that accepts only SSL-enabled connections. We populated the server with a software repository compliant with APT's requirements, so that APT can connect automatically to the server and download the needed metadata in the repository. Then, we weaved (using AspectC++ weaver) the elaborated aspect with the different variants of our application and APT. We first executed our own hardened application and made it connect successfully to our local HTTPS-enabled web server using HTTPS. Afterwards, we iterated over many connections to a remote web server, and a few index page's bytes were successfully retrieved. We also applied the pattern for securing connections into APT. The resulting hardened software was capable of performing both HTTP and HTTPS package acquisition, based on the parameters in the configuration file. After building and deploying the modified APT package, we tested successfully its functionality by refreshing APT's package database, which forced the software to connect to both our local web server (Apache-ssl) using HTTPS and remote servers using HTTP to update its list of packages. The experimental results in Figures 3, 4 and 5.1 show that the new secure applications are able to connect using both HTTP and HTTPS connections, exploring the correctness of the security hardening process.

In the sequel, we provide brief explanations of our results. Figure 3 shows the packet capture, obtained using WireShark software, of the unencrypted

HTTP traffic between our version of APT and its remote package repositories. The highlighted line shows an HTTP connection to the `www.getautomatix.com` APT package repository. On the other hand, Figure 4 shows the connections between our version of APT and the remote package repositories on the local web server. The highlighted lines show TLSv1 application data exchanged in encrypted form through HTTPS connections, exploring the correctness of the security hardening process. Moreover, Figure 5.1 shows an extract of Apache's access log, edited here for conciseness, where the package metadata was successfully obtained from our local server by the applications.

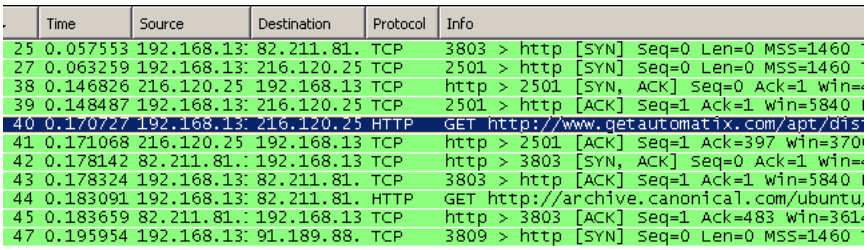

Fig. 3. Packet Capture of Unencrypted APT Traffic

5.2 Adding Authorization

Adding authorization is a problem of authorizing or denying access to a resource or operation (i.e. Access control). It requires to know which principal is interacting with the application, and what are its associated rights. In this section, we illustrate our elaborated solutions for adding authorization to applications by following the approach's methodology and using the proposed *SHL* language. In this context, we developed our own Java application, in which we decided to add authorization check on some of its methods. We implemented this program multiple times, with different internal structure, in order to ensure the flexibility and correctness of our hardening solution and cover as much as possible the implementation scenarios used in the current Java applications.

***SHL* Hardening Plan** In Listing 5, we include an example of effective security hardening plans specified in *SHL* for adding authorization into the aforementioned application.

Listing 5. *SHL* Hardening Plans for Adding Authorization

```
Plan     Own_Add_Authorization_Plan
BeginPlan
    PatternName
    Parameters
            Language = Java
            API      = JAAS
            Type     = ACL
```

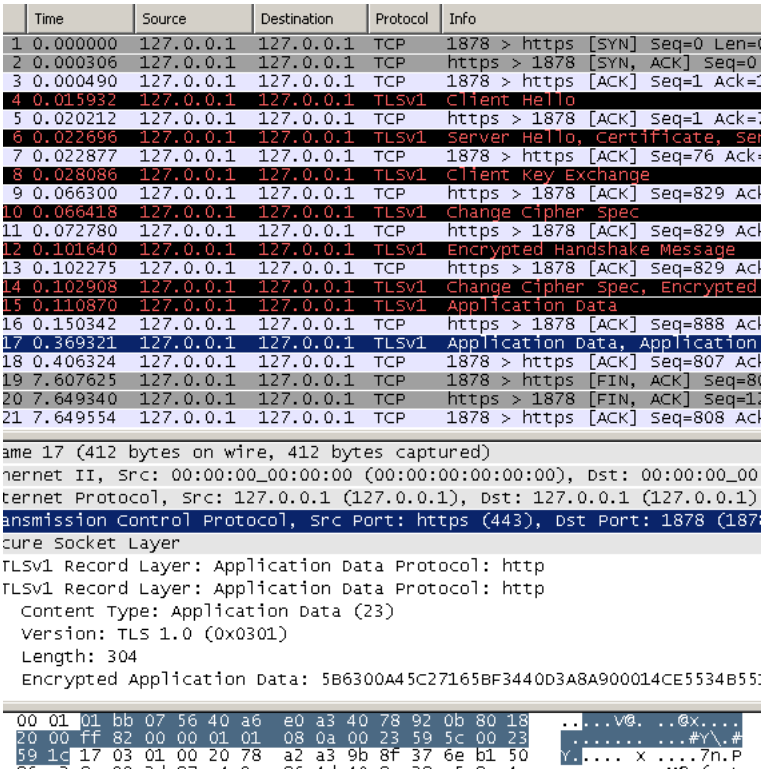

Fig. 4. Packet Capture of SSL-protected APT Traffic

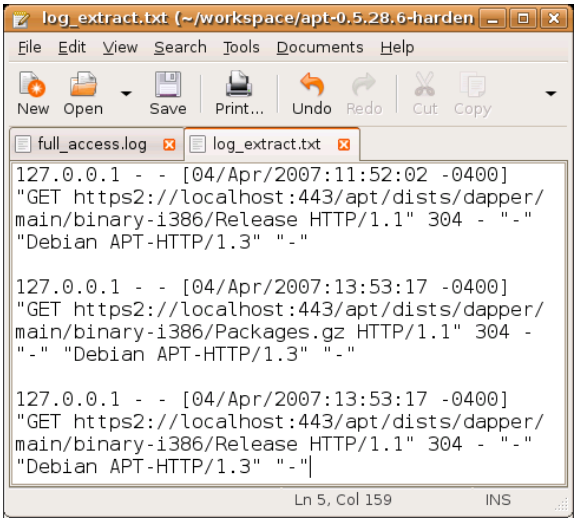

Fig. 5. Excerpt of Apache Access Log

```
    Where    test.java
EndPlan
```

***SHL* Hardening Pattern** Listing 6 describes the hardening pattern elaborated in *SHL* for adding authorization to the aforementioned application. The Java code of the functions used in the `Code_Behavior` parts of the pattern is illustrated in Listing 7. Its usage scenario assumes that interface changes are undesirable and that a policy is specified and loaded separately from what programmers can directly specify (which is the case for technologies like Java). It requires some forms of authentication in order to have the working user credentials that are used in the access control decisions.

Listing 6. *SHL* Hardening Pattern for Adding Authorization

```
Pattern Add_Authorization_Pattern
Parameters
     Language = Java
     API      = JAAS
     Typte    = ACL
BeginPattern

Before
FunctionExecution <dosomething>
BeginBehavior
    //Get the user name of
    GetUserName;
    //Get the permission name of the matched methods
    GetMethodPermissionName;
    //Check this username has permission to access the method
    CheckPermission;
EndBehavior

EndPattern
```

Listing 7. Functions Used in Add Authorization Pattern

```
GetUserName
  Subject subject = (Subject) subjects.get( System.getProperty (" user.name
      "));

GetMethodPermissionName
  String permissionName = (thisJoinPoint.getSignature()).
      getDeclaringTypeName().concat(".".concat((thisJoinPoint.getSignature()
      ).getName()));

CheckPermsion
  AuthPermission perm = new AuthPermission (permissionName );
  perm.checkGuard (null);
```

Hardening Aspect We have implemented an example of access control as an AspectJ aspect (see Listing 8) that uses Java Authentication and Authorization service API (JAAS) for authorization. The rights are specified in a policy file, which is not included here. We assume a local login, in this case, and we obtain the user name from the virtual machine. The permissions are specified in the format `package.class.function`.

Listing 8. Excerpt of an Aspect for Adding Authorization

```
public aspect AddAccessControl {

  protected static Hashtable subjects = new Hashtable();
  abstract class Action implements PrivilegedExceptionAction{};

  pointcut test(): call(void doSomething());

  String getPermissionName(Signature sig){
    return sig.getDeclaringTypeName().concat(".".concat(sig.getName()));
  }

  void around(): test(){
    try{
      //get the Subject instance based on the current user name
       Subject subject = (Subject) subjects.get(System.getProperty("user.
            name"));

        //anonymous inner class for the privileged action
        //however, we should have them static to avoid unnecessary overhead
        PrivilegedExceptionAction action = new Action()
        {
          public Object run() throws Exception{
            String permissionName = getPermissionName(thisJoinPoint.
                 getSignature());
            AuthPermission perm = new AuthPermission(permissionName);
            perm.checkGuard(null); //throws exception if not having
                 permission
            proceed();//execute the original code that way
            return null;
          }

        };

        // Enforce Access Controls
        Subject.doAs(subject, action);
    }catch (Exception e){e.printStackTrace();}
  }

}
```

Experimental Results We applied verification on the functional and security correctness of the hardened application. This task has been performed by either adding or removing the access right to execute the target method in the policy file. The practical impact of removing the right and then executing the method threw an access right violation exception by the Java virtual machine, which illustrates the correctness of the authorization deployed.

6 Conclusion

We presented in this paper our accomplishment towards developing a framework for systematic security hardening. This framework, which is based on AOP, illustrates our proposition and methods that allow developers to perform the security hardening of software in a systematic way and without the need to have expertise in the security solution domain. At the same time, it allows the security experts to provide the best solutions to particular security problems

with all the details on how and where to apply them. This is done by providing an abstraction over the actions required to improve the security of the programs and elaborating a high-level AOP language called *SHL*. This language is used for the description and specification of security hardening plans and patterns. In this context, we reviewed first the contributions in the field of security patterns, secure programming and practices and AOP for securing software. Afterwards, we summarized our security hardening approach for systematic security hardening and presented the syntax and semantics of *SHL*. Finally, we explored the experimental results and illustrated the efficiency and relevance of the approach and *SHL* language into case studies for several security issues.

Regarding our future work, we are currently working on extending *SHL* and improving the compiler, applying our solutions on wider range of open source software, building other case studies and developing solutions for several security issues.

References

1. Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.
2. Matt Bishop. How Attackers Break Programs, and How to Write More Secure Programs, 2005. `http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html` (accessed 2007/04/19).
3. Bob Blakley, Craig Heath, and members of The Open Group Security Forum. Security design patterns. Technical Report G031, Open Group, 2004.
4. Ron Bodkin. Enterprise security aspects. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.
5. K. Bollert. On weaving aspects. In *International Workshop on Aspect-Oriented Programming at ECOOP99*, 1999.
6. Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. Tropyc: A pattern language for cryprographic software. Technical Report IC-99-03, Institute of Computing, UNICAMP, January 1999.
7. Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 121–134, New York, NY, USA, 2007. ACM Press.
8. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of Foundations of software Engineering, Vienne, Austria*, 2001.
9. B. DeWin, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In *Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security: Advances in Network and Distributed Systems Security*, pages 193–194, 2001.
10. Bart DeWin. *Engineering Application Level Security through Aspect Oriented Software Development*. PhD thesis, Katholieke Universiteit Leuven, 2004.
11. Bart DeWin, Bart Vanhaute, and Bart De Decker. How aspect-oriented programming can help to build secure software. *Informatica (Slovenia)*, 26(2), 2002.

12. Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.
13. Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM Press.
14. F. Lee Brown, Jr., James DiVietri, Graziella Diaz de Villegas, and Eduardo B. Fernandez. The authenticator pattern. In *Proceedings of the 6th Annual Conference on the Pattern Languages of Programs (PLoP99)*, 1999.
15. Eduardo B. Fernandez and Reghu Warrier. Remote authenticator/authorizer. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP 2003)*, 2003.
16. Michael Howard and David E. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.
17. Minhuan Huang, Chunlei Wang, and Lufeng Zhang. Toward a reusable and generic security aspect library. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.
18. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Overview of aspectj. In *Proceedings of the 15th European Conference ECOOP 2001, Budapest, Hungary*. Springer Verlag, 2001.
19. Darrell M. Kienzle and Matthew C. Elder. Final technical report: Security patterns for web application development. Technical Report DARPA Contract # F30602-01-C-0164, 2002. `http://www.modsecurity.org/archive/securitypatterns/dmdj_final_report.pdf` (accessed 2007/04/19).
20. Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository, 2002. `http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf` (accessed 2007/04/19).
21. H. Kim. An aosd implementation for c#. Technical Report TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.
22. Azzam Mourad, Marc-André Laverdière, and M. Debbabi. Security hardening of open source software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM Press, 2006.
23. Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. Towards an aspect oriented approach for the security hardening of code. In *Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems*. IEEE Press, 2007.
24. Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM Press.
25. Sasha Romanosky. Security design patterns part 1, 2001. `http://www.romanosky.net/` (accessed in 2005).
26. Markus Schumacher. *Security Engineering with Patterns*. Springer, 2003.
27. Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

28. Robert C. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.
29. C. Shah and F. Hill. Aspect-oriented programming security framework. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 03)*, pages 143–145. IEEE Press, 2003.
30. Viren Shah. An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, 2003.
31. Viren Shah and Frank Hill. Using aspect-oriented programming for addressing security concerns. In *Procceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE)*, pages 115–119, 2002.
32. Pawel Slowikowski and Krzysztof Zielinski. Comparison study of aspect-oriented and container managed security. In *Proceedings of the ECCOP workshop on Analysis of Aspect-Oriented Software*, 2003.
33. O. Spinczyk, A. Gal, and W. chroder Preikschat. Aspectc++: An aspect-oriented extension to c++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia*, 2002.
34. B. Vanhaute and B. DeWin. Security and genericity. In *Proceedings of the 1st Belgian AOSD Workshop, Belgium, November 8*, 2001.
35. J. Viega, J.T. Bloch, and C. Pravir. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.
36. David A. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. `http://www.dwheeler.com/secure-programs/` (accessed 2007/04/19).
37. Dianxiang Xu, Vivek Goel, and Kendall Nygard. An aspect-oriented approach to security requirements analysis. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 79–82, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
38. Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs (PLoP97)*, 1997.