AN ASPECT-ORIENTED FRAMEWORK FOR SYSTEMATIC SECURITY HARDENING OF SOFTWARE

AZZAM MOURAD

A THESIS

IN

THE DEPARTMENT

OF

ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

DECEMBER 2008

Your file Votre référence
ISBN: 978-0-494-63365-6
Our file Notre référence
ISBN: 978-0-494-63365-6

ABSTRACT

An Aspect-Oriented Framework for Systematic Security Hardening of Software

Azzam Mourad, Ph.D.

Concordia University, 2008

In this thesis, we address the problems related to the security hardening of open source software. Accordingly, we first propose an aspect-oriented and pattern-based approach for systematic security hardening. It is based on the full separation between the roles and duties of the security experts and the developers performing the hardening. Such proposition constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with the details on why, how and where to apply them. Moreover, it allows the developers to use these solutions to harden open source software without the need to have high security expertise. We realize the proposed approach by elaborating a programming independent and aspect-oriented based language for security hardening called *SHL*, developing its corresponding parser, compiler and facilities and integrating all of them into a framework for software security hardening. We also illustrate the feasibility of the elaborated framework by developing several security hardening case studies that deal with known security requirements and vulnerabilities and applying them on large scale software. Second, we enrich *SHL* and the aspect-oriented languages with new pointcut and primitive constructs (*GAFlow*, *GDFlow*, *ExportParameter* and *ImportParameter*) that provide features missing in the current AOP proposals and needed for systematic security hardening concerns. We also explore the viability of the proposed pointcuts and primitives by elaborating and implementing their algorithms and presenting the result of explanatory case studies. Finally, we improve the proposed framework by proposing a new approach for applying security hardening on the *Gimple* representation of software and elaborating formal syntax for *SHL* and *Gimple* together with an operational semantics for *SHL* weaving based on *Gimple*. We realize our proposition by integrating into the *GCC* compiler few features described in the *SHL* weaving semantics and developing a demonstrative case study.

Acknowledgments

I would like to express my gratitude to Almighty GOD, the most Beneficent and the most Merciful, for granting me the ability and opportunity to complete this thesis.

I would like to thank to my supervisor, Prof. Mourad Debbabi, for his advices, ideas and effort to ensure a continuous supervision of this thesis. His insights and encouragements have had a major impact on this work, which would not be possible without his guidance and support. Working with him was a very valuable experience for me. He deserves all my acknowledgements.

I would like to thank Prof. Christine Choppy, Joey Paquet, Roch Glitho and Chadi Assi who honored me by being members of the examiner committee and reviewing this thesis. Their time and effort are greatly appreciated.

I would like thank my colleagues Hadi Otrok and Syrine Tlili who shared with me the precious years of my thesis. Also, a very special thank is due to all my TFOSS team colleagues for all their collaboration in this research. Moreover, further thanks are extended to all the members of the Computer Security Laboratory.

Finally, I am very grateful to my father, mother, brother, sisters and family members for their encouragement, love and endless support.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

1.1 Motivations and Problem Statement

The software market has been dominated by Commercial-Off-The-Shelf (COTS) products during the past two decades. These products offer a myriad of functionalities. However, their intrinsic limitations such as closed source code, expensive upgrades and lock-in effect have emerged over time. Moreover, some organizations, notably governments, require high-level of assurance for the security of systems, a need that simply may not be answered by some COTS software. The result was the development of a parallel economy based on Free and Open Source Software (FOSS). A great deal of production systems rely on FOSS for their operations, where source code is made available for use, modification and maintenance without the expensive fees imposed by COTS software vendors. Some countries are taking advantage of this openness, as it answers their need for trustworthy and validated software. Currently, a plethora of high-quality FOSS projects, that are implemented in different programming languages, mostly C, C++ and Java, are widely available for use and modification at no (or small) cost and are carried out via Internet collaboration. Many of these FOSS products are considered to be as mature as their equivalent COTS. They are now perceived as viable long-term solution that deserves careful consideration because of the potential for significant cost savings, improved reliability, and support advantages over proprietary software.

In parallel, security is taking an increasingly predominant role in today 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 box, even on programs that have not been designed with security in mind. The challenge is even greater when such systems must be adapted to networked/web environments, while they are not originally designed to fit into such high-risk environments. Tools and guidelines for security have been available for developers for few years already, but their practical adoption is limited so far. Nowadays, software developers must face the challenge of improving the security of programs and are often under-equipped to do so. In some cases, little can be done to improve the situation, especially for COTS software products that are no longer supported, or their source code is not available.

However, whenever the source code is present, as it is the case of open-source software, a wide range of security improvements could be applied once a focus on security is decided. On the other hand, the secure integration of FOSS in IT infrastructures is very demanding and requires the adoption of particular methodologies, tools and technical policies in order to reliably compose large software systems. Moreover, many of those open source software are designed without security in mind, their security models are not well developed and/or their code encloses low level security vulnerabilities, from which the need to find methodologies to improve the security of such software.

As a result, integrating security into open source software becomes a very challenging and interesting domain of research. In this context, we first define 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.* Few concepts and approaches emerged in the literature to help and guide developers to secure software. We can distinguish from them the hardening methods at the operating system and network levels, secure programming solutions published in many books and reviews [15, 50], security code injection using aspect-oriented programming (AOP) [20, 31], security design patterns [70], security patches, etc. In the sequel, we summarize and discuss briefly the propositions that can be relevant and noteworthy to build our framework for systematic security hardening of software.

Security design patterns are proposed as part of the security engineering concept, which aims at considering security early during the development life cycle of software. They are considered as a guide to improve and integrate security during the architecture and design phases. They approach the problem by encapsulating expert knowledge in the form of well-defined solutions to common security problems. Many security design patterns are available to help software engineers in designing their security models and securing their applications [18, 22, 38, 39, 56, 68, 70, 90]. Their concept of organizing and providing the solutions as patterns seems interesting and it can be adapted to be useful for security hardening of existing software.

Secure coding is another useful approach that presents either safe programming techniques, or list of programming errors together with their corresponding solutions [15, 16, 50, 73, 87]. For instance, several publications compiled common errors and vulnerabilities in code production languages such as `C` and `C++`. Their intent is to instruct software developers to avoid these errors. They may also target the security of existing software by correcting manually the programming errors causing the vulnerabilities.

More recently, few initiatives have been introduced for code injection, via an aspect-oriented computational style, into source code for the purpose of improving its security [20,31,52,74,78]. This approach is based on the idea of separating out the security concerns from the rest of the application, such that they can be addressed independently and applied globally. Aspect Oriented Programming [55,79] is a relatively new programming paradigm that provides a more advanced modularization mechanism on top of the traditional object-oriented programming (OOP). It is based on the idea that computer systems are better programmed by separately specifying the various concerns (i.e., separation of concerns), and then relying on underlying infrastructure called weaver to compose them together. The techniques in this paradigm were precisely introduced to address the development problems that are inherent to crosscutting concerns. This paradigm seems to be very promising for security hardening of code, and hence we can build on top of it to achieve systematic security hardening of software.

These propositions, together with others that address specific security issues, are likely to provide solutions to several security problems and requirements, and hence yield valuable insights to build up the security hardening solutions. However, they have important shortcomings regarding their methodologies for integrating the security solutions into software. Their solutions are applied manually and in an ad-hoc manner and require high security expertise, which contradict somehow the purpose of proposing them. None of them offer the developers well-defined methodologies, mechanisms and/or frameworks that assist and lead them during the application and integration of the security modules into software. Moreover, most of them (e.g., security design patterns and secure coding) target security issues during the development of new software, which limit to some extent their usefulness to secure already developed code.

Besides, security hardening has difficult and critical procedures. If applied manually, they often require important and significant implementation decisions to be taken by the developers, which entails high security expertise. They also require lot of time to be tackled and may create other vulnerabilities, especially when dealing with large scale software (e.g., thousands and millions lines of code). Moreover, there is always a difficulty in finding the software engineers and developers who are specialized in both the security solution domain and the software functionality domain. In fact, this is an open problem raised by several IT managers (e.g., Bell Security Labs, Ericsson Research Labs). As such, any proposition for security hardening of open source software should address all the aforementioned problems and take into consideration how to provide the hardening solutions for security problems, how to avoid the manual application of the hardening solutions and how to avoid the need to have high security expertise to apply the hardening solutions.

1.2 Objectives

The primary intent of this thesis is to create and elaborate into a framework well-defined and organized methodology, language, mechanisms, compiler and facilities needed to harden systematically and consistently security models, components and code into open source software. More specifically, our objectives are:

- Address the problems related to security hardening of open source software and elaborate a methodology for performing systematic hardening without the need to have high security expertise.
- Elaborate a dedicated language to express in a perspicuous and elegant way the security hardening components.
- Realize the elaborated methodology by designing and implementing its components and integrating them into the corresponding security hardening framework.
- Ascribe a formal specification of the elaborated security hardening framework.
- Demonstrate the relevance and usefulness of our propositions by developing several security hardening case studies and applying them on large scale software.

1.3 Approach Overview and Contributions

To pursue our objectives and solve the aforementioned related problems, we elaborated and developed an approach, a language and a framework based on aspect-oriented programming for systematic security hardening of software. We also addressed few limitations of AOP for security hardening concerns by elaborating new pointcut and primitive constructs. Moreover, we enriched our framework by elaborating a new programming language independent weaving approach and building a formal specification of weaving based on the *Gimple* representation of software. In the sequel, we discuss in details each of the aforementioned contributions.

1.3.1 Aspect-Oriented and Pattern-Based Approach for Security Hardening

The primary contribution of this work is building the needed approach and facilities that allow the developers to perform security hardening of software by applying well-defined solutions and without requiring from the developer to have expertise in the security solution domain. At the same time, the security hardening is applied in an organized and systematic way. The related contributions are:

- Proposing a methodology that provides an abstraction over the actions required to improve the security of a program and adopting AOP to develop and integrate the solutions. The developers, with no security expertise, will be able to specify the hardening plans that use and instantiate the security hardening patterns. The security hardening patterns are well-defined solutions to well-known security problems, including detailed information on how and where to inject each component of the solution into an application. The combination of hardening plans and patterns constitutes the concrete security hardening solutions.

- Elaborating a programming independent language for security hardening (*SHL*) that allows the developers to describe and specify the hardening plans and patterns needed to harden systematically security into software. It is a core language built on top of the current aspect-oriented technologies that are based on advice-poincut model. It can also be used in conjunction with them.

- Designing and implementing into a framework the parser of *SHL*, compiler and interface that realize the proposed methodology and allow specifying the plans and patterns and performing security hardening of software.

- Developing several security hardening case studies and applying them on large scale software, which demonstrate the usefulness and relevance of the proposed framework.

1.3.2 New Aspect-Oriented Constructs for *SHL* Targeting Security Concerns

The main contribution of this work is addressing two limitations of the current AOP technologies for security hardening concerns and building their corresponding solutions into *SHL*. Our experiments explored the usefulness of separating the security concerns from the other software features, then using AOP for weaving them together. On the other hand, we have also distinguished, together with other related work in the literature [21,47,54,58,61], the limitations of the available AOP technologies for few security issues. Indeed, some security hardening activities could not be applied due to such limitations. Adopting AOP into the elaborated framework makes dealing with these problems, or at least some of them, necessary to reach our objectives. The related achievements are:

- Proposing AOP pointcuts (*GAFlow* and *GDFlow*) that allow to identify particular join points in a program control flow graph (CFG), exploring their usefulness and necessity for security hardening and elaborating their corresponding algorithms.
- Proposing AOP primitives (*ExportParameter* and *ImportParameter*) that allow passing parameters between two pointcuts, exploring their usefulness and necessity for security hardening and elaborating their corresponding algorithms.

1.3.3 Formal Semantics of *SHL* Weaving

The main contribution of this work is twofold. It provides a formal specification and weaving semantics for the elaborated security hardening framework. Simultaneously, it constitutes a novel approach for applying aspect-oriented weaving into the *Gimple* representation of software. The related achievements are:

- Elaborating a formal specification of *SHL* weaving based on the *Gimple* representation of software. We built *SHL* formal syntax and formal semantics for *Gimple* weaving. This formal specification constitutes an initial attempt and a guide toward developing a complete weaver for *Gimple*. Moreover, it provides a potential model for verifying formally *SHL* security hardening solutions.

- Augmenting the security hardening framework by proposing a new aspect-oriented weaving approach for *Gimple* to be integrated into the *GCC* compiler. This approach allows compiling the security hardening pattern and applying the hardening on the *Gimple* representation (tree) of software instead of the source code. In other words, it allows bypassing the refinement of pattern into aspect for some security hardening solutions, and consequently avoids using the current AOP weavers to harden software. This provides more systematization and automation to our original approach. Moreover, this approach allows exploiting the *Gimple* intermediate representation to weave an application written in a specific programming language with code written in a different one.

- Realizing the proposed approach and semantics by implementing several described weaving capabilities, integrating them into the *GCC* compiler, and building a security hardening case study to demonstrate the usability and relevance of the proposed approach and semantics.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, we present an overview of the current approaches for securing software. We first describe the different levels of computer and information security. Then, we provide a taxonomy for software security hardening. Afterwards, we discuss the security engineering, security patterns and secure programming approaches, and highlight their relevant points and limitations for software security hardening.

In Chapter 3, we explore the relevance of AOP for security hardening. We first describe the AOP concepts, models and languages. Then, we provide AOP solutions for several security issues and present a literature review on the approaches related to this area.

In Chapter 4, we present the core approach, components, language, compiler and implementation of the systematic security hardening framework. In this context, we describe the proposed approach and the main framework components including the security hardening plans, patterns and language. We also present the grammar, structure and informal semantics of the security hardening language *SHL* and provide the compilation phases and the implementation methodology of the proposed framework. Moreover, we explore the useability and relevance of our propositions by presenting security hardening case studies for different security issues and problems and illustrating the experimental results of applying them on large scale software.

In Chapter 5, we address few limitations of the current AOP technologies and propose new pointcuts and primitives needed for security hardening concerns. In this context, we explore the limitations of the current AOP technologies for security and present the literature review related to this domain. We also describe the proposed pointcuts and primitives and illustrate into examples their necessity and usefulness for security hardening concerns. Moreover, we provide the methodology and algorithms for implementing our proposals and discuss their viability and correctness in case studies.

In Chapter 6, we provide a formal specification and semantics for the elaborated security hardening framework and propose a novel approach for applying aspect-oriented weaving into the *Gimple* representation of software. In this context, we describe the *Gimple* approach for systematic security hardening, the syntax of *SHL* and *Gimple* and the operational semantics for *Gimple* weaving. We also provide the methodology and results of implementing several *Gimple* weaving capabilities into the *GCC* compiler together with a security hardening case study demonstrating their usability and correctness.

In Chapter 7, we summarize briefly the achievements and contributions of this thesis, provide concluding remarks, state the plans for future work, and present the list of publications derived from this thesis.

Chapter 2

Techniques for Securing Software: Background and Scope

2.1 Introduction

The primary objective of this thesis is to elaborate and develop the methodology and mechanisms needed to improve and add security at the software level. As such, we first identify the scope of software security with respect to the other fields of IT security. Then, we present an overview and assessment of the current literature on the approaches that may be useful for securing software, and thus guide us in developing our security hardening framework. Accordingly, we highlight in this chapter the relevance and limitations of secure programming and security engineering using design pattern propositions. We leave the discussion about how aspect-oriented programming can be used for security code injection till the next chapter.

The rest of the chapter is organized as follows. In Section 2.2, we provide a global introduction on information and computer security. In Section 2.3, we introduce our main approach by providing a definition and a taxonomy for software security hardening. In Section 2.4, we discuss the security engineering and security patterns approaches. Similarly, we offer in Section 2.5 a discussion about the secure programming techniques. Finally, in Section 2.6, we provide concluding remarks about what is covered in this chapter.

2.2 Computer and Information Security

Information security can be defined as follows: "security regards the protection of valuable information against different kinds of threats, such as disclosure, unauthorized access and use, modification, destruction and so forth" [31]. The aforementioned valuable information can be local data, data transmitted over the networks, data related to the core functionality of the software/application, etc. Information security is mainly concerned with the confidentiality, integrity and availability of data.

Computer security is a category of information security applied to computers. The aim of computer security is to prevent attackers from achieving their objectives through unauthorized access or unauthorized use of computers, networks and/or valuable assets [50,70]. This can be achieved by taking measures and precautions against theft, espionage and/or sabotage [31]. In this context, computer security imposes requirements in the form of constraints on what computers are not supposed to do. However, information security, computer security and information assurance are interrelated and frequently used interchangeably. They share the common objectives of protecting the confidentiality, integrity and availability of information. Their main differences lie in the methodologies used and the area of concentration.

Before explaining the approaches, layers and requirements of computer security, we provide in the sequel definitions for some security concepts and explore the relations between them. This is required as many different definitions are used in the security literature, so a clarification of the relations between these concepts help to get a better understanding of the overall computer security field.

- *Asset*: An asset is an information or a resource that has value to an organization or person. Applications, systems and networks are counted as assets. The weak assets are those that have vulnerabilities.

- *Security Flaw*: A security flaw is a defect that poses a potential security risk. A software defect is the result of encoding of human error(s) into the software, including omissions.

- *Vulnerability*: A vulnerability is a set of conditions that allows an attacker to violate an explicit or implicit security policy. Attackers exploit vulnerabilities to break the security of an asset. Not all the security flaws lead to vulnerabilities, however, a security flaw can cause a program to be vulnerable to attacks when executed in risky environment and conditions. In other words, a vulnerability is an exploitable flaw.

- *Risk*: A risk is the probability that an attack to an asset succeeds. Vulnerabilities increase the risk of security breaches, while countermeasures reduce it.

- *Exploit*: An exploit is a piece of software or technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy. There are different forms of exploits including worms, viruses, trojans, etc.

- *Security Objective*: A security objective is a statement of intent to counter and address threats and satisfy the identified security needs. The state of security is achieved when the protection against threats is guaranteed.

- *Security Requirement*: A security requirement is a necessity for protecting an asset against exploitation and attacks.

- *Security Policy*: A Security policy is a set of rules that specifies or regulates how a system or an organization provides security services to protect sensitive and critical resources.

- *Countermeasure or Mitigation*: A countermeasure or mitigation is action(s) taken in order to protect an asset against exploits and attacks. In other words, they are methods, techniques, mechanisms, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities. Countermeasures reduce the risk of security breaches.

There exist many and various security requirements for information and computer security described in many documents, standards and books [50, 70]. There is no consensus on a standard list of requirements, which are chosen according to the application domain and the desired security level. In the sequel, we briefly overview the most important high-level security requirements and we focus on the list provided by the ISO standard 7498-2 in the context of distributed systems [8, 31]:

- *Authentication*: Corroborating of the identity of an entity or source of information.

- *Access Control*: Restricting access to resources to privileged entities.
- *Data Confidentiality*: Keeping information secret from all but those who are authorized to see it.
- *Data Integrity*: Ensuring that information has not been altered by unauthorized or unknown means.
- *Non-Repudiation*: Preventing the denial of previous commitments or actions.

Other requirements may exist in the security literature such as availability, anonymity, auditing, certification, privacy, revocation, freshness, etc.

The enforcement of security requirements to protect valuable assets can be achieved at different levels and in different forms. In this context, many security mechanisms and countermeasures exist in the literature. We distinguish from them the encryption for confidentiality, hash functions for integrity, message authentication code and digital signatures for authentication and so forth [50, 70]. However, these security mechanisms may not always be sufficient to ensure the above requirements. For instance, some low level security problems such as buffer overflows are not covered by this list. The following sections discuss such problems in detail.

Once the threats and/or the security requirements are well identified and categorized, it is possible to determine the appropriate technique(s) to mitigate and/or enforce them. The literature often portrays threats and vulnerabilities accompanied with a mapping to known counter-measures addressing them. Please refer to Table 1 for an instance of such a mapping.

Threat Type	**Mitigation Techniques**
Spoofing Identity	Appropriate Authentication, Protect Secret Data
Tampering with Data	Appropriate Authorization, Hashes, Message Authentication Codes, Digital Signatures
Repudiation	Digital Signatures, Timestamps, Audit Trails
Information Disclosure	Authorization, Encryption, Protect Secrets
Denial of Service	Appropriate Authentication and Authorization, Filtering, Throttling, Quality of Service
Elevation of Privilege	Run with Least Privilege

Table 1: Mapping Between Threats and Mitigations (Excerpt from [50])

Apart from the physical protection of assets, typical approaches of improving computer security can be applied to different components of computer systems i.e., communication/network, hardware and software. In the following, we explain briefly each one of them:

- Communication/Network level security consists of securing the network infrastructure, shared resources (e.g. printers, network-attached storage, etc.), and access to individual computers.
- Hardware level security consists of imposing restrictions and rules on computer operating system and software.
- Software level security consists of improving the security of the software itself and the operating system through remedying existing vulnerabilities and/or adding security features such as access control, authentication, encryption, etc.

In this thesis, we are only concerned with software security and the problems and solutions related to it. In this context, we provide in the following sections the methodologies and requirements addressing this field.

2.3 Software Security Hardening

The security hardening term at the software/application level is relatively unknown in the current literature. As such, we define 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.*

We also propose a taxonomy of security hardening methods that refer to area to which the solution is applied. We established our taxonomy by studying the solutions of software security problems in the literature. We also investigated the security engineering of applications at different levels, including specification and design issues [15, 18, 50]. From this information, our practical experiments, and some hardening advice existing in the literature, we were able to draw out the following classification of methodologies for software security hardening:

Code-Level Hardening

Code-Level hardening constitutes *changes in the source code in a way that prevents vulnerabilities without altering the software structure.* During software creation, vulnerabilities are created and are a direct result of the programming phase of the project. Code level hardening constitutes of removing these vulnerabilities by implementing the proper coding standards that were not enforced originally.

Software Process Hardening

Software Process hardening is the *addition of security features in the software build process without changes in the original source code.* Software process hardening considers choosing appropriate platforms, library implementations, compilers, aspects, etc. that result in increased security. It is also possible to use compilers and aspects that add some protections in the object code, which were not specified in the source code, and that prevent or complicates the exploitation of vulnerabilities existing in the program. To a certain extent, it externalizes the security concerns from the program, but has the disadvantages of being harder to audit and may lack portability.

Design-Level Hardening

Design-Level hardening is the *re-engineering of the application in order to integrate security features that were absent or insufficient.* It refers to changes in the application design. Some security vulnerabilities cannot be resolved by a simple change in the code or by a better environment, but are due to a fundamentally flawed design. Changes in the design are thus necessary to solve the vulnerability or to ensure that a given security policy is enforceable. Moreover, some security features need to be added for new versions of existing products. This category of hardening practices target more high-level security such as access control, authentication and secure communication. In this context, best practices, known as security design patterns [18], can be used to guide the redesign effort. Although such patterns are targeting the security engineering of new systems, such approach can also be redirected to cover deploying security into existing software.

Operating Environment Hardening

Operating Environment hardening consists of *improvements to the security of the execution context (network, operating systems, libraries, utilities, etc.) that is relied upon by the software*. It impacts the security of the software in a way that is unrelated to the program itself. This addresses the operating system (typically via configuration), the protection of the network layer, the configuration of the middleware, the use of security-related operating system extensions, the normal system patching, etc. [6, 87]. Many security appliances can be deployed and integrated into the operating environment in a way that provides some high-level security services. These hardening practices fall within the scope of proper management of an IT department and, as much as they can prevent exploitation of vulnerabilities, they do not remedy them.

2.4 Security Engineering Using Design Patterns

Computer security professionals have been promoting, for many years, tools and best practices guidelines to be used by the software development industry [14, 18, 50, 71]. Developers, often pressed by a dominating time-to-market priority, must deal with a large set of technical and non-technical issues, in which case security concerns are not thoroughly addressed. As such, the concept of security engineering has been proposed to explore the importance of addressing security issues early during software development and provide guidance.

The initial proposals in this domain have been suggested and known as secure software design advices or tips. In this context, Bishop, in a set of instructional slides [14], offers advice on dividing software in processes that have exactly the right level of privilege to perform their task. In [15], the reader will also learn secure software design from design principles and case studies. Howard and LeBlanc [50] suggest to use threat modeling as a tool for correctly choosing the right security mechanisms and their proper deployment into systems. They show how secure software should not be structured in an arbitrary manner, but that design decisions should be directly correlated with the requirements and policies. In [51], Howard and Lipner describe the process used by Microsoft to create secure software, named the Security Development Lifecycle (SDL). They insists on the need to create defense in depth, since code-based vulnerabilities are not fully avoidable in practice. As such, they show a methodology used within the SDL for design named attack surface reduction. Attack surface reduction is a strategy that dictates the reduction of entry points, privileges and amount of executing code. One particular concern that it deals with is the presence of anonymous paths that are of a higher risk. Graff and Wyk [42] have also written on the principles and practices behind the construction of secure software. Their book covers all the steps of the software development process, and includes a coverage of security architecture and design.

All these books are useful, but often lack direct advice on how to well design applications while taking security into consideration. For a decade now, security design patterns have emerged in order to answer this need. Security design patterns have been proposed recently as a guide for the improvement of software security during the design phase. Since the appearance of this research topic in 1997, several pattern catalogs have emerged, and the security design pattern community has produced many contributions [18,38,71]. In the sequel, we present an overview of the pattern concepts and particularly security patterns.

2.4.1 Pattern Concepts

Patterns are structured documentations that capture well-defined solutions to recurring problems. The basic idea is to write down best practices and lessons learned from a given problem domain in an organized way [70, 71]. Christopher Alexander *et al.* [11] provided the first definition of a pattern and its structure in the field of building architecture, which was later reused in the object-oriented world:

"Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem...".

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution".

The main concepts of the pattern idea are explored in the aforementioned definitions. According to Alexander [11], each pattern should have the same structure and template in order to achieve better comprehension, comparison, and usage of the patterns. The core elements of such template are the name, context, problem, trade-offs, and solution. Other elements could be added to give more detail and explanation on the pattern if needed. The following is a brief description of these elements:

- *Name*: The name of the pattern is the common-usage short expression that encapsulates the pattern meaning. The name is determined and assigned according to the community to which the pattern belongs. It should be expressive, reflect clearly the content of the pattern, and easy to refer to.

- *Context*: The context describes the environment where the pattern could be applied. In other words, it explores when and where the pattern will work and the preconditions under which the problem and its solution appear.
- *Problem*: The problem is a short description of the problem that this pattern aims at solving. It describes the goals and objectives a pattern needs to achieve.
- *Trade-Offs*: The trade-offs section aims at explaining in more details the nature of advantages and disadvantages of this pattern over quality attributes.
- *Solution*: The solution is a textual description of the pattern that solves the problem. It is presented as a proven solution of the problem proposed by domain experts. Proven solution means that this solution worked at least once in a well defined environment. The level of abstraction of the solution is directly related to the type of the problem. For instance, we can find some patterns that provide design level solutions, while others provides code level solutions. Typically, a problem can have more than one solution, and the best one is only determined by the context where the problem occurs.
- *Related Patterns*: Patterns related to this pattern, or patterns that inspired this pattern, are listed in this section. Patterns do not exist in isolation, and the presentation of this relationship provides linkage to subsequent patterns of pattern collection.

Pattern Organization

Patterns do not exit in isolation, as such, there are different concepts for collecting and organizing them depending on the relationships among them. Pattern catalogs, pattern systems, and pattern languages are the main approaches [24,70] that are adopted for pattern organization. The following is a brief description of these approaches:

- *Pattern Catalogs*: A catalog of patterns consists of patterns belonging to the same community (e.g., security), without the necessity to have relationships among them. It is the result of merging several individual patterns into a bigger collection. A pattern catalog is more a loosely coupled set of patterns [70]. However, typically and not mandatory, the same template and structure is used for all the patterns of the catalog.

- *Pattern Systems*: A pattern system for software architecture, as defined in [70], is a collection of patterns for software architecture, together with guidelines for their implementation, combination, and practical use in software development. It is a more tightly coupled set of patterns than a pattern catalog and it precisely describes and explores the relationships and interactions between individual patterns. These patterns work together to solve a more complex problem in a particular domain.

- *Pattern Languages*: A pattern language consists of patterns that have a common pre-defined goal and each one of them contributes to provide the solution to one overall problem. These patterns are composed together to form a big pattern that provides the solution for the problem. Each pattern, by itself, is not considered as a solution for a particular problem. We can notice that the meaning of a language in the domain of patterns is different than its conventional one.

Pattern Categories

In [24], Buschman et al. divided the patterns into the following three categories:

- *Architectural Patterns*: Architectural patterns address the problems faced during the architecture level of the software development process. They provide solutions about the structural organization of software systems and the relationships among their components.
- *Design Patterns*: Design patterns address the problems faced during the design of software systems. Most of the patterns available in the literature belong to this category. This type of patterns are still independent of the implementation.
- *Idioms or Implementation Patterns*: Implementation patterns address the problems faced during the implementation of the software systems. They are mostly programming language dependent. Very few implementation patterns exist in the literature.

Pattern Mining and Quality Assurance

The major claim of patterns is that they are proven solutions for recurring problems. However, this is difficult to be true and guaranteed, particulary because there is no well-defined procedure for assessment. The only way that is currently used is to publish the patterns and review them by a community of experts. On the other hand, there are several basic approaches for pattern mining, which may also be used to prove the validation of the proposed solutions. The following are the aforementioned approaches presented in [53]:

- *Introspective*: In this approach, the developers build their own systems, analyze them and identify the solutions that work well. Then, these solutions are written and presented as patterns. These patterns are limited to individual experience, as such, the authors need to make sure that the other experts agree on these solutions.

- *Artifactual*: In this approach, the authors of the patterns are not among the people who design and develop the systems. In this case, the authors investigate the solutions and write the corresponding patterns. There is a possibility that the authors are not experts in the pattern domain, and as such, the patterns may need additional refinements.

- *Sociological*: In this approach, the resulted patterns are the most solid and guaranteed. Several experts contribute to building the patterns by developing several systems that solve a particular problem, discussing the proposed solutions and then determining the best one. The chosen solution will constitute the pattern.

2.4.2 Security Patterns

Security patterns are patterns that belong to the security community. They approach the problem from the same perspective, by encapsulating expert knowledge in the form of proven/well-defined solutions to common security problems. These patterns will fit at different levels of abstraction and areas of concerns. Schumacher provided in [70, 71] the following definition for 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".

Security patterns have a structure similar to the one of design patterns, i.e., they have an expressive name, context, problem, solution, and their relation to other patterns. These components form together the template for security patterns. Like other patterns, there are also some optional elements, which can be used to improve the comprehension of a security pattern. The aim of the template elements is to explain the use of a pattern. For instance, if a developer wants to use a pattern, he first checks the context and problem, then, if they fit with his problem and context, he applied the provided solution. Moreover, all the issues of pattern organization, pattern categories and pattern mining and validation are applied to security patterns, with the only difference that they are applied to the IT security domain and community.

2.4.3 Literature Review

The current research in the domain of security patterns is characterized by various publications. However, the field is lacking a core reference similar to the "Gang of Four" patterns [40] in typical software design and has no established criteria for evaluation. This situation makes the security design patterns hard to use for software designers and maintainers alike, which limits their adoption in the industry, and thus lowers their positive impact on software security.

In [90], 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 a starting point towards a collection of patterns that can help developers to address security issues when developing new applications.

Kienzle et al. [56] have created a 29-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 derived from the implementation of specific web application security policies.

Romanosky [68] introduced another set of design patterns. The discussion however has focused on architectural and procedural guidelines more than security patterns.

Brown and Fernandez [38] 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 [39].

Braga et al. [22] also investigated security-related patterns specialized for cryptographic operations. They showed how cryptographic transformation over messages could be structured as a composition of instances of the cryptographic meta-pattern.

The Open Group [18] has introduced an important list of security design patterns. Their catalog proposes 13 patterns, and is based on architectural framework standards such as the ISO/IEC 10181 [7].

The most recent work in this domain is from Schumacher et al. [71]. They offered into a book a list of forty-six patterns applied to different fields of IT security, however most of them are rewriting of previously proposed patterns.

2.4.4 Evaluation

The design principles and patterns for secure systems development are quite sound but are not meant to deal with already developed software, although they could be useful guides for a system redesign. However, the basic principle of the pattern is promising and should be leveraged in order to clearly specify how security hardening is to be done.

Security patterns are currently written in a way that requires manual adaptation and makes their usefulness limited. This seems more like a limitation of the expression format, and not of the concept itself. This conclusion brought us to adopt the idea of patterns as a way to specify the security hardening solutions, while also considering a new and extended form of pattern expression that we describe in Chapter 4.

2.5 Secure Programming

The field of secure programming is focused on avoiding common programming mistakes that result in security vulnerabilities. As such, this field is highly tied to the technology used, typically being the programming language and the operating system.

A lot of research has been published in the field of secure programming for C and C++ programs, since a large amount of production-level software has been written in those languages [17, 50, 73, 87]. The design of those languages also allows the existence of many types of security vulnerabilities that are not possible in modern languages. They require manual memory management and offer low type safety, both being sources of many programming errors. Furthermore, since these languages are capable of interacting with the operating system directly, they are able to use system resources in a manner that is not always safe. It is thus very important to understand the security implications of improper C and C++ programming and learn good practices to avoid them.

We will now proceed to a non-exhaustive survey of existing vulnerabilities and their corresponding available solutions. After that, we will examine the current contributions in the field.

2.5.1 Security/Safety Vulnerabilities

Regarding low level security, deploying security at that level is mainly categorized as Code-Level and Software Process hardening. As such, this type of hardening will be extremely dependent on the programming language and the platform. In this context, the C and C++ programming languages have a bad reputation in the security world because they were designed for maximal performance, at the expense of some safety-enhancing techniques. The C and C++ memory management left to the programmer discretion and the lack of type safety are the major causes of security flaws. Such flaws do not exist in the Java programs because the compiler takes care of most of the issues that cause these security problems and handles all the memory management operations. In the sequel, we list the major safety vulnerabilities that are introduced in the source code during the implementation and we discuss their impacts on software security. Moreover, we provide a brief description of the mitigation techniques used to remedy them.

Buffer Overflow

Buffer overflow exploit common programming errors that arise mostly from weak or non-existent bounds checking of input being stored in memory buffers. Attacks that exploit these vulnerabilities are considered as one of the most dangerous security threats since it can compromise the integrity, confidentiality, and availability of the target system. Buffers on both the stack and the heap can be corrupted [50, 91]. The following are the common causes of buffer overflow: Boundary condition errors, input validation errors, assumption of null-termination, and improper format string.

Many APIs and tools have been deployed to solve the problem of buffer overflow or to make its exploitation harder [13, 50, 60, 91]. In this context, the following are some design and programming tips and assumptions that can help to solve the buffer overflow problem [17]:

- Always assume that input may overflow a buffer and design the program in a way that provide proper input validation conditions.
- Use functions that respect buffer bounds such as fgets, strncpy, and strncat.
- Ensure NULL-termination of strings, even if using those functions.
- Invalidate stack execution, since stack-based buffer overflow are the easiest to exploit.
- Check the number of arguments of printing functions to make sure that the format string argument is explicitly specified.

Table 2 summarizes the hardening solutions for the buffer overflow security problems with respect to the aforementioned security hardening classification.

Hardening Level	**Product/Method**
Code	Bound-checking, memory manipulation functions with length parameter, null-termination, ensuring proper loop bounds
Software Process	Compile with canary words, inject bound-checking aspects
Design	Input validation, input sanitization
Operating Environment	Disable stack execution, use libsafe, enable stack randomization

Table 2: Hardening for Buffer Overflows

Integer Operations

Integer security issues arise either on the conversion (either implicit or explicit) of integers from one type to another, or because of their inherently limited range [73]. C and C++ compilers distinguish between signed and unsigned integer types and silently perform operations such as implicit casting, integer promotion, integer truncation, overflows and underflows. Such silent operations are typically overlooked, which can cause various security vulnerabilities. Integer vulnerabilities may be used to write to an unauthorized area of memory. A first instance, is the allocation of less memory than thought, allowing to write to unwanted parts of the heap. Another instance is to access invalid memory areas with a negative index or memory copying operation. In some cases, if the access is to an invalid page, the result will be a denial of service via an application crash. They can also cause other security problems by bypassing preconditions and expected protocol values that are specific to the program being exploited. The following are the common causes of these vulnerabilities:

- *Integer Sign Conversion*: Sign conversion errors occurs because the C and C++ languages conserve the bit pattern when converting between signed and unsigned integers of the same size, so we can find negative or unexpectedly large values when not expected. This is due to the fact that most processors store signed integers with the first bit as the sign bit, whereas the same bit is used by unsigned integers in the same way as other bits.

- *Integer Signedness Errors*: Signedness errors occur when the program expects an unsigned value, but instead finds a signed one. Because of the inappropriate assumption, the program does not validate if the value is positive, potentially resulting in a security vulnerability. These errors typically happen in conjunction with conversion errors.

- *Integer Truncation Errors*: A truncation error occurs when an integer is converted to one of a smaller type. The bit pattern of a subset of the original integer is preserved as is. If the smaller type is signed, this may result in a negative value.

- *Overflow and Underflow*: Integer overflow and underflow happen when adding or multiplying beyond the integer maximum value or dividing by -1 (overflow) or subtracting below its minimal value (underflow). This will result in errors similar to integer conversion. It is also noteworthy to remember that unsigned integers obey modular arithmetic rules in case of overflow, resulting in smaller values than expected, but that are still positive.

Those vulnerabilities can be solved using sound coding practices. The generalized use of unsigned integers can simplify things for the programmer, and the addition of range checking before sensitive operations can avoid unexpected results. Some compilers provide built-in supports for the detection of integer issues, and it is possible to replace integer operations with safer calls [73]. Table 3 summarizes the hardening solutions for the integer security problems with respect to the aforementioned security hardening classification.

Hardening Level	**Product/Method**
Code	Use of functions detecting integer overflow/underflow, migration to unsigned integers, ensuring integer data size in assignments/casts
Software Process	Compiler option to convert arithmetic operation to error condition-detecting functions
Design	-
Operating Environment	-

Table 3: Hardening for Integer Vulnerabilities

Memory Management

The C and C++ programming languages allow programmers to dynamically allocate memory for objects during program execution. C and C++ memory management is an enormous source of safety and security problems. The programmer is in charge of pointer management, buffer dimensions, allocation and deallocation of dynamic memory space. Thus, memory management functions must be used with precaution in order to prevent memory corruption, unauthorized access to memory space, buffer overflow, etc. The following are the major errors caused by improper memory management in C and C++

- *Using Uninitialized Memory*: In C and C++ programming, the memory space pointed to by newly declared pointer is not initialized, and it typically points to a random location. The consequence of de-referencing these pointers include denial of service, information disclosure, and memory corruption.

- *Accessing Freed Memory*: A pointer on which the `free` function was called can still be accessed. It is however possible that the memory range was allocated for another use, and that this access will result of reading invalid data or corrupting data used by another part of the process.

- *Freeing Unallocated Memory*: The `free` function must be called on a memory location previously allocated using the `alloc` family of functions. Otherwise, freeing an unallocated memory location can cause memory corruption and denial of service. This problem arises because the free call is performed with the uninitialized pointer, either through programming error or a failed memory allocation.

- *Memory Leaks*: The dynamically allocated memory must be freed after usage and before the pointer to its location goes out of scope. Failure to do so results in a memory leak. Memory leaks degrade performance and can cause a program to run out of memory and crash.

There are no known API or library solutions solving such problems as a whole. However, hardened memory managers can prevent multiple freeing vulnerabilities. Other than that, only improvements in programming practices can be useful in hardening against such problems. The following are some hints and best practices: Initialize each declared pointer and make it point to a valid memory location, do not allow a process to de-reference or operate on a freed pointer, and apply error checking on memory allocation calls. Table 4 summarizes the hardening solutions for the memory management security problems with respect to the aforementioned security hardening classification.

Hardening Level	Product/Method
Code	NULL assignment on freeing and initialization, error handling on allocation
Software Process	Using aspects to inject error handling and assignments, compiler option to force detection of multiple-free errors
Design	-
Operating Environment	Use a hardened memory manager (e.g. `dmalloc, phkmalloc`)

Table 4: Hardening for Memory Management Vulnerabilities

File Management

The `C` and `C++` programming language provides functions for creation, deletion and manipulation of files and directories. File management problems occur when an access to or a modification of a restricted file happens. Some problems are closely related to race conditions. File management errors can lead to many security vulnerabilities such as data disclosure, data corruption, code injection and denial of service. The following are two major sources of vulnerabilities in file management:

- *Unsafe Temporary File*: Many programs use temporary files, with default access restrictions. If the access permissions are incorrectly set, it is possible for the temporary file to be used as an attack vector for the application or another system file [87]. Another contributing factor to this problem is the fact that the name of the temporary file could be predictable.

- *Improper File Creation Access Control Flags*: A file is typically opened using the process default file creation bit mask. This mask is typically inherited from the launching process. An attacker could alter this mask to make the information accessible, whereas it should be protected.

Vulnerabilities related to unsafe temporary file creation can be minimized by using secure library calls [87]. In some cases, we can redesign the application to use inter-process communication instead of temporary files. File creation mask vulnerabilities, in UNIX-like systems, can be resolved using proper file creation-related system calls and specifying appropriate access rights. Table 5 summarizes the hardening solutions for the file management security problems with respect to the aforementioned security hardening classification.

Hardening Level	**Product/Method**
Code	Use proper temporary file functions, default use of restrictive file permissions, setting a restrictive file creation mask
Software Process	Set a wrapper changing file creation mask
Design	Refactor to avoid temporary files
Operating Environment	Restricting access rights to relevant directories

Table 5: Hardening for File Management Vulnerabilities

2.5.2 Literature Review

Currently, security solutions can be found in secure coding books [15, 50], in programmer/reviewer checklists, and in the mind of many experts. The focus of this help is to allow the creation of new programs that are designed, implemented and maintained for security, but does not offer practical support on how to deal with legacy code and how to harden security systematically into existing software. On the topic of secure programming of C and C++ programs, developers are offered a good selection of useful and highly relevant books and other materials.

One of the newest and most useful additions is from [73], which offers in-depth explanations on the nature of all known low-level security vulnerabilities in C and C++ . Their treatment of integer overflows is the best we found in the literature.

Another common reference is from Microsoft [50], and includes all the basic security problems and solutions, as well as code fragments of functions allowing to safely implement certain operations (e.g., safe memory wiping). The authors also describe high-level security issues, threat modeling, access control, etc.

Slides from Bishop, in addition to his landmark book [15, 16], provide a comprehensive view on information assurance, as well as security vulnerabilities in C notably on the topic of environmental issues. In addition, he provides some hints and practices to solve some existing security issues.

Wheeler [87] offers the widest-reaching book on system security available online. It covers operating system security, safe temporary files, cryptography, multiple operating platforms, spam, etc. We consider his solutions as the most relevant to the problem of insecure temporary files.

The Secure Programming Cookbook for C and C++ [83] is a hands-on solution for programmers looking for direct solutions to typically-encountered security problems. The authors mention recipes for safe initialization, access control, input validation, cryptography, networking, etc.

2.5.3 Evaluation

The field of secure programming offers many highly relevant, although sometimes repetitive, contributions that can drastically help programmers to write secure code. They are also of help for developers who need to improve manually the security of existing systems.

As such, the secure programming references could be used as a primary resource for the construction of security hardening solutions dealing with related low-level security issues.

On the other hand, this approach aims at educating programmers about the causes of vulnerabilities in order to help remedying them. Its limitation is that the hardening solutions are applied manually by developers and depends considerably on human decisions. This means that the developers responsible for performing the hardening should have high security expertise. Beside, due to the manual application of hardening, such approach cannot prevent human errors and cannot be applied on large scale software (e.g., thousands or even million(s) lines of code).

2.6 Conclusion

We presented in this chapter the major approaches in the literature that are relevant somehow for integrating security into software, and hence constitute together a guide to build on top of them and elaborate the intended systematic security hardening framework. First, we went trough the security engineering using design patterns approach, explained its concepts and components and explored that the principle of pattern is promising and should be leveraged in order to clearly specify how security hardening is to be done. However, we concluded that this approach misses the methodologies required for applying the security solutions, which limits their usefulness by non experts in security. Beside, it addresses security during the design of new software. Second, we discussed the secure programming techniques and provided an overview of the security vulnerabilities that they deal with. This field offers many highly relevant contributions that can drastically help programmers writing secure code. As such, the secure programming references should be used as a primary resource for the construction of security hardening solutions dealing with related lower-level security issues. However, we found also that secure programming practices are applied manually by programmers and are too reliant on their sagacity and decisions, which enquires high security expertise and limits their useability for systematic security improvements. Regarding the use of AOP for integrating security into software, this approach offers strong potential for systematic security code injection, and hence it will be discussed rigourously in the next chapter

Chapter 3

Towards Security Hardening Via Aspect-Oriented Programming

3.1 Introduction

Software security hardening requires radical transformation of the original source code such as changing what is available, augmenting it, and/or even removing it. In this context, few initiatives have been introduced recently for code injection, via an aspect-oriented computational style, into source code for the purpose of improving a security requirement or remedying a vulnerability [20,31,52,74,78]. These approaches are based on the idea of separating out the security concerns from the rest of the application, such that they can be addressed independently and applied globally. However, the current corpus of research in AOP-based security is still recent and falls short of a well-defined and organized systematic solution usable for applying security hardening by developers non-expert in security. A developer still needs to build the security solution and specify it into aspect(s).

On the other hand, AOP offers the facilities that allow to select and match some join points in the code (e.g., function call, function declaration, class declaration, etc.), insert code before and/or after the matched join points and replace the matched join points with new code. This process of code matching and injection is called weaving. In this context, AOP seems to be very a promising paradigm that provides features required to elaborate our methodology, and hence we can build on top of it to achieve our intended framework for systematic security hardening of software. A detailed discussion about AOP and its usability and applicability for security hardening is provided in the this chapter.

The rest of the chapter is organized as follows. We first present in Section 3.2 the existing AOP models, components, and languages. Then, we explore through practical examples in Section 3.3 its relevance for software security hardening. Afterwards, we provide in Section 3.4 the literature review related to this research domain. Finally, in Section 3.5, we provide concluding remarks about what is covered in this chapter.

3.2 Aspect-Oriented Programming

AOP is a relatively new programming paradigm that provides a more advanced modularization mechanism generally on top of the traditional object-oriented programming. It is based on the idea that computer systems are better programmed by separately specifying the various concerns, and then relying on underlying infrastructure to compose them together into a big program. The techniques in this paradigm were precisely introduced to address the development problems that are inherent to crosscutting concerns. 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. Then, these aspects are composed and merged with the core functionality modules into one single application. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

3.2.1 AOP Models

There are many AOP models, the most important ones are the following: Pointcut-Advice, Multi-Dimensional Separation of Concerns and Adaptive Programming. Just to note that some references consider only the Pointcut-Advice model as AOP, while the other two are concepts similar to AOP [75].

Pointcut-Advice Model

The approach adopted by most of the AOP languages 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 application. 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 a pointcut expression, whose matching criteria restricts the set of a program join points for which the advice will be injected. A join point is an identifiable execution point in the application 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. 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. AspectJ [55] and AspectC++ [79] are instances of the languages that are based on the pointcut-advice model.

Multi-Dimensional Separation of Concerns

The Multi-Dimensional Separation of Concerns (MDSOC) [64] approach provides developers with simultaneous separation of concerns in software according to multiple and arbitrary dimensions of composition and decomposition. It treats all the concerns equally, and this includes the program components and aspects. On the other hand, most AOP approaches do not support composition between program components or between aspects and they only enable the aspects to be composed with components. In other words, the MDSOC is a symmetric approach, as opposite to the pointcut-advice one where aspects are woven in the original application. Hyper/J [80] is an instance of the languages that are based on Hyperspaces, which is a particular approach of MDSOC. In Hyperspaces, the software is modeled as a set of hyperslices, where each hyperslice is a set modules representing a single concern.

Adaptive Programming

The idea of AOP has been used several years ago before the proposition of AOP with this exact nomination. They have been proposed by Demeter group [43] in the Adaptive Programming approach. The programming style rule for loose coupling between the structure and behavior concerns is part of the *demeter* law. Following this rule in software development results in a large number of small methods scattered throughout the program. To avoid such problem, adaptive programming with traversal strategies has been proposed to better support the loose coupling of concerns. The *demeter* methodology is defined in three steps [63]: Derive a class graph that captures the structure of the application, derive traversal methods by finding a traversal path for each program operation and derive visitor methods by attaching specific behavior to certain classes that are visited.

3.2.2 AOP Languages

There are many AOP languages that have been proposed. These languages are used for code implementation and are programming language dependent. We distinguish from them AspectJ [55] built on top of the Java programming language, AspectC [27] built on top of the C programming language, AspectC++ [79] built on top of the C++ programming language, AspectC# [57] built on top of the C# programming language, and an AOP version for Smalltalk [19]. AspectJ and AspectC++ are dominant propositions in the field of AOP.

Other related and special purpose languages have been also proposed. Tribe [26] offers an approach based on virtual class families. The AWED language [62] was developed for distributed applications and it also supports sequences. TOSCANA [37] is a toolkit for kernel-level AOP programming. It allows to modify the memory kernel to perform autonomic (i.e., self-managing) computing. Their language is quite simple, but is restricted to C . The Arachne system [35], part of the OBASCO project is providing an interesting aspect-oriented language for which the sequence of events is encoded in the aspect. Their approach is C-centric and works on the in-memory binary process. Since AspectJ and AspectC++ are quite similar, both are based on the pointcut-advice model and most of our experiment aspects are coded in AspectC++ language, we will discuss only AspectC++. The reader can also refer to [55] if needed to understand the examples implemented in AspectJ.

3.2.3 AspectC++ Programming

AspectC++ defines an aspect-oriented extension to the C++ programming language based on the pointcut-advice model. It also provides tool support for the modularization of crosscutting concerns. AspectC++ is similar to AspectJ, but, due to the natures of C++ and Java, in some regards it is fundamentally different. For instance, the weaving in AspectC++ is performed at the source code level, while in AspectJ it is applied at the bytecode level. In the sequel, we only explain the concepts and elements of the AspectC++ language that are important and needed for this thesis work. Detailed information on AspectC++ is available in [3, 79].

AspectC++ supports the separate programming of crosscutting concerns. One can define additional behavior to be integrated and woven into the code of the original application using a special purpose tool, the AspectC++ weaver. The weaver must be provided with all the relevant application and aspect files. The result of the weaving process is a new set of source code files describing the weaved application. Figure 1 illustrates first the compilation process alone, then the compilation and weaving processes together. Crosscutting behaviors in AspectC++ are specified using aspects, which enclose all the other elements

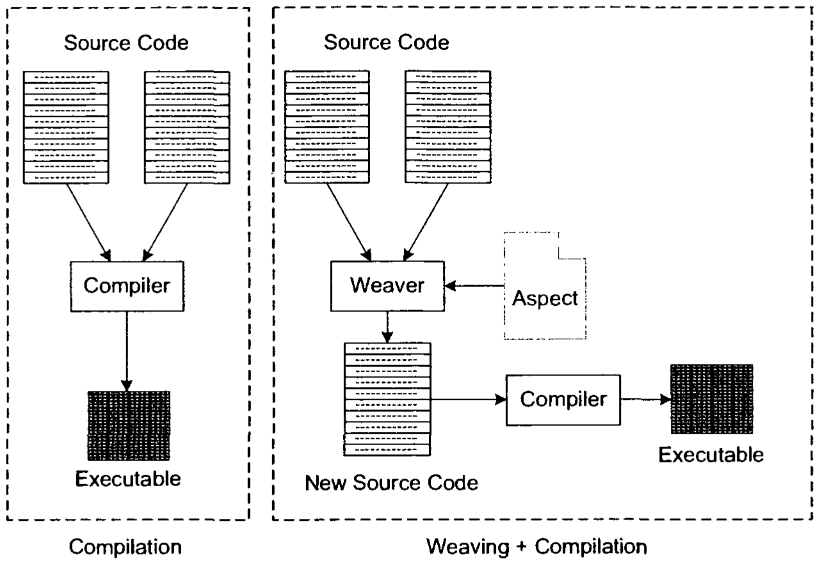


Figure 1: AspectC++ Weaving

of the AspectC++ language. Aspects consist of advices that describe the behavior and pointcuts that specify the join points in the application code where this behavior must be weaved.

Join Point and Pointcut

As aforementioned in the pointcut-advice model, a join point is a location in the source code of the application where the code of the additional behavior should be inserted. This location could be static (e.g., function call) as well as dynamic (e.g., execution flows). In other words, a join point is the selected place for the composition of different concerns. Each join point can either refer to a function, an attribute, a type, a variable, or a point from which a join point is accessed, so that this condition can be for instance the event of reaching a designated code position. Matching a join point is mainly based on properties such as function name, object type, etc.

A pointcut constitutes the constructor that designates a set of join points to determine on which condition the aspect shall take effect. Depending on the kind of pointcuts, they are evaluated at compile time or at runtime. AspectC++ provides a set of predefined pointcut designators and functions that allows specifying the joint points to be matched in the code. They can have arguments to select particular join points out of the set of all available join points. Furthermore, pointcuts can be combined using logical operators. In the sequel, we present the most important pointcut designators and their corresponding functionalities as specified and presented in the AspectC++ language documentations [3]:

- *call(pointcut)*: Returns all the join points where the entity (i.e., class or function) specified in the pointcut is called. If the pointcut refers to a class, then all the calls to its methods will be provided by the designator.
- *execution(pointcut)*: Returns all the join points that refer to the implementation of the entity (i.e., class or function) specified in the pointcut. If the pointcut refers to a class, then all the implementation of its methods will be provided by the designator.
- *base(pointcut)*: Returns all the base classes of the ones specified in the pointcut.
- *derived(pointcut)*: Returns all the classes specified in the pointcut and all the ones derived from them.
- *cflow(pointcut)*: Returns all the join points that take place in the dynamic execution context of the entity (i.e., class or function) specified in the pointcut. The current AspectC++ language still has some restrictions with respect to the features that are used in the argument of the *cflow* pointcut. The cflow designator does not support an

argument list containing context variable bindings or other pointcut designator that needs to be evaluated at runtime like *cflow* itself.

- *within(pointcut)*: Returns all the join points that are within the entity (i.e., class or function) specified in the pointcut.
- *construction(pointcut)*: Returns all the join points where an instance of the class specified in the pointcut is constructed. This designator works even if there is no constructor defined explicitly.
- *destruction(pointcut)*: Returns all the join points where an instance of the class specified in the pointcut is destructed. This designator works although a destructor does not need to be defined explicitly.
- *that(type pattern)*: Returns all the join points where the current *this* pointer refers to an instance of a type matching the one described in the pattern.
- *target(type pattern)*: Returns all the join points where the target object of a call is an instance of a type matching the one described in the pattern.
- *result(type pattern)*: Returns all the join points where the result object of a call/execution is an instance of a type matching the one described in the pattern.
- *args(type pattern, ...)*: Returns all the join points where their argument type(s) is(are) matching the one(s) described in the pattern(s).
- *pointcut && pointcut*: Returns all the join points resulting from the intersection of the join points matched in both pointcuts.

- *pointcut* || *pointcut*: Returns all the join points resulting from the union of the join points matched in both pointcuts.

- *! pointcut*: Returns all the join points resulting from exclusion of the join points matched in the pointcut.

Advice and Join Point APIs

In AspectC++, an advice is used to describe the behavior that needs to be merged at the matched join points specified in the pointcut. An advice is somehow similar to a function of a class that contains the code statements describing a behavior. However, it also specifies how these statements must be woven with respect to the join point (e.g., before the join point). Advices are divided into two categories: advices for the join points matched in the dynamic control flow of the running program (e.g., function call or execution), and advice for static join points (e.g., introductions into classes). If the aspect has header files included, in this case their code containing the advice definition is compiled prior to the affected join point location in both advice categories. There are three advice constructs depending on the place where the code needs to be added with respect to the matched join points:

- *before(...)*: Integrates the advice code before the matched join points specified in the pointcut.

- *after(...)*: Integrates the advice code after the matched join points specified in the pointcut.

- *around(...)*: Integrates the advice code in place of the matched join points specified in the pointcut.

Around advice can be considered as the combination of a *before* and an *after* advice, with the option to not invoke the original behavior. In order for the weaver to know where the behavior must be woven, an advice must always be accompanied by a pointcut. A pointcut in this context might be a reference to a stand-alone pointcut, or it could be a nameless pointcut, which only contains a pointcut body. Within the boundaries of an *around* advice, AspectC++ provides a functionality (i.e., `tjp->proceed()`) that is used to invoke the original behavior.

AspectC++ provides also the *JointPoint* API to be used within the advice code body. The functions of this API are called through the built-in object *tjp* of the class *JoinPoint*. The *JoinPoint* API functions allow to manipulate the information related to the matched join points inside the code. For instance, the user can call a function that gets the pointer to the memory position holding the first argument in order to use it in the code of the advice body. In the sequel, we present the *JoinPoint* API functions and their corresponding functionalities as specified and presented in the AspectC++ language documentations [3]:

- *static AC::Type type()*: Returns the encoded C++ type of a matched join point.
- *static int args()*: Returns the number of arguments of a function join point matched by a call or execution designator.
- *static AC::Type argtype(int number)*: Returns the encoded C++ type of the argument of a function join point matched by a call or execution pointcut designator.
- *static const char *signature()*: Returns the textual description of a matched join point (e.g., function name, class name).

- *static unsigned int id()*: Returns the numeric identifier of a matched join point.
- *static AC::Type resulttype()*: Returns the encoded C++ type of the result of a function join point matched by the call or execution pointcut designator.
- *static AC::JPType jptype()*: Returns a unique identifier describing the type of the join point matched by the call or execution pointcut designator.
- *void *arg(int number)*: Returns a pointer to the memory position holding the argument value with index number of a function join point matched by the call or execution pointcut designator.
- *Result *result()*: Returns a pointer to the memory location designated for the result value of a function join point matched by the call or execution pointcut designator or returns 0 if the function has no result value.
- *That *that()*: Returns a pointer to the object initiating a function call or returns 0 if the called function is static or global.
- *Target *target()*: Returns a pointer to the object that is the target of a function call or returns 0 if the called function is static or global.
- *void proceed()*: Integrates the original code of a matched join point code and is only used in an around advice.
- *AC::Action &action()*: Returns the runtime action object that encloses the execution environment to execute the original code encapsulated by an around advice.

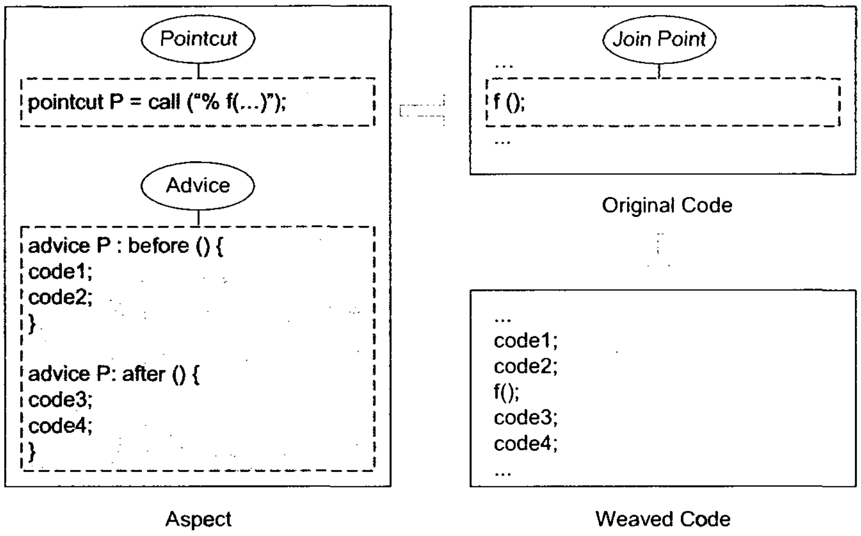


Figure 2: Aspect Structure / Matching and Weaving

Aspect

An aspect in AspectC++ is composed of zero, one, or more pointcuts and advices. In addition to this, we can use within an aspect, more specifically within the advice body, all the types and functionalities provided by the `C` and `C++` libraries by including the required header files as in a standard `C` and `C++` program. We can also add our own libraries by implementing them inside the aspect or including them as header files. Figure 2 describes the structure of an aspect and illustrates the matching and weaving mechanisms. The presented example shows a pointcut `P` matching the calls to a function `f()` and two advices to insert code before and after `P` respectively.

3.3 Appropriateness of AOP for Injecting Security Concerns

Adding security functionalities and remedying vulnerabilities into software requires a technology that enables selecting particular point(s) into a program, matching them into the original source code and allowing radical transformation over classes, functions, variables, statements, etc. Few contributions explored that AOP provides such kind of features and allows integrating security concerns into software [20, 31, 52, 74, 88]. They showed how AOP can be used to specify separately several security functionalities and then weave them with the needed components of the original programs (as illustrated in Figure 3). In the sequel, we explore through examples the appropriateness and applicability of AOP for security hardening by presenting and describing AspectJ and AspectC++ solutions for few security and safety issues.

3.3.1 Adding Identification and Authentication Using JAAS

We present in the following an example for adding identification and authentication to a bank client application. This solution has been presented in detail in [78] and its referenced citing. The corresponding bank client application is presented in Listing 3.1.

The JAAS library [5] is used for authentication. The security manager should be initialized in order to authenticate the client application before its creation (i.e., configure a login module, create an instance of `LoginContext` and `login`). Listing 3.2 illustrates

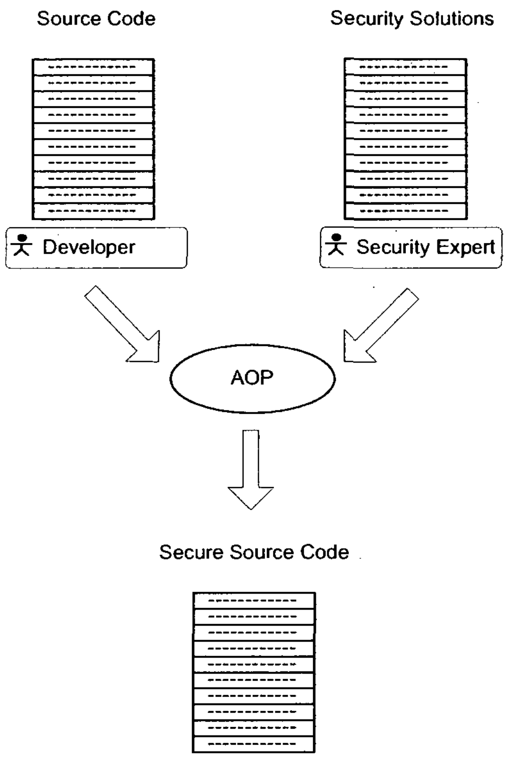


Figure 3: Separation of Security Concerns

```
class BankClient {

public static void main(String[ ] args) {
  // ...
  BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank" );
  Bank bank = homeBank.create();
  System.out.println( bank.getAccountInfo( "bill" ) );
  // ...
}

}
```

Listing 3.1: Bank Client

an AspectJ solutions describing these steps. The pointcut `pointcut mainExecution(): execution( public static void main( .. ) );` matches inside the function `main`. The first `before` advice adds the code needed to authenticate the client of bank component at the beginning of `main` (i.e., before creating the application), while the second `after` advice add the code needed to log out at the end of `main` (i.e., after executing the application).

```
aspect BankAspect {

LoginContext lc = null;

pointcut mainExecution(): execution( public static void main( .. ) );

// Login before execution of main()
before(): mainExecution() {
  AppCallbackHandler handler = new AppCallbackHandler( "scott", "echoman
     " );
  try {
    lc = new LoginContext( "Bank", handler );
    lc.login();
  } catch( LoginException e ) {
    // ...
  }
}

// Logout after execution of main()
after() returning: mainExecution() {
  try {
  lc.logout();
  } catch( LoginException e ) {
  // ...
  }
}

}
```

Listing 3.2: Aspect for Identification and Authentication

Weaving the application in Listing 3.1 with the aspect in Listing 3.2 using AspectJ compiler produces a class file with same functionalities as the application presented in Listing 3.3. The code of authentication and identification mechanisms has been added before and after the creation and execution of the bank application.

```
class BankClient {

LoginContext lc = null;
public static void main(String[] args)
{
  // Callback to get username and password. Required by LoginContext
  AppCallbackHandler handler = new AppCallbackHandler( "scott", "echoman
     " );
  try {
    lc = new LoginContext( "Bank", handler );
    lc.login();
  } catch( LoginException e ) {
      // ...
  }

  // ...
  //Start Original Code
  BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank" );
  Bank bank = homeBank.create();
  System.out.println( bank.getAccountInfo( "bill" ) );
  //End Original Code
  // ...

  try {
    lc.logout();
  } catch( LoginException e ) {
      // ...
  }
}

}
```

Listing 3.3: Weaved Bank Client

3.3.2 Detecting SQL Injection

SQL injection attack consists of embedding malicious SQL commands into the parameters of a query sent by a web application to a database [48]. This malicious query results in an attack that can corrupt, destroy and/or disclose the database contents. The most popular techniques used for SQL injection are tautology, union, additional declaration and comments. In the following, we present an AspectJ solution for detecting SQL injection.

This solution has been described in detail in [48] and used with the Tomcat application server and the MySQL database manager. The corresponding aspect, which needs to be weaved with the server application (i.e., Tomcat), is illustrated in Listing 3.4.

```
aspect SQLInjectionAspect {

pointcut dbWrite(String query):
  (call(* java.sql.Statement.addBatch(String))
  || call(* java.sql.Statement.execute(String))
  || call(* java.sql.Statement.executeQuery(String))
  || call(* java.sql.Statement.executeUpdate(String)))
  && args(query);

pointcut getParameter():
  call(String javax.servlet.http.HttpServletRequest.getParameter(String)
     );

Object around(String query): dbWrite(query){
  Object ret = validator.Validator().validateQuery(proceed());
  return ret;
}

String around(): getParameter(){
  return new validator.Validator().validate(proceed());
}

}
```

Listing 3.4: Aspect for Detecting SQL Injection Aspect

The pointcut `pointcut getParameter()` allows intercepting all the calls to the HTTP requests parameters. These requests are preceded by a validation described in the advice corresponding to this pointcut. Another validation is also applied on the database queries. The pointcut `pointcut dbWrite(String query)` allows intercepting the data base queries that have a query as parameter. The validation mechanism is described in the advice corresponding to this pointcut. The validation consists of verifying that the parameter or query is not malicious. More detail information and scenarios about SQL injection and detection can be found in [48].

3.3.3 Securing Connection Using GnuTLS/SSL

Securing channels between two communicating parties constitutes an approach to avoid eavesdropping, tampering with the transmission, or session hijacking. In the following, we present a solution that we have elaborated as part of a case study for securing the connection of client applications using GnuTLS/SSL library [4]. More detailed information about the complete solutions can be found in Chapter 4. The client application, which is presented in Listing 3.5, is implemented in `C` and `C++` and allows to connect and exchange HTTP request and data with a web server. 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, we implemented this program multiple times, with different internal structures.

Listing 3.6 illustrates excerpt of an aspect developed using AspectC++ to secure the connections and data transmission for the client application presented in Listing 3.5. For detailed information, a complete aspect for securing the connections of client applications using GNUTLS/SSL, but with different target functions, is presented in Listings 4.14 and 4.15 of Chapter 4. The first advice-pointcut matches the call to the content of the function `main` to initialize the GnuTLS API at the beginning and de-initialize it at the end. The second advice-pointcut intercepts all the calls to the function `connect`, initializes the TLS session before and adds the TLS handshake after. The third advice-pointcut intercepts all the calls to the function `send` and replaces each one by the function from TLS `gnutls_-record_send`. Similarly, the fourth advice-pointcut intercepts all the call to the function `recv` and replaces each one by the function from TLS `gnutls_record_recv`. The

```
#define MAX_MSG 100
const char * HTTPrequest = "GET / HTTP/1.1\nHost: www.encs.concordia.ca\
    n\n";

int dosend(int sd){
  int rc;
  rc = send(sd, HTTPrequest, strlen(HTTPrequest) + 1, 0);
  return rc;
}

int doreceive(int sd, char * buffer, unsigned int bufSize){
  int rc;
  rc = recv(sd, buffer, bufSize, 0);
  return rc;
}

int main (int argc, char *argv[]) {
  int sd, rc;
  int server_port = 443;
  struct sockaddr_in localAddr, servAddr;
  struct hostent *h;
  const char * server = "www.encs.concordia.ca";
  char buf[MAX_MSG];

  /*get host via DNS*/
  ...

  /*create socket data structure*/
  ...

  /* create socket */
  ...

  /* connect to server */
  rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));

  /* Sending*/
  rc = dosend(sd);

  /* Receiving
  rc = doreceive(sd, buf, MAX_MSG);

  /* Shutdown */
  close(sd);

  return 0;
}
```

Listing 3.5: Http Client

reader will notice also 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 parameters between the application components at runtime.

Weaving the application in Listing 3.5 with the aspect in Listing 3.6 using AspectC++ compiler produces the application presented in Listings 3.7 and 3.8. The resulting application supports https requests and data transmission through secure channels.

3.3.4 Remedying Buffer Overflow Vulnerabilities

Buffer overflow attacks exploit flaws that arise mostly from weak or non-existent bounds checking of input being stored in memory buffers. Attacks that exploit these vulnerabilities are considered as one of the most dangerous security threats since it can compromise the integrity, confidentiality and availability of the target system. We present in Listing 3.9 an aspect developed using AspectC++ as part of a complete case study applied on several software and presented in Chapter 4. This solution addresses and remedies three vulnerabilities exploited by buffer overflow. The three advices-pointcuts illustrated in Listing 3.9 match respectively all the calls to the functions `sprintf, gets, and strcat` and replace them by their corresponding secure ones `snprintf, fgets, and strncat`. Many safety vulnerabilities can be remedied in similar ways.

```
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();
}
//Same as the last advice for replacing recv() by gnutls_record_recv()
};
```

Listing 3.6: Excerpt of Aspect for Securing Connections

```
#define MAX_MSG 100
const char * HTTPrequest = "GET / HTTP/1.1 \nHost: www.encs.concordia.ca
    \n\n";

int dosend(int sd){
  int rc;

  hardening_sockinfo_t socketInfo;
  socketInfo = hardening_getSocketInfo(sd);
  if (socketInfo.isSecure)
    rc = gnutls_record_send(socketInfo.session, HTTPrequest, strlen(
        HTTPrequest) + 1);
  else
    rc = send(sd, HTTPrequest, strlen(HTTPrequest) + 1, 0);

  return rc;
}

int doreceive(int sd, char * buffer, unsigned int bufSize){
  int rc;

  hardening_sockinfo_t socketInfo;
  socketInfo = hardening_getSocketInfo(sd);
  if (socketInfo.isSecure)
    rc = gnutls_record_recv(socketInfo.session, buffer, bufSize);
  else
    rc = recv(sd, buffer, bufSize, 0);

  return rc;
}

int main (int argc, char *argv[]) {
  int sd, rc;
  int server_port = 443;
  struct sockaddr_in localAddr, servAddr;
  struct hostent *h;
  const char * server = "www.encs.concordia.ca";
  char buf[MAX_MSG];

  /*Initialization of the API*/
  ...
```

Listing 3.7: Weaved Http Client (Part 1)

```
  /*get host via DNS*/
  ...

  /*create socket data structure*/
  ...

  /* create socket */
  ...

  hardening_sockinfo_t socketInfo;
  const int cert_type_priority[3] = { GNUTLS_CRT_X509,
     GNUTLS_CRT_OPENPGP, 0};
  gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
  ...

  /* connect to server */
  rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));

  socketInfo.isSecure = true;
  socketInfo.socketDescriptor = sd; // socket is a variable matched by
     sd
  hardening_storeSocketInfo(sd, socketInfo);
  gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr)
     sd); // socket is a variable matched by sd
  rc = gnutls_handshake (socketInfo.session);

  /* Sending*/
  rc = dosend(sd);

  /* Receiving
  rc = doreceive(sd, buf, MAX_MSG);

  /* Shutdown */
  close(sd);

  /*De-initialization of the API*/
  ...

  return 0;
}
```

Listing 3.8: Weaved Http Client (Part 2)

```
aspect SafetyVul {

advice call("% sprintf(...)") : around () {
  snprintf((*(char **)tjp->arg(0)), strlen((*(char **)tjp->arg(0)))+1,
      (*(char **)tjp->arg(1)), (*(char **)tjp->arg(2)));
}

advice call("% gets(...)") : around () {
  if (fgets(((char *)tjp->arg(0)), strlen(((char *)tjp->arg(0)))-3,stdin
      ) == NULL) {
    printf("diagnosed undefined behavior.\n");
}

advice call("% strcat(...)") : around () {
  strncat((*(char **)tjp->arg(0)), (*(char **)tjp->arg(1)), strlen((*(
      char **)tjp->arg(0)))-strlen((*(char **)tjp->arg(1)))-1);
}

};
```

Listing 3.9: Aspect for Remedying some Safety Vulnerabilities

3.4 Aspect-Oriented Approaches for Improving Security

The research contributions in this domain are proposed as languages targeting security and/or case studies that explore the usefulness of AOP for developing and injecting security concerns into code. These propositions are useful initiatives towards separating the security code from the rest of the application code and systemizing their merging process. However, they target particular security vulnerabilities or requirements and show their corresponding AOP solutions (mostly in AspectJ). Besides, none of them proposed a global methodology for performing security hardening systematically by developers non-experts in the security solution applied. A developer still needs to build the security solution and specify it into aspect(s). The later statement remains true for all the current approaches, which we overview in the following.

Cigital labs proposed an AOP language called CSAW [74–76, 82], 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. Their high level aspects address the problems of event ordering, signal race condition, and type safety. This language is limited to C programming language and addresses priory defined set of related security vulnerabilities.

De Win et al. discussed two aspect oriented approaches and explored their use in integrating security aspects within applications [31–33, 81]. 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 for the external communication of the applications. Their second approach, the weaving-based Aspect-Oriented Software Development (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 join points 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 AspectJ to enforce access control and modularize the audit and access control features of an FTP server. This proposition is limited on exploring the usability of using AspectJ to implement access control concerns and integrate them into Java applications.

In [20], 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. This proposition is limited on exploring the usability of using AspectJ to implement authentication and authorization concerns and integrate them into Java applications.

Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang et al. [52] 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 pointcuts. This approach is a useful first step, however it still requires the developer to be a security expert who knows exactly where each piece of code should be injected. Moreover, its claimed goal is to prove the feasibility of reusing and integrating pre-built aspects.

Shlowikowski and Ziekinski discussed in [78] 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. This proposition is limited on exploring the usability of using AspectJ to implement some security concerns and integrate them into Java applications.

In [48], Hermosillo et al. proposed a security aspect called AProSec for detecting SQL injection and Cross Scripting Site (XSS). AProSec has been developed with AspectJ and JBoss to be weaved with web server applications. This aspect allows to intercept on the server side the HTTP requests parameters and the data base queries and pass them to a validation process depending on the options that the administrator selects in a configuration file. This proposition is limited on exploring the usability of using AspectJ to detect SQL injection into Java applications.

3.5 Conclusion

This chapter constitutes an introduction to our main approach for systematic security hardening. We described AOP in detail and explored through practical examples its usability for injecting security concerns into software. We have concluded that AOP is a very promising paradigm that provides the required features and offers strong potential for systematically injecting security code into software. However, the current corpus of research in AOP-based security is still recent and falls short of a well-defined and organized solution usable for applying security hardening systematically by non-experts. A developer needs to build the security solution and specify it into aspect(s), which still requires high security expertise. Hence, adopting the AOP concept and benefiting from the advantages of the other approaches presented in Chapter 2 constitute a base to build on top of it and elaborate the intended framework for the systematic security hardening of software.

Chapter 4

Aspect-Oriented and Pattern-Based Approach for Security Hardening

4.1 Introduction

Software security hardening 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 (e.g., security patterns, secure coding, etc.). However, security hardening is a difficult and critical procedure. Applying it manually requires high security expertise and lot of time to be tackled. Other vulnerabilities may also be created. Moreover, there is a problem resulting from the difficulty in finding the software engineers and developers who are specialized in both the security solution domain and the software functionality domain. In fact, this is an open problem raised by several IT managers (e.g., Bell Security Labs, Ericsson Research Labs). As such, any attempt to address security concerns must take into consideration the aforementioned problems. In this context, the main intent of this research is to create methods and solutions to integrate systematically and consistently security models and components into software.

One way of achieving these objectives is by separating out the security concerns from the rest of the application, such that they can be addressed independently and applied globally. 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 [20,31,52,74]. AOP is an appealing approach that allows the separation of crosscutting concerns. This paradigm seems to be very promising to integrate security into software.

Our approach is based on AOP and inspired by the 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 propose a language called *SHL* (Security Hardening Language) 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 chapter provides the core approach, components, language, compiler and implementation of the elaborated framework for the systematic security hardening of software. It also presents the case studies and experimental results that explore the usability and relevance of the proposed approach and framework. The remainder of this chapter is organized as follows. In Section 4.2, we provide an overview of the approach for systematic security hardening. Afterwards, we describe the framework components, i.e., the security hardening plans, patterns and language (*SHL*), in Sections 4.3, 4.4 and 4.5 respectively. We also present in Section 4.5 the grammar, structure and informal semantics of *SHL*. Then, we provide the *SHL* compilation phases and the framework implementation methodology in Section 4.6. After that, in Section 4.7, we illustrate the usability of the security hardening framework into case studies for different security issues and problems. Finally, we offer concluding remarks in Section 4.8.

4.2 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 elaborated an approach based on aspect orientation to perform security hardening in a systematic way. The approach architecture is illustrated in Figure 4.

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 requirements. These plans contains the abstract actions required for security hardening and uses the security hardening patterns that

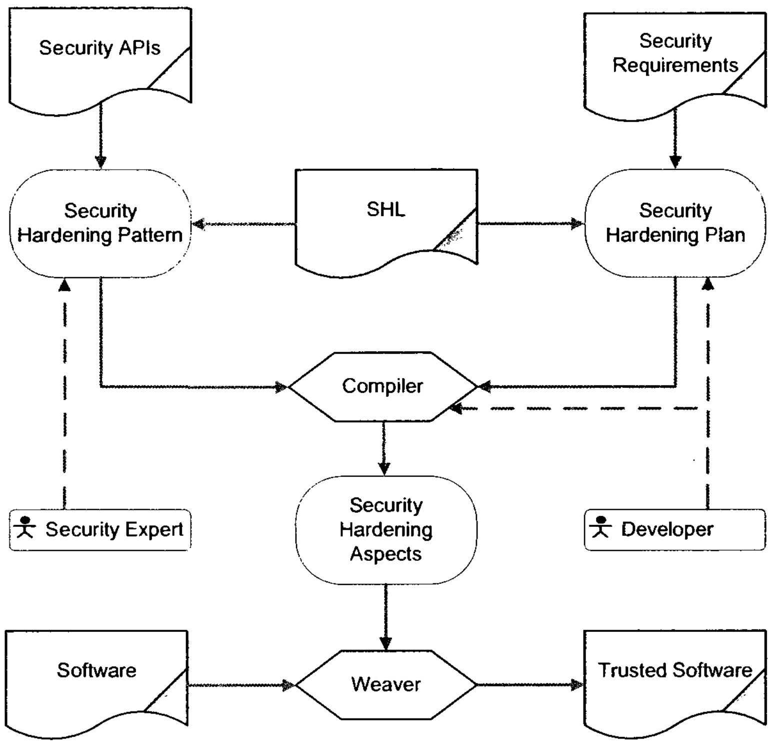


Figure 4: Framework Architecture

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 is applied in an organized and systematic way. 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*. 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 given parameters, allow to specify the precise steps to be performed for security hardening, taking into consideration technological issues such as platforms, libraries and languages. We built *SHL* on top of the current AOP languages.

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 security expertise. Afterwards, the framework compiler can be used to build and run the corresponding hardening plan and pattern and execute the appropriate AOP weaver (e.g., AspectJ, AspectC++) to harden the aspects into the original source code. 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.

4.3 Security Hardening Plans

A security assessment brings any decision-maker to perform a risk analysis, which will finally determine the security requirements. A given set of security requirements may be implemented through different combinations of mechanisms and software improvements. As such, a software developer must select the combination of solutions deemed optimal for specific program to harden. This decision is written in a security hardening plan, effectively translating such requirements into a specification of software modifications. The plans are written as a list of parameterized patterns with a `Where` clause, which indicates where in the software the pattern is to be applied. Each pattern is responsible to document the parameters it requires and supports. The developer is able to write the different hardening plans that are required for each part of the software. Example of a hardening plan is presented in Listing 4.10.

4.4 Security Hardening Patterns

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. Security hardening patterns specify the steps and actions needed to harden systematically security into the code. A security hardening pattern may also contain additional information illustrated into a template similar to the one used for security design patterns (Please see Chapter 2 for more information). This additional information allows the user to understand the solution provided by the pattern, its applicability, advantages, limitations, etc. Example of a hardening pattern is presented in Listings 4.11 and 4.12.

4.5 Security Hardening Language *SHL*

The elaborated language, *SHL*, allows the description and specification of security hardening patterns and plans that are used to systematically harden the security of 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 4.7. 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 and systematic way.
- Description and specification of security hardening.

- Description and specification of reusable security hardening patterns and plans.
- Instantiation of the security hardening patterns through the security hardening plans.
- Independency of programming language.
- High expressiveness and facility to use by non-security experts.
- Intermediary abstractness between English and programming languages.
- Ease translation to available AOP languages (e.g. AspectJ and AspectC++).

4.5.1 Grammar

In this section, we present the syntactic constructs of *SHL* and their informal semantics. Figure 5 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 using *SHL* for specifying security hardening plans and patterns are presented in Section 4.7.

Hardening Plan Structure A hardening plan always starts with the keyword `Plan`, followed by the plan name and then the plan code that starts and ends respectively by the keywords `BeginPlan` and `EndPlan`. Regarding the plan code, it consists 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

```
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 | Integer | Collection
Collection                 ::=  { ( Identifier | Integer) (, Identifier | Integer) * }
Module_Identification      ::=  Identifier

SH_Pattern                 ::=  Pattern Pattern_Name
                                Matching_Criteria?
                                SH_Pattern_Body
Matching_Criteria          ::=  Parameters Pattern_Parameter+
SH_Pattern_Body            ::=  BeginPattern
                                Location_Behavior*
                                EndPattern
Location_Behavior          ::=  Behavior_Insertion_Point Location
                                Primitive*?
                                Behavior_Code
Behavior_Insertion_Point   ::=  Before
                           |    After
                           |    Replace
Location                   ::=  Location_Identifier | Boolean_Location
Location_Identifier        ::=  FunctionCall <Signature> (Arguments) ?
                           |    FunctionExecution <Signature> (Arguments) ?
                           |    WithinFunction <Signature> (Arguments) ?
                           |    CFlow <Location_Identifier>
                           |    GAFlow <Location_Identifier>
                           |    GDFlow <Location_Identifier>
                           |    ...
Boolean_Location           ::=  Location and Location
                           |    Location or Location
                           |    not Location
Signature                  ::=  Identifier
Primitive                  ::=  ExportParameter <Identifier>
                           |    ImportParameter <Identifier>
                           |    ...
Arguments                  ::=  ( Star_Or_Identifier (, Star_Or_Identifier)* )
Behavior_Code              ::=  BeginBehavior
                                Code_Statement
                                EndBehavior
```

Figure 5: *SHL* Grammar

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 name, then the keyword `Parameters` followed by the matching criteria and finally the pattern 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 consists of one or many `Location_Behavior` constructs. Each one of them constitutes the location 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 code will be illustrated in Section 4.5.2.

4.5.2 Informal Semantics

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

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 advice-pointcut combination in an aspect. A pattern may include one or many `Location_Behavior`. Each `Location_Behavior` is composed of the `Behavior_Insertion_Point`, `Location`, 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 left opened for any needed extension. 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 specified signature is called.
- `FunctionExecution <Signature>`: Provides all the join points referring to the implementation of a function matching the specified signature.
- `WithinFunction <Signature>`: Filters all the join points that are within the functions matching the specified signature.
- `CFlow <Location>`: Captures the join points occurring in the dynamic execution context of the join points specified in the input `Location_Identifier(s)`.
- `GAflow <Location>`: Operates on the control flow graph (CFG) of a program. Its input is a set of join points defined as a `Location` 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>`: Operates on the CFG of a program. Its input is a set of join points defined as a `Location` 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` constructs can be composed with algebraic operators to build up the `Boolean_Location` as follows:

- `Location && Location`: Returns the intersection of the join points specified in the two `Location_Identifier` constructs.
- `Location || Location`: Returns the union of the join points specified in the two `Location_Identifier` constructs.
- `! Location`: Excludes the join points specified in the `Location_Identifier` construct.

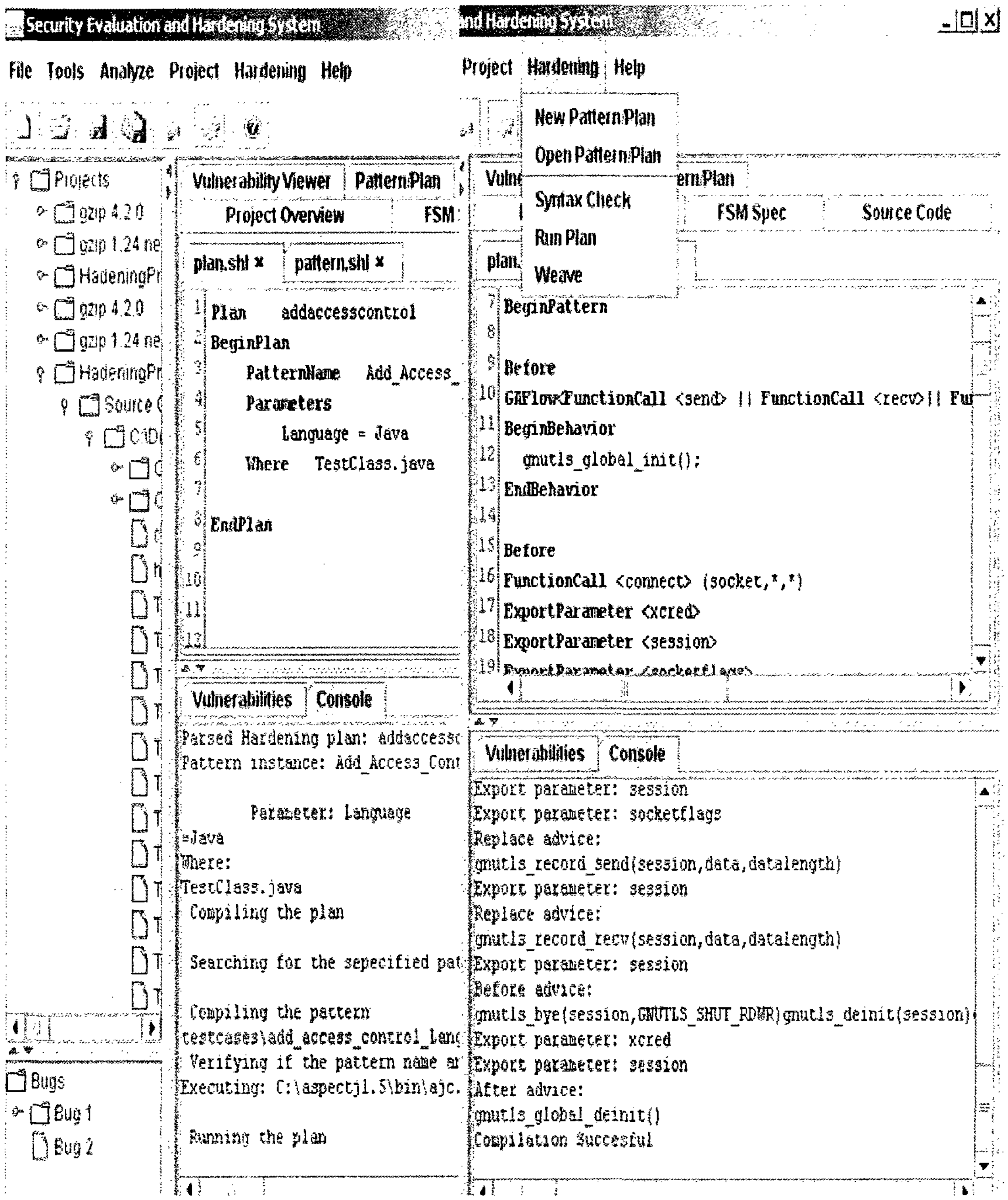
Primitive Is an optional functionality that allows to specify the variables that should be passed between two `Location` constructs. The following are the constructs responsible of passing the parameters:

- `ExportParameter <Identifier>`: Defined at the origin `Location`. It allows to specify a set of variables and make them available to be exported.
- `Importparameter <Identifier>`: Defined at the destination `Location`. It allows to specify a set of variables and import them from the origin `Location` where the `ExportParameter` has been defined.

Behavior_Code May contain code written in any language, programming language, or even informal e.g., English 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 security expertise. Example of such code behavior is presented in Listings 4.11 and 4.12 in Section 4.7.

4.6 *SHL* Compiler and Framework Implementation

We implemented the BNF specification of *SHL* using ANTLR V3 Beta 6 and its associated ANTLRWorks development environment [66]. 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 plan 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 6 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 automatically. The success of one phase leads to execute the next one. In the sequel, we present and explain these phases.



```
Plan    addaccesscontrol
BeginPlan
    PatternName   Add_Access_
    Parameters
        Language = Java
    Where   TestClass.java

EndPlan
```

```
BeginPattern


Before
GAFlow<FunctionCall <send> || FunctionCall <recv>|| Fu
BeginBehavior
  gnutls_global_init();
EndBehavior


Before
FunctionCall <connect> (socket,*,*)
ExportParameter <xcred>
ExportParameter <session>
```

```
Parsed Hardening plan: addaccessc
Pattern instance: Add_Access_Cont


        Parameter: Language
=Java
Where:
TestClass.java
 Compiling the plan

 Searching for the sepecified pat

 Compiling the pattern
testcases\add_access_control_Lan(
 Verifying if the pattern name ar
Executing: C:\aspectj1.5\bin\ajc.

 Running the plan
```

```
Export parameter: session
Export parameter: socketflags
Replace advice:
gnutls_record_send(session,data,datalength)
Export parameter: session
Replace advice:
gnutls_record_recv(session,data,datalength)
Export parameter: session
Before advice:
gnutls_bye(session,GNUTLS_SHUT_RDWR)gnutls_deinit(session)
Export parameter: xcred
Export parameter: session
After advice:
gnutls_global_deinit()
Compilation Succesful
```

Figure 6: Screenshot of the Security Hardening System

Plan Compilation

This phase consists 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 the bug. This statement also applies 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 the authentication of Java will be named *AuthenticationJava.SHL*, while another one of 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 selected weaver.

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 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 re-integrated in the original software that requires to be re-built for the hardening to take place.

Aspect Generation

Aspect generation is an additional feature launched separately to assist the developer during the refinement of a pattern by generating automatically part of the corresponding aspect. The same aforementioned compilation and matching mechanisms are used to compile the pattern specified in the plan. Then, each `Location_Behavior` in the pattern is refined into a combination of pointcut declarations and an advice that contains the same body as the one of the `Behavior_Code`. The generated poincuts and advices are enclosed into an aspect that has the same name as the pattern concatenated to its parameters and saved in a file with extension (.ah) for AspectC++ or (.aj) for AspectJ. The developer will have to refine the advice bodies into programming language code (i.e, C++ or Java) and then run the plan to apply the weaving.

4.7 Case Studies: Plans, Patterns and Aspects for Security Hardening

We demonstrated the feasibility of our approach and framework 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, C++ and Java and example code that implement the security hardening of the aforementioned requirements and vulnerabilities. We will show some of our findings here.

4.7.1 Hardening of Secure Connection Features into APT

In this section, we illustrate our elaborated solutions for securing the connections of client applications by following our methodology and using the proposed *SHL* language and its corresponding framework. Securing channels between two communicating parties allows to avoid eavesdropping, tampering with the transmission, and session hijacking. In this context, we selected an open source software called APT to add HTTPS support and secure its connections using GnuTLS/SSL library [4]. We also applied similar experiments on client applications that we developed.

APT is an automated package downloader and manager for the Debian Linux distribution [1]. 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. APT is organized in few components that allow extensibility. All package acquisition methods are separated from the package management logic and are grouped individually as programs. The library (`libapt`) creates the process of the method and communicates with it using the standard input and output of the created process. The library sends acquisition requests to the method, which will parse and process it. The acquisition method is responsible of writing the downloaded files to disk. The functions of the library can be used by different software packages, but the source code includes many command-line tools. In our case, we used the `apt-get` command-line tool and we created an HTTPS method based on the existing HTTP method. In the sequel, we are going to present the hardening plans, pattern and aspect elaborated to secure the connections of APT.

SHL Hardening Plan

In Listing 4.10, we include an example of effective security hardening plan specified in *SHL* for securing the connection of APT. It contains the name of the pattern to select (`Secure_Connection_Pattern`), parameters (`Language, API`, etc.) and components/files where to apply the pattern (`http.cc` and `connect.cc`).

```
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
```

Listing 4.10: *SHL* Hardening Plans for Securing Connection

SHL Hardening Pattern

Listing 4.11 and Listing 4.12 present the pattern elaborated in *SHL* for securing the connection of client applications using GnuTLS/SSL. It contains the pattern name (`Secure_Connection_Pattern`), the parameters (`Language, API`, etc.) and a list of `Location_Behaviors`. Each `Location_Behavior` starts with a `Behavior_Insertion_Point`, followed by a `Location`, a `Primitive` and a `Behavior_Code`. The first `Location_Behavior` matches the beginning of the function `HttpMethod::Loop` to initialize the library, the second `Location_Behavior` matches before the calls to the functions `connect` to initialize the session, the third `Location_Behavior` matches after the calls to the functions `connect` to perform the handshake, the fourth `Location_Behavior` matches the calls to the functions `write` to replace them by the secure ones, the fifth `Location_Behavior` matches the calls to the functions `read` to replace them by the secure ones, the sixth `Location_Behavior` matches before the calls to the functions `close` to close the session and finally the seventh `Location_Behavior` matches the end of the function `HttpMethod::Loop` to de-initialize the library.

The code of the functions used in the `Behavior_Code` parts of the pattern is illustrated in Listing 4.13. 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 and 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 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.

Hardening Aspect

We refined and implemented (using AspectC++) in Listing 4.14 and Listing 4.15 the corresponding aspect of the pattern presented in Listing 4.11 and Listing 4.12. The first advice-pointcut matches the content of the function `HttpMethod::Loop` to initialize the GnuTLS API at the beginning and de-initialize it at the end. The second advice-pointcut intercepts all the calls to the function `connect`, initializes the TLS session before and adds the TLS handshake after. The third advice-pointcut intercepts all the calls to the function `write` and replaces each one by the TLS function `gnutls_record_send`. Similarly,

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

Before
FunctionExecution <HttpMethod::Loop> //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 <write>
ImportParameter <session>
```

Listing 4.11: *SHL* Hardening Pattern for Securing Connection (Part 1)

```
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 <read>
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 <HttpMethod::Loop>
BeginBehavior
    // Deinitialize the TLS library
    DeinitializeTLSLibrary;
EndBehavior

EndPattern
```

Listing 4.12: *SHL* Hardening Pattern for Securing Connection (Part 2)

```
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();
```
Listing 4.13: Functions Used in the Pattern for Secure Connection

the fourth advice-pointcut intercepts all the calls to the function `read` and replaces each one by the TLS function `gnutls_record_recv`. Finally, the fifth advice intercepts all the calls to the function `close`, terminates the TLS session before and de-initializes the created data structure after performing the call.

The reader will notice the appearance of `hardening_sockinfo_t`. These are the data structures and functions that we developed to distinguish between secure and non secure channels and export the parameter between the application 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 functions.

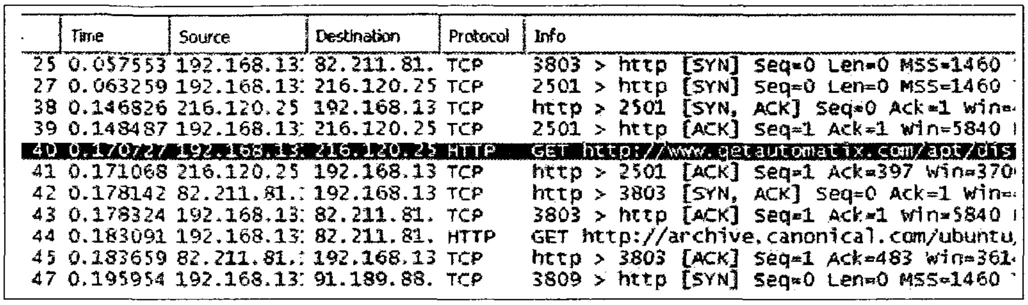
Experimental Results

In order to validate the hardened APT software, we used the Debian apache-ssl package [2], an HTTP server that accepts only SSL-enabled connections. We populated the server with a software repository compliant with APT 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 APT. The resulting hardened software was capable of performing both HTTP and HTTPS

```
aspect https{

advice execution( "% HttpMethod::Loop()" ) : around () {
//init gnutls lib
hardening_initGnuTLSSubsystem(NONE); hardening_socketInfoStorageInit();
tjp->proceed();
//deinit libs
hardening_socketInfoStorageDeinit(); hardening_deinitGnuTLSSubsystem();
}

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);
       gnutls_set_default_priority (socketInfo.session);
    gnutls_certificate_type_set_priority (socketInfo.session,
       cert_type_priority); gnutls_certificate_allocate_credentials (&
       socketInfo.xcred);
    gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
       socketInfo.xcred);

    //check if non-blocking. If so, make blocking until we are done with
        the handshake
    int socketflags = fcntl(*(int *)tjp->arg(0),F_GETFL);
    if ((socketflags & O_NONBLOCK) != 0) fcntl(*(int *)tjp->arg(0),
       F_SETFL, socketflags ^ O_NONBLOCK);
    //Connect + Handshake
    tjp->proceed();
    if (*tjp->result() < 0) {
      if ((socketflags & O_NONBLOCK) != 0)  fcntl(*(int *)tjp->arg(0),
         F_SETFL, socketflags);
      return;
    }
    gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr)
        (*(int *)tjp->arg(0)));
    int result = gnutls_handshake (socketInfo.session);
    if ((socketflags & O_NONBLOCK) != 0){
      fcntl(*(int *)tjp->arg(0),F_SETFL, socketflags); //restore non-
         blocking state if it was like that
      gnutls_transport_set_lowat(socketInfo.session,0); //now make
         gnutls aware that we are dealing with non-blocking sockets
    }
```
Listing 4.14: Aspect for Adding HTTPS Functionality (Part 1)

```
    //Save Information in hash table
    socketInfo.isSecure = true; socketInfo.socketDescriptor = *(int *)
        tjp->arg(0);
    hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
    *tjp->result() = result;
}

//replacing write() by gnutls_record_send() on a secured socket
advice call("% write(...)") : around () {
  hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)tjp
      ->arg(0));
  if (socketInfo.isSecure)
     *(tjp->result()) = gnutls_record_send(socketInfo.session, *(char
         **) tjp->arg(1), *(int *)tjp->arg(2));
  else
     tjp->proceed();
}

//replacing read() by gnutls_record_recv() on a secured socket
advice call("% read(...)") : around () {
  hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)tjp
      ->arg(0));
  if (socketInfo.isSecure)
     *(tjp->result()) = gnutls_record_recv(socketInfo.session, *(char
         **) tjp->arg(1), *(int *)tjp->arg(2));
  else
     tjp->proceed();
}

advice call("% close(...)") : around () {
  hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)tjp
      ->arg(0)); /* socket matched by sd*/
  if(socketInfo.isSecure ){
     gnutls_bye(socketInfo.session, GNUTLS_SHUT_RDWR);
  }
  tjp->proceed();
  if(socketInfo.isSecure ){
     gnutls_deinit(socketInfo.session);
         gnutls_certificate_free_credentials(socketInfo.xcred);
     hardening_removeSocketInfo(*(int *)tjp->arg(0));
     socketInfo.isSecure = false; socketInfo.socketDescriptor = 0;
  }
}

};
```

Listing 4.15: Aspect for Adding HTTPS Functionality (Part 2)

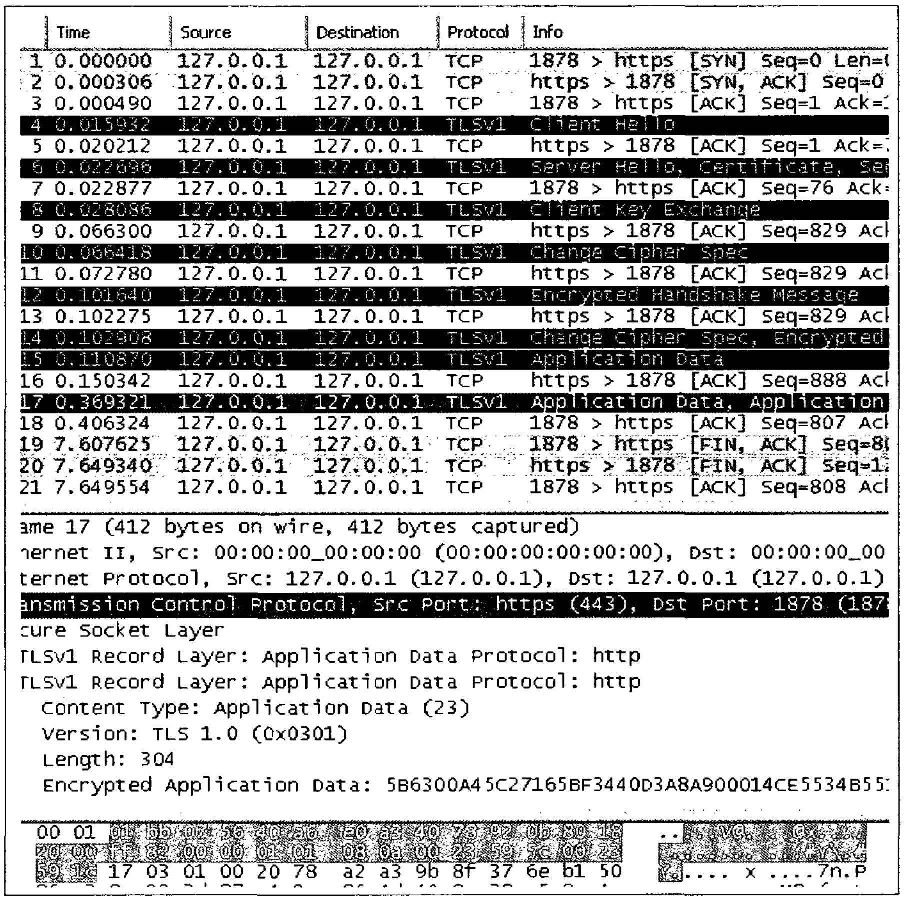


```
.   Time     Source          Destination     Protocol  Info
25 0.057553 192.168.13: 82.211.81.  TCP     3803 > http [SYN] Seq=0 Len=0 MSS=1460
27 0.063259 192.168.13: 216.120.25  TCP     2501 > http [SYN] Seq=0 Len=0 MSS=1460
38 0.146826 216.120.25  192.168.13  TCP     http > 2501 [SYN, ACK] Seq=0 Ack=1 Win=
39 0.148487 192.168.13: 216.120.25  TCP     2501 > http [ACK] Seq=1 Ack=1 Win=5840
40 0.170727 192.168.13: 216.120.25  HTTP    GET http://www.getautomatix.com/apt/dis
41 0.171068 216.120.25  192.168.13  TCP     http > 2501 [ACK] Seq=1 Ack=397 Win=370
42 0.178142 82.211.81.: 192.168.13  TCP     http > 3803 [SYN, ACK] Seq=0 Ack=1 Win=
43 0.178324 192.168.13: 82.211.81.  TCP     3803 > http [ACK] Seq=1 Ack=1 Win=5840
44 0.183091 192.168.13: 82.211.81.  HTTP    GET http://archive.canonical.com/ubuntu,
45 0.183659 82.211.81.: 192.168.13  TCP     http > 3803 [ACK] Seq=1 Ack=483 Win=361
47 0.195954 192.168.13: 91.189.88.  TCP     3809 > http [SYN] Seq=0 Len=0 MSS=1460
```

Figure 7: Packet Capture of Unencrypted APT Traffic

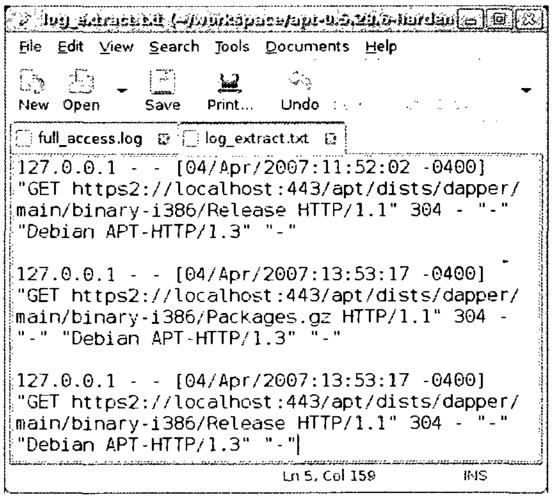
package acquisition, based on the parameters in the configuration file. After building and deploying the modified APT package, we tested successfully its functionality by refreshing APT package database, which forced the software to connect to both our local web server (Apache-ssl) using HTTPS and remote servers using HTTP to update its list of packages. The experimental results in Figures 7, 8, and 9 show that the new secure APT software is 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 7 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 8 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 9 shows an extract of Apache access log, edited here for conciseness, where the package metadata was successfully obtained from our local server by the hardened software.



```
   Time      Source     Destination  Protocol  Info
 1 0.000000  127.0.0.1  127.0.0.1    TCP       1878 > https [SYN] Seq=0 Len=(
 2 0.000306  127.0.0.1  127.0.0.1    TCP       https > 1878 [SYN, ACK] Seq=0
 3 0.000490  127.0.0.1  127.0.0.1    TCP       1878 > https [ACK] Seq=1 Ack=:
 4 0.015932  127.0.0.1  127.0.0.1    TLSv1     Client Hello
 5 0.020212  127.0.0.1  127.0.0.1    TCP       https > 1878 [ACK] Seq=1 Ack=:
 6 0.022696  127.0.0.1  127.0.0.1    TLSv1     Server Hello, Certificate, Se
 7 0.022877  127.0.0.1  127.0.0.1    TCP       1878 > https [ACK] Seq=76 Ack:
 8 0.028086  127.0.0.1  127.0.0.1    TLSv1     Client Key Exchange
 9 0.066300  127.0.0.1  127.0.0.1    TCP       https > 1878 [ACK] Seq=829 Ac
10 0.066418  127.0.0.1  127.0.0.1    TLSv1     Change Cipher Spec
11 0.072780  127.0.0.1  127.0.0.1    TCP       https > 1878 [ACK] Seq=829 Ac
12 0.101640  127.0.0.1  127.0.0.1    TLSv1     Encrypted Handshake Message
13 0.102275  127.0.0.1  127.0.0.1    TCP       https > 1878 [ACK] Seq=829 Ac
14 0.102908  127.0.0.1  127.0.0.1    TLSv1     Change Cipher Spec, Encrypted
15 0.110870  127.0.0.1  127.0.0.1    TLSv1     Application Data
16 0.150342  127.0.0.1  127.0.0.1    TCP       https > 1878 [ACK] Seq=888 Ac
17 0.369321  127.0.0.1  127.0.0.1    TLSv1     Application Data, Application
18 0.406324  127.0.0.1  127.0.0.1    TCP       1878 > https [ACK] Seq=807 Ac
19 7.607625  127.0.0.1  127.0.0.1    TCP       1878 > https [FIN, ACK] Seq=8(
20 7.649340  127.0.0.1  127.0.0.1    TCP       https > 1878 [FIN, ACK] Seq=1;
21 7.649554  127.0.0.1  127.0.0.1    TCP       1878 > https [ACK] Seq=808 Ac

ame 17 (412 bytes on wire, 412 bytes captured)
nernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00
ternet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
ansmission Control Protocol, Src Port: https (443), Dst Port: 1878 (187
cure Socket Layer
TLSv1 Record Layer: Application Data Protocol: http
TLSv1 Record Layer: Application Data Protocol: http
  Content Type: Application Data (23)
  Version: TLS 1.0 (0x0301)
  Length: 304
  Encrypted Application Data: 5B6300A45C27165BF3440D3A8A900014CE5534B55:

 00 01 01 bb 07 56 40 a6  e0 a3 40 78 92 0b 80 18   .....V@. ..@x....
 20 00 ff 82 00 00 01 01  08 0a 00 23 59 5c 00 23   ...........#Y\.#
 59 1c 17 03 01 00 20 78  a2 a3 9b 8f 37 6e b1 50   Y..... x ....7n.P
```

Figure 8: Packet Capture of SSL-protected APT Traffic



```
127.0.0.1 - - [04/Apr/2007:11:52:02 -0400]
"GET https2://localhost:443/apt/dists/dapper/
main/binary-i386/Release HTTP/1.1" 304 - "-"
"Debian APT-HTTP/1.3" "-"

127.0.0.1 - - [04/Apr/2007:13:53:17 -0400]
"GET https2://localhost:443/apt/dists/dapper/
main/binary-i386/Packages.gz HTTP/1.1" 304 -
"-" "Debian APT-HTTP/1.3" "-"

127.0.0.1 - - [04/Apr/2007:13:53:17 -0400]
"GET https2://localhost:443/apt/dists/dapper/
main/binary-i386/Release HTTP/1.1" 304 - "-"
"Debian APT-HTTP/1.3" "-"
```

log_extract.txt (~/workspace/apt-0.5.28.6-hardened)
File Edit View Search Tools Documents Help
New Open Save Print... Undo
full_access.log
log_extract.txt
Ln 5, Col 159 INS

Figure 9: Excerpt of Apache Access Log

4.7.2 Hardening of Low-Level Security Vulnerabilities in MySQL

The C and C++ programming languages have been designed for maximal performance, at the expense of some safety-enhancing techniques. Keeping memory management left to the programmer discretion and the lack of type safety are the major causes of security vulnerabilities in C and C++ . Related security vulnerabilities and their causes have been published in books, papers and reviews. Detailed discussion about this issue has been provided in Chapter 2. In this section, we illustrate our elaborated solutions by adding several low-level security vulnerabilities into MySQL software and then remedying them through following the proposed methodology and using the *SHL* language and its corresponding framework. MySQL is a relational database management system that runs as a server providing multi-user access to a number of databases. It is written in C++ and is composed of more than 250 000 source lines of code. We added vulnerable pieces of code in diverse components that have direct impact on the execution of the software. This caused stack and memory crash and MySQL failed to continue its execution.

SHL Hardening Plan

In Listing 4.16, we include an example of effective security hardening plan specified in *SHL* for remedying low-level security vulnerabilities present in MySQL software. It contains the name of the pattern to select (`Safety_Vul_MySQL_Pattern`), the parameter (`Language`) and all the components/files of the application where to apply the pattern (`*.cc`).

```
Plan     Safety_Vul_MySQL_Plan
BeginPlan
    PatternName   Safety_Vul_MySQL_Pattern
    Parameters
            Language = C/C++
    Where   *.cc
EndPlan
```

Listing 4.16: *SHL* Hardening Plan for Remedying MySQL Safety Vulnerabilities

SHL Hardening Pattern

Listing 4.17 describes the hardening pattern elaborated in *SHL* for remedying several low-level security vulnerabilities present in MySQL software. It contains the pattern name (`Safety_Vul_MySQL_Pattern`), the parameter (`Language`) and a list of `Location_Behaviors`. Each `Location_Behavior` starts with a `Behavior_Insertion_Point`, followed by a `Location`, a `Primitive` and a `Behavior_Code`. The first four `Location_Behaviors` match the calls to the functions `sprintf, gets, strcpy and strcat` and replace them by their secure ones. These functions do not apply bound checking and verification on the string parameters and arguments. These missing features constitute major vulnerabilities exploited by the buffer overflow attacks. Replacing these functions by secure ones, which are also provided by newer versions of the `C` and `C++` libraries, is one solution to address such flaws. Another error related to memory management is double freeing the same pointer, causing memory corruption. The solution provided for such vulnerability consists of setting the pointer to NULL after freeing it, so in this case freeing it another time won't corrupt useable memory data. The fifth `Location_Behavior` matches after the calls to the functions `free` to set their arguments to NULL. The last addressed problems are related to renaming and accessing files. Setting privileges and adding access restrictions are know solutions for such problems. The last two `Location_Behaviors` match before the calls to the functions `rename` and `fopen` to address the aforementioned problems.

Hardening Aspect

We refined and implemented (using AspectC++) in Listing 4.18 the corresponding aspect of the pattern presented in Listing 4.17. The first four advices-pointcuts match respectively all the calls to the functions `sprintf, gets, strcpy and strcat` and replace them by their corresponding secure ones `snprintf, fgets, strncpy and strncat.` The fifth advice-pointcut matches before all the calls to the `free` and sets its argument (pointer) to NULL. The sixth advice-pointcut matches before all the calls to the function `rename` and add user and group privilege. The last advice-pointcut matches before all the calls to the function `fopen` and add access restrictions.

Experimental Results

In order to verify the provided hardening solutions, we ran first the vulnerable MySQL software and applied the attacks corresponding to the injected vulnerabilities. The software crashes and stop its execution. Then, we weaved our solutions into MySQL and ran again the same attacks. None of them succeeds and the hardened software continues its regular execution, exploring the absence of vulnerabilities, and hence the success of the security hardening process.

```
Pattern Safety_Vul_MySQL_Pattern
Parameters
     Language = C/C++

BeginPattern

Replace
FunctionCall <sprintf>
BeginBehavior
    snprintf;
EndBehavior

Replace
FunctionCall <gets>
BeginBehavior
    fgets;
EndBehavior

Replace
FunctionCall <strcpy>
BeginBehavior
    strncpy;
EndBehavior

Replace
FunctionCall <strcat>
BeginBehavior
    strncat;
EndBehavior

After
FunctionCall <free>
BeginBehavior
    SetArgumentPointertoNull;
EndBehavior

Before
FunctionCall <rename>
BeginBehavior
    getUserID;
    SetUserID;
    getGroupID;
    SetGroupID;
EndBehavior

Before
FunctionCall <fopen>
BeginBehavior
    SetUserMask;
EndBehavior
```

Listing 4.17: *SHL* Hardening Pattern for Remedying MySQL Safety Vulnerabilities

```
aspect SafetyVul {

advice call("% sprintf(...)") : around () {
  snprintf((*(char **)tjp->arg(0)), strlen((*(char **)tjp->arg(0)))+1,
      (*(char **)tjp->arg(1)), (*(char **)tjp->arg(2)));
}

advice call("% gets(...)") : around () {
  if (fgets(((char *)tjp->arg(0)), strlen(((char *)tjp->arg(0)))-3,stdin
      ) == NULL) {
    printf("diagnosed undefined behavior.\n");
}

advice call("% strcpy(...)") : around () {
  strncpy((*(char **)tjp->arg(0)), (*(char **)tjp->arg(1)), strlen((*(
      char **)tjp->arg(0))));
}

advice call("% strcat(...)") : around () {
  strncat((*(char **)tjp->arg(0)), (*(char **)tjp->arg(1)), strlen((*(
      char **)tjp->arg(0)))-strlen((*(char **)tjp->arg(1)))-1);
}

advice call("% free(...)") : after () {
  *(char **)(tjp->arg(0))=NULL;
}

advice call("% rename(...)") : before () {
  //Get the effective user of the running process.
  //This will be the program's user or group owner if setuid or setgid
      is used.
  uid_t init_uid = geteuid();
  gid_t init_gid = getegid();
  //Drop to the privileges of the user who is runnig the process.
  seteuid(getuid());
  setegid(getgid());
}

advice call("% fopen(...)") : before () {
  //Set the umask such that any files created won't allow the group or
      the world to read, write, or execute.
  umask(S_IRWXG | S_IRWXO);
}

};
```

Listing 4.18: Aspect for Remedying Safety Vulnerabilities

4.7.3 Adding Authorization to Applications

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 our methodology and using the *SHL* language and its corresponding framework. 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 of our hardening solution. We deal with one of them in this case study.

SHL Hardening Plan

In Listing 4.19, we include an example of effective security hardening plan specified in *SHL* for adding authorization into the aforementioned application. It contains the name of the pattern to select (`Add_Authorization_Pattern`), the parameters (`Language, API and Type`) and the component/file of the application where to apply the pattern (`test.java`).

```
Plan     Own_Add_Authorization_Plan
BeginPlan
    PatternName Add_Authorization_Pattern
    Parameters
            Language = Java
            API      = JAAS
            Type     = ACL
    Where   test.java
EndPlan
```

Listing 4.19: *SHL* Hardening Plan for Adding Authorization

SHL Hardening Pattern

Listing 4.20 describes the hardening pattern elaborated in *SHL* for adding authorization to the aforementioned application. It contains the pattern name (`Add_Authorization_-Pattern`), the parameters (`Language, API and Type`) and a `Location_Behavior`. The `Location_Behavior` starts with a `Behavior_Insertion_Point`, followed by a `Location`, a `Primitive` and a `Behavior_Code`. It matches the beginning of the method `dosomething` to get the user name, get the method permission and check for authorization. The Java code of the functions used in the `Behavior_Code` parts of the pattern is illustrated in Listing 4.21. 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.

```
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 4.20: *SHL* Hardening Pattern for Adding Authorization

```
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);
```

Listing 4.21: Functions Used in the Pattern for Adding Authorization

Hardening Aspect

We refined and implemented (using AspectJ) in Listing 4.22 the corresponding aspect of the pattern presented in Listing 4.20. The advice-pointcut match all the calls to the method `dosomething` and replace them by new ones that check for authorization before providing the same functionalities. The resulting access control aspect uses Java Authentication and Authorization service API (JAAS) for authorization. The rights are specified in a separate policy file. 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`.

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.

```
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();}
}

}
```

Listing 4.22: Excerpt of an Aspect for Adding Authorization

4.8 Conclusion

We addressed in this chapter the problems related to the current methodologies for performing software security hardening. In this context, we proposed an AOP and pattern-based approach for systematic security hardening. Our proposition allows the 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. Moreover, we realized the proposed approach by elaborating the *SHL* language needed to describe the security hardening plans and patterns and developing its corresponding parser, compiler and facilities. The resulting framework allows to develop the components of a security hardening solution and perform all its required procedures. Beside, we explored the feasibility of the proposed approach by elaborating several case studies of security hardening and applying them on large scale software.

Chapter 5

New Aspect-Oriented Constructs for *SHL* Targeting Security Concerns

5.1 Introduction

Our approach for systematic security hardening and the experiments presented in Chapter 4, together with other related proposals for security code injection via AOP [20,31,52,74,78], explored that AOP constitutes a promising paradigm for the systematic security hardening of software. However, AOP was not initially designed to address security issues, which resulted in some limitations in the current technologies [21,47,54,58,61]. Indeed, we were not able to apply some security hardening activities due to missing features. For instance, while implementing an AOP-based solution for securing the connections of client applications, we opted to intialize/de-initialize and build/de-build the data structures and objects needed for GNU/TLS security library in the *main* function (please see Listing 5.29 in Section 5.5.3 for more detail). Although this solution works for small size applications with single features, it is not applicable and relevant for large scale applications with multiple functionalities. Many APIs initialization and unneeded operations may be performed, even if their corresponding features (i.e., the features using them) are not called during an execution context of a program. Such solution could also be ruinable for embedded applications, where the energy and memory resources are limited.

Moreover, during our security hardening experiments, we faced the problem of passing needed variables and parameters related to GNU/TLS library (e.g., TLS Session) between the application components. Such limitations forced us, when applying security hardening practices, to perform programming gymnastics (when possible), resulting in integrating additional modules and changing several functions in the application to pass the needed variables (please see Listing 5.29 in Section 5.5.3 for more detail). Such solution is not realistic in the case of large scale applications with multiple features, where there are complex dependencies and relations between their components. Any changes in one component lead to apply several modifications in all its dependent ones, which requires many complex re-engineering actions to be performed.

In this chapter, we present new pointcuts and primitives to *SHL* and AOP languages that are needed for systematic hardening of security concerns. The two proposed pointcuts allow the identification of particular join points in a program control flow graph (CFG). The first one is the *GAFlow*, Closest Guaranteed Ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second one is the *GDFlow*, Closest Guaranteed Descendant, which returns the closest child join point that can be reached by all paths starting from the pointcut of interest. The two proposed primitives are called *ExportParameter* and *ImportParameter* and are used to pass parameters between two pointcuts. They allow to analyze a program call graph in order to determine how to change function signatures for passing the parameters associated with a given security hardening task.

We find these pointcuts and primitives to be necessary because they are needed to perform many security hardening practices and, to the best of our knowledge, none of the existing AOP pointcuts and primitives and their combinations can provide their functionalities. Although, the interest of the proposed pointcuts and primitives may cover other domains, we restrict ourselves to security and discuss only the utilities related to software security hardening. Moreover, we show the viability of the proposed pointcuts and primitives by elaborating and implementing their methodologies and algorithms and presenting the result of explanatory case studies.

This chapter provides the new contributions toward developing our AOP-based framework for systematic security hardening framework. Adopting AOP in our approach makes enriching the AOP technology and *SHL* with new poincuts and primitives for security hardening concerns an essential task to reach our objectives. The remainder of this chapter is organized as follows. Section 5.2 explores the limitations associated with the current AOP technologies for security as well as the related security pointcuts proposed for these concerns. Then, a brief background on the program representation is presented in Section 5.3 and the proposed pointcuts and primitives are defined and specified in Section 5.4. Afterwards, the usefulness of our propositions and their advantages are discussed in Section 5.5. In Section 5.6, the algorithms necessary for implementing the proposed pointcuts and primitives are presented. This section also shows the implementation results via case studies. We move on to the conclusion in Section 6.8.

5.2 Security-Related Pointcuts

Our experiments explored the usefulness of AOP in reaching the objective of having systematic security hardening. On the other hand, we have also distinguished, together with other documented related work [21, 47, 54, 58, 61], the limitations of the available AOP technologies and languages for some security issues. Addressing such limitations can be achieved by elaborating pointcuts and primitives that improve the conditions on which we can inject appropriately the security code. Many authors have made contributions in this field, which we will list now.

A dataflow pointcut that is used to identify join points based on the origin of values is defined and formulated in [58] for security purposes. The authors expressed the usefulness of their pointcut by presenting an example on sanitizing web-applications. For instance, such a pointcut can detect if the data sent over the network depends on information read from a confidential file. This poincut is not fully implemented yet.

In [47], Harbulot and Gurd proposed a model of a loop pointcut that explores the need for a loop join point that predicts infinite loops, which are used by attackers to perform denial of service attacks. Their approach for recognizing loops is based on a control-flow analysis at the bytecode level in order to avoid ambiguities due to alternative forms of source-code that would produce identical loops. This model contains also a context exposure mechanism for writing pointcuts that select only specific loops.

In [21], Bonér discussed a poincut that is needed to detect the beginning of a synchronized block and add some security code that limits the CPU usage or the number of executed instructions. The author also explores the usefulness of capturing synchronized blocks in calculating the time acquired by a lock and thread management. This result can also be applied in the security context and can help in preventing many denial of service attacks.

A predicted control flow (`pcflow`) pointcut was introduced by Kiczales in a keynote address [54] without a precise definition. Such pointcut may allow to select points within the control flow of a join point starting from the root of the execution to the parameter join point. In the same presentation, an operator is introduced in order to obtain the minimum of two `pcflow` pointcuts, but it is never clearly defined what this minimum can be or how can it be obtained. These proposals could be used for software security, in the enforcement of policies that prohibit the execution of a given function in the context of the execution of another one.

Local variables set and get poincuts were introduced in [46] for increasing the efficiency of AOP for security concerns. They allow to track the values of local variables inside a method. It seems that these poincuts can be used to protect the privacy and integrity of sensitive data. Their idea is based on the approach presented in [61], which describe an extension of Java called JFlow. This language allows to statically checks information flow annotations within programs and provides several new features such as decentralized label model, label polymorphism, run-time label checking and automatic label inference. It also supports objects, sub-classing, dynamic type tests, access control, and exceptions.

Aberg et al. presented in [9] an aspect system that addresses the crosscutting of event notifications scattered over kernel code to support Bossa, an event-based framework for process-scheduler development. This aspect system uses temporal logic to precisely describe code insertion points and sequences of instructions that require events to be inserted.

In each case, the choice of event depends on properties of one or a sequence of instructions. They propose to guide the event insertion by using a set of rules, amounting to an aspect, that describes the control flow contexts in which each event should be generated.

In a position paper [29], Cottenier et al. argued that Aspect-Oriented Modeling (AOM) technologies have the potential to simplify the deployment and the ability to reason about a category of crosscutting concerns that have been categorized in the literature as stateful aspects. Stateful aspects trigger on a sequence of join points instead of a single join point. They identified three properties of AOM languages that enable them to provide more natural solutions to the stateful aspect problem. They also presented a JAsCo aspect example that captures a sequence of events(e.g., `methodA - methodB - methodC`) and attaches an advice to the last event (i.e., methodC).

5.3 Program Representation

Our propositions, together with their corresponding algorithms, are based and operate on the control flow and call graphs representation of software. In this context, we present in the following a brief background and some references to familiarize the reader with these concepts.

5.3.1 Control Flow Graphs

A control flow graph (CFG) is a representation, using graph notation, of all possible flow of execution that might be traversed through a program. A CFG is a cyclic directed graph that supports loops. Each node in the graph represents a basic block, which is composed of one or more code statements without branching. Most CFG representations have the entry block, through which control enters in the flow graph, and the exit block, through which all control flows leave. The directed edges of the graph are possible transitions from one basic block to another in the control flow, typically due to a conditional branching (e.g. `if`) or a function call. It is not possible to determine which path will be executed without the use of other techniques (e.g., data flow analysis). Control flow graphs can be used in optimizing compilers [10] as well as for certain static analysis methods [72].

Here are some references related to algorithms that operate on CFG and that can be useful for the elaborated algorithms of the pointcuts proposed in this chapter. In [28], the authors proposed a simple and fast algorithm to calculate the dominance information (e.g. dominator set) of CFG nodes. A dominator set of a node *n* (i.e., *Dom(n)*) contains the nodes that lie on every path from the entry node of the CFG to *n*. They also surveyed most of the related algorithms and approaches and compared them to their proposition. An implementation of one of these algorithm (Class DominaceInfo) has been provided in [49] as part of the Machine-SUIF control flow analysis (CFA) library. It is built on top of the control flow graph (CFG) [49] library and provides dominance analysis and natural-loop analysis. Other approaches that use lattice theory allow to efficiently compute a Lower Upper Bound (LUB) ancestor and Greater Lower Bound (GLB) descendant over lattices [12]. However, their results do not guarantee that all paths will be traversed by the results of LUB and GLB, which is a central requirement for our related propositions. Moreover, the lattices do not support the full range of expression provided by the CFG, as the latter can be a directed cyclic graph.

5.3.2 Call Graphs

A call graph is a potentially cyclic directed graph that is used to represent the calling structure between a program routines. Each node in a call graph represents a routine (procedure) and each edge (a,b) indicates that routine "a" calls routine "b". A cycle in the graph indicates recursive procedure calls. Call graphs can be dynamic or static. A dynamic call graph only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program, which means every call relationship that occur is represented in the graph. Call graphs can also be either context sensitive or context-insensitive. In a context-sensitive graph, for each procedure, the graph contains a separate node for each call stack this procedure can be activated with. In a context-insensitive, there is only one node for each procedure and all the calls targeting this procedure are related to this node. Context-insensitive call graph construction algorithms, such as the ones proposed by Ryder [69], do not take into consideration the value of the variables used to call the functions. The elaborated algorithms of the primitives proposed in this chapter operate on context-insensitive call graphs.

We provide here some references for algorithms that operate on call graphs. Ryder [69] provided one of the earliest contributions for efficient context-insensitive call graph construction in procedural languages, and this contribution was quickly followed by the notion of context sensitivity by Callahan et al. [25]. The construction of call graphs has been documented by Grove et al. [45] in the case of object-oriented languages, and an elaborated study of different algorithms was provided by Grove and Chambers in [44].

5.4 Pointcut and Primitive Definitions

In this section, we define the syntax and definitions of the proposed pointcuts and primitives. Table 1 illustrates the syntax that defines a pointcut p and an advice declaration after adding *GAFlow*, *GDFlow*, *ExportParameter* and *Importparameter*.

```
p ::= call(s) | execution(s) | GAFlow(p) | GDFlow(p) | p||p |
p&&p

advice <p> :  (before|after|around) [:  e | i | e,i]
{<advice-body>}

e ::= ExportParameter(<paramList>)
i::= ImportParameter(<paramList>)
paramList ::= parameter [,paramList]
parameter ::= <type> <identifier>
```

Table 1: Syntax of the Pointcuts and Primitives

A function signature is denoted by *s*. The *GAFlow* and the *GDFlow* are the new control flow based pointcuts. Their parameters are also pointcuts. The new primitives *ExportParameter* and *Importparameter* are *e* and *i* respectively. The arguments of *ExportParameter* are the parameters to pass, while the arguments of *ImportParameter* are the parameters to receive. In the following, we present the definition of each pointcut and primitive.

5.4.1 GAFlow and GDFlow Pointcuts

The *GAFlow* pointcut operates on the CFG of a program. Its input is a set of join points defined as a pointcut and its output is a single join point. 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 is (1) the closest common parent node of all the nodes specified in the input set and (2) through which all the possible paths that reach them pass. In the worst case, the closest common ancestor will be the starting point in a program.

The *GDFlow* pointcut operates on the CFG of a program. Its input is a set of join points defined as a pointcut and its output is a single join point. 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 the common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes. In the worst case, the first common descendant will be the end point in a program.

5.4.2 ExportParameter and ImportParameter Primitives

The *ExportParameter* and *ImportParameter* primitives operate on the call graph of a program to pass parameters between two pointcuts. They should always be combined and used together in order to provide the information needed for parameter passing from one join point to another. The origin node is the join point where *ExportParameter* is called, while the destination node is the join point where *ImportParameter* is called.

5.5 Discussion

This section discusses the usefulness, advantages and limitations of the proposed pointcuts and primitives.

5.5.1 Usefulness of GAFlow and GDFlow for Security Hardening

Many security hardening practices require the injection of code around a set of join points or possible execution paths [16, 50, 73, 87]. Examples of such cases would be the injection of security library initialization/deinitialization and data structure construction, privilege level changes, atomicity guarantee, logging, etc. The current AOP models allow us only to identify a set of join points in the program, and therefore inject code before, after and/or around each one of them. However, to the best of our knowledge, none of the current pointcuts enables the identification of a join point, common to a set of other join points and satisfying the criteria of *GAFlow* et *GDFlow*, where we can inject the code when it is needed and once for all of them. In the sequel, we present briefly the necessity and usefulness of our proposed pointcuts for some security hardening activities.

Security Library Initialization/Deinitialization and Data Structure Construction

During the development of an AOP-based solution for securing the connections of client applications, we intialiazed/de-initialized and built/de-built the data structures and objects needed for GNU/TLS security library in the *main* function. Such solution works for small size applications with single features. However, it is not relevant for large scale applications with multiple functionalities. Many APIs initialization and unneeded operations may be performed, even if their corresponding features are not called during an execution context of a program. In the case of embedded applications where the energy and memory resources are limited, such solution could be ruinable. The proposed pointcuts allow to solve this problem by executing these operations for the branches of code where they are needed by identifying their *GAFlow* and/or *GDFlow*. Having both pointcuts would also avoid the need to keep global state variables about the current state of library initialization. We use as an example a part of an aspect that we elaborated for securing the connections of a client application. With the current AOP pointcuts, the aspect targets the main function as the location for the TLS library initialization, deinitialization and data structure construction, as depicted in Listing 5.23. In listing 5.24, we see an improved aspect targeting the pointcuts *GAFlow* and *GDFlow* to perform these operations and offering more efficient and wider applicable results.

```
advice execution ("% main (...) ") : around () {
  hardening_socketInfoStorageInit ();
  hardening_initGnuTLSSubsystem(NONE);
  tjp -> proceed ();
  hardening_deinitGnuTLSSubsystem ();
  hardening_socketInfoStorageDeinit();
  *tjp -> result () = 0;
}
```

Listing 5.23: Excerpt of Hardening Aspect for Securing Connections Using GnuTLS

```
advice GAFlow(call("% connect(...)") || call("% send(...)") || call("%
    recv(...)")): before(){
  hardening_socketInfoStorageInit();
  hardening_initGnuTLSSubsystem(NONE);
}

advice GDFlow(call("% connect(...)") || call("% send(...)") || call("%
    recv(...)") || call("% close(...)")): after(){
  hardening_deinitGnuTLSSubsystem();
  hardening_socketInfoStorageDeinit();
}
```

Listing 5.24: Excerpt of Improved Hardening Aspect for Securing Connections Using GnuTLS

Principle of Least Privilege

For processes implementing the principle of least privilege, it is necessary to increase the active rights before the execution of a sensitive operation, and to relinquish such rights directly after its completion. Our pointcuts can be used to deal with a group of operations requiring the same privilege by injecting the privilege adjustment code at the *GAFlow* and *GDFlow* join points. This is applicable only in the case where no unprivileged operations are in the execution path between the initialization and the deinitialization points. The example in Listing 5.25 (made using combined code examples from [50]) shows an aspect implementing a lowering of privilege around certain operations. It uses restrict tokens and the SAFER API available in Windows XP. This solution injects code before and after each of the corresponding operations, incurring overhead, particularly in the case where the operations a, b and c would be executed consecutively. This could be avoided by using *GAFlow* and *GDFlow*, as we show in Listing 5.26.

Atomicity

In the case where a critical section may span across multiple program elements (such as function calls), there is a need to enforce mutual exclusion using tools such as semaphores around the critical section. The beginning and end of the critical section can be targeted using the *GAFlow* and *GDFlow* join points.

Listing 5.27, although correct-looking, can create unwanted side effects if two calls (say, `a` and `b`) were intended to be part of the same critical section (i.e., in the same execution path), as the lock would be released after `a`, and acquired again before `b`, allowing

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice abc: around(){
  SAFER_LEVEL_HANDLE hAuthzLevel;
  // Create a normal user level.
  if(SaferCreateLevel(SAFER_SCOPEID_USER, SAFER_LEVELID_CONSTRAINED, 0,
      &hAuthzLevel, NULL)){
    //  Generate the restricted token that we will use.
    HANDLE hToken = NULL;
    if(SaferComputeTokenFromLevel(hAuthzLevel, NULL, &hToken,0,NULL)){
      //sets the restrict token for the current thread
      HANDLE hThread = GetCurrentThread();
      if(SetThreadToken(&hThread,hToken)){
        tjp->proceed();
        SetThreadToken(&hThread,NULL); //removes restrict token
      }
      else{//error handling}
    }
    SaferCloseLevel(hAuthzLevel);
  }
}
```

Listing 5.25: Hypothetical Aspect Implementing Least Privilege

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice GAFlow(abc): before(){
  SAFER_LEVEL_HANDLE hAuthzLevel;
  // Create a normal user level.
  if(SaferCreateLevel(SAFER_SCOPEID_USER, SAFER_LEVELID_CONSTRAINED, 0,
      &hAuthzLevel, NULL)){
    //  Generate the restricted token that we will use.
    HANDLE hToken = NULL;
    if(SaferComputeTokenFromLevel(hAuthzLevel, NULL, &hToken,0,NULL)){
      //sets the restrict token for the current thread
      HANDLE hThread = GetCurrentThread();
      SetThreadToken(&hThread,NULL);
    }
    SaferCloseLevel(hAuthzLevel);
  }
}

advice GDFlow(abc): after(){
  HANDLE hThread = GetCurrentThread();
  SetThreadToken(&hThread,NULL); //removes restrict token
}
```

Listing 5.26: Improved Aspect Implementing Least Privilege

```
static Semaphore sem = new Semaphore(1);

pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice abc: before(){
  try{
    sem.acquire();
    } catch(InterruptedException e) {//...}
}

advice abc: after(){
  sem.release();
}
```

Listing 5.27: Aspect Adding Atomicity

the execution of another unwanted critical section, possibly damaging b internal state. Improving this aspect in order to handle this case requires foreknowledge of the program event flow, contradicting the core principle of separation of concerns and thus complicating further maintenance activities and preventing aspect reuse. In contrast, by using our proposal, the lock is acquired and released independently of the individual join points while guaranteeing that they will be, altogether, considered as one critical section. Listing 5.28 shows this improvement.

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice GAFlow(abc): before(){
  static Semaphore sem = new Semaphore(1);
  try{
    sem.acquire();
    } catch(InterruptedException e) {//...}
}

advice GDFlow(abc): after(){
  sem.release();
}
```

Listing 5.28: Improved Aspect Adding Atomicity

Logging

It is possible that a set of operations are of interest for logging purposes, but adding individual log entry for each one of them would be redundant or of little use. This is why it is desirable to use *GAFlow* and/or *GDFlow* in order to insert log statements before and/or after a set of interesting transactions.

5.5.2 General Advantages of GAFlow and GDFlow

It is clear that the proposed pointcuts support the principle of separation of concerns by allowing to implement program modification on a set of join points based on a specific concern. We now present some general advantages of the proposed pointcuts:

- ***Ease of use***: Programmers can target places in the application control flow graph where to inject code before or after a set of join points without needing to manually determine the precise point where to do so.

- ***Ease of Maintenance***: Programmers can change the program structure without needing to rewrite the associated aspects that were relying on explicit knowledge of the structure in order to pinpoint where the advice code would be injected. For example, if we need to change the execution path to a particular function (e.g., when performing refactoring), we also need to find manually the new common ancestor and/or descendant, whereas this would be done automatically using the proposed pointcuts.

- ***Execution Time and Memory Consumption***: Programmers can inject certain pre-operations and post-operations where needed in the program, without having to resort to injection in the catch-all `main`. This can improve the apparent responsiveness of the application since certain lengthy operations (such as library initialization) can be avoided if the branches of code requiring them are not executed, thus saving CPU cycles and memory usage. Also, this avoids the execution of the pre-operations and post-operations needed around each targeted join point, which is the default solution using the actual AOP techniques. This is replaced by executing them only once around the *GAFlow* and *GDFlow*.

- ***Raising the Abstraction Level***: Programmers can develop more abstract and reusable aspect libraries.

5.5.3 Usefulness of ExportParameter and ImportParameter for Security Hardening

This section illustrates the necessity and usefulness of *ExportParameter* and *ImportParameter* for some security hardening activities. This is done by (1) presenting an example that secures a connection using the current AOP technologies, (2) exploring the need for parameter passing and (3) presenting the solution of this example using our proposition.

Securing Connection using the Current AOP Technologies

Securing channels between two communicating parties is the main security solution applied to avoid eavesdropping, tampering with the transmission and/or session hijacking. The Transport Layer Security (TLS) protocol is widely used for this task. We thus present in this section a part of a case study, in which we implemented an AspectC++ aspect that secures a connection using TLS and weaved it with client applications to secure their connections. To generalize our solution and make it applicable on wide 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 and receiving data on the secure channels are replaced by the ones provided by TLS. On the other hand, the other functions that operate on the non-secure channels are kept untouched. Moreover, we addressed also the cases where the connection processes and the functions that send and receive the data are implemented in different components (i.e different classes, functions, etc.). In Listing 5.29, we see an excerpt of AspectC++ code allowing to harden a connection.

In Listing 5.29, the reader will notice the appearance of `hardening_sockinfo_t` as well as some other related functions, which are underlined for the sake of convenience. These are the data structure and functions that we developed to distinguish between secure and insecure channels and export the parameter between the application components at runtime. 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 sharing memory directly, we opted for a hash table that uses the Berkeley 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()` are replaced at runtime by the secure sending functions if the the socket is protected. This effort of sharing the parameter has both development and runtime overhead that could be avoided by the use of a primitive automating

```
aspect SecureConnection {
advice execution ("% main(...)") : around () {
  hardening_socketInfoStorageInit();
  hardening_initGnuTLSSubsystem(NONE);
  tjp->proceed();
  hardening_deinitGnuTLSSubsystem();
  hardening_socketInfoStorageDeinit();
}

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);
  gnutls_set_default_priority (socketInfo.session);
  gnutls_certificate_type_set_priority (socketInfo.session,
     cert_type_priority);
  gnutls_certificate_allocate_credentials (&socketInfo.xcred);
  gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
     socketInfo.xcred);
  //Connect
  tjp->proceed();
  if(*tjp->result()<0) {perror("cannot connect ");
   exit(1);}
  //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)
    //if the channel is secured, replace the send by gnutls_send
    *(tjp->result()) = gnutls_record_send(socketInfo.session, *(char**)
        tjp->arg(1), *(int *)tjp->arg(2));
  else
    tjp->proceed();
}
};
```

Listing 5.29: Excerpt of an AspectC++ Aspect Hardening Connections Using GnuTLS

the transfer of concern-specific data within advices without increasing software complexity. Furthermore, other experiments with another security feature (encrypting sensitive memory) showed that the use of hash table could not be generalized.

Need to Features for Passing Parameters

Our study of the literature and our previous experiments presented in Chapter 4 showed that it is often necessary to pass state information from one part to another of the program in order to perform security hardening. For instance, in the example provided in Listing 5.29, we need to pass the `gnutls_session_t` data structure from the advice around `connect` to the advice around `send` in order to properly harden the connection. The current AOP models do not allow to perform such operations. To address this limitation, we integrated additional modules and data structures and changed some functions within the application in order to pass the parameters. In the case of large scale applications with multiple features and complex dependencies and relations between their components, such solution is not realistic. It requires many complex re-engineering actions to be performed since any changes in one component lead to apply several modifications in all its dependent ones.

Securing Connection using ExportParameter and ImportParameter

We modified the example of Listing 5.29 by using the proposed approach for parameter passing. Listing 5.30 presents excerpt of the new code. All the data structure and algorithms (underlined in Listing 5.29) are removed. An *ExportParameter* for the parameters *session* and *xcred* is added on the declaration of the advice of the pointcut that identifies the function *connect*. On the other side, an *ImportParameter* for the parameter *session* is added on the declaration of the advice of the pointcut that identifies the function *send*.

5.6 Methodology, Algorithms and Implementation

This section presents the elaborated methodologies and algorithms for dominator set, graph labeling, *GAFlow*, *GDFlow*, *ExportParameter* and *ImportParameter*. Algorithms that operate on CFG have been developed for decades now, and many graph operations are considered to be common knowledge in computer science. Despite this theoretical richness, we are not aware of existing methods allowing to determine the *GAFlow* or *GDFlow* node for a particular set of nodes (i.e., join points) in a CFG by considering all the possible paths. On the other hand, the algorithms used to calculate the Dominator and Post-Dominator sets of a CFG node can be extended to consider such criteria and build the algorithms of *GAFlow* and *GDFlow*.

In this context, we propose two different sets of algorithms for *GAFlow* and *GDFlow*. The first set is based on the Dominator and Post-Dominator algorithms of classical CFG, while the second one operates on labeled graph (i.e., a label is associated to each node). Choosing between these algorithms is considered only during the implementation phase and left for the developers. We assume that the CFG is shaped in the traditional form, with a single start node and a single end node. In the case of program with multiple starting points, we consider each starting point as a different program in our analysis. Most of these assumptions have been used so far [41]. With these statements in place, we ensure that our algorithms will return a result (in the worst case, the start node or the end node) and that

```
aspect SecureConnection {

advice execution ("% main(...)") : around () {
  gnutls_global_init ();
  tjp->proceed();
  gnutls_global_deinit();
}

advice call("% connect(...)") : around () : ExportParameter(
   gnutls_session session, gnutls_certificate_credentials xcred){
  //variables declared
  static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
     GNUTLS_CRT_OPENPGP, 0};
  //initialize TLS session info
  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);
  //Connect
  tjp->proceed();
  if(*tjp->result()<0) {perror("cannot connect "); exit(1);}
  //TLS handshake
  gnutls_transport_set_ptr (session, (gnutls_transport_ptr) (*(int *)tjp
     ->arg(0)));
  *tjp->result() = gnutls_handshake (session);
}

//replacing send() by gnutls_record_send() on a secured socket
advice call("% send(...)") : around () : ImportParameter(gnutls_session
   session){
  //Check if the channel, on which the send function operates, is
     secured or not
  if(session != NULL)
    //if the channel is secured, replace the send by gnutls_send
    *(tjp->result())  = gnutls_record_send(*session, *(char**) tjp->arg
       (1), *(int *)tjp->arg(2));
  else
    tjp->proceed();
}
};
```

Listing 5.30: Hardening of Connections using GnuTLS and Parameter Passing

this result will be a single and unique node for all the inputs.

5.6.1 GAFlow and GDFlow using Dominator and PostDominator

The problem of finding the dominators in a control-flow graph has a long history in the literature and many algorithms have been proposed, improved and implemented [28, 49]. To compute dominance information, as presented in [28], the compiler can annotate each node in the CFG with a *DOM* and *PDOM* sets.

DOM(b) A node *n* in the CFG dominates *b* if *n* lies on every path from the entry node of the CFG to *b*. The set *DOM(b)* contains every node *n* that dominates *b*, including *b*. The dominators of a node *n* are given by the maximal solution to the following data-flow equations:

$$Dom(entry) = \{entry\} \tag{1}$$

$$Dom(n) = \left(\bigcap_{p \in preds(n)} Dom(p) \right) \bigcup \{n\} \tag{2}$$

Where *entry* is the start node and *preds(n)* is the set of all the predecessors of *n*. The dominator of the start node is the start node itself. The set of dominators for any other node *n* is the intersection of the set of dominators for all predecessors *p* of *n*. The node *n* is also in the set of dominators for *n*. To solve these equations, the iterative algorithm presented in [28] can be used.

We elaborated and implemented Algorithm 1 to calculate the Dominator set. It is based on the mechanisms identifying the possible paths reaching one destination from one source node [34]. However, any other available algorithm that gives the same result can be useful. This choice is completely left for the developer who is expert in such domain.

The proposed algorithm for finding the non-trivial dominator nodes of a node n, starting from an entry point node is based on finding all the connecting (execution) paths between node n and the $entryPoint$ node and then keeping only the common nodes of these paths. The algorithm used for finding the connecting paths is using a marking map overlay that is created recursively on the graph nodes starting from node n and finishing at the $entryPoint$ node or the root node. More precisely, at each marked node we have a map containing key and value pairs, with the keys corresponding to the previously connecting nodes and increasing marking values (except for node n which has a mark entry with itself as key and 0 as the initial marking value) with respect to the corresponding connecting nodes.

Once the marking is completed, we can trace the paths by exploring the markings in a recursive procedure that is tracking (adding the current node in the list upon entry and popping it before exiting) with each recursion the explored nodes in a list of ascendants constituting the currently explored path. The latter is added to the list of paths whenever the currently explored node is in fact the target ($entryPoint$) node. In essence, at every explored node, the markings of the parent node are iterated and compared against the markings of the current node in order to find an adjacent sequence of increasing values denoting an unvisited branch. Whenever one is found, the corresponding marking is removed from the marking map of the current node and the path tracing function is called recursively for the parent node. Upon return, the removed marking is restored in order to allow for the

Algorithm 1 Algorithm to Determine the Dominator Set

```
Set DOM(Node $entryPointNode$, Node $n$)
mark($entryPointNode$, $n$);
Stack $pathList$ = new Stack();
tracePath($entryPointNode$, $n$, new Stack(), $pathList$);
Set meetSet = {};
if !$pathList$.isEmpty() then
    $meetSet$.addAll($pathList$.pop());
    for each path $pth$ in $pathList$ do
        $meetSet = meetSet \bigcap$ (Set)$pth$
    end for
end if
return $meetSet$;

markNode(Node $targetNode$, Node $currentNode$, Node $branchingNode$, int $markIndex$)
if $\nexists$ $currentNode.pathMarkingMap$.get($branchingNode$) then
    $currentNode.pathMarkingMap$.put($branchingNode$, $markIndex$);
    if $currentNode$ != $targetNode$ then
        $markIndex = markIndex + 1$;
        for each parent $p$ in $currentNode.parentList$ do
            markNode($targetNode$, $p$, $currentNode$, $markIndex$);
        end for
    end if
end if
tracePath(Node $targetNode$, Node $currentNode$, Stack $ascendList$, Stack $pathList$)
$ascendList$.push($currentNode$);
if $currentNode == targetNode$ then
    List $path$ = new List();
    $path$.addAll($ascendList$);
    $pathList$.add($path$);
else
    for each parent $p$ in $currentNode.parentList$ do
        if $\exists$ $p.pathMarkingMap$.get($currentNode$) then
            $pathMarkValue$=$p.pathMarkingMap$.get($currentNode$);
            for each $markingKey$ in $currentNode.pathMaringMap$.keySet() do
                int $markingValue$ = $currentNode.pathMarkingMap$.get($markingKey$);
                if $markingValue + 1 == pathMarkValue$ then
                    $currentNode.pathMarkingMap$.remove($markingKey$);
                    tracePath($targetNode$, $p$, $ascendList$, $pathList$);
                    $currentNode.pathMarkingMap$.put($markingKey$, $markingValue$);
                    break;
                end if
            end for
        end if
    end for
end if
$ascendList.pop()$;
```

discovery of other paths passing through the same node.

PDOM(b) A node *n* in the CFG post-dominates *b* if *n* lies on every path from *b* to the exit node of the CFG. The set *PDOM(b)* contains every node *n* that post-dominates *b*, including *b*.

A simple method to calculate the post-dominator sets is to reverse the edge direction of the CFG, start from the exit node and apply the dominator algorithm [49]. The post-dominator of the exit node is the exit node itself. In the case of multiple end points, we consider each ending point as different program in our analysis (in fact, each ending point will be a starting point after applying the CFG reverse edge direction mechanism).

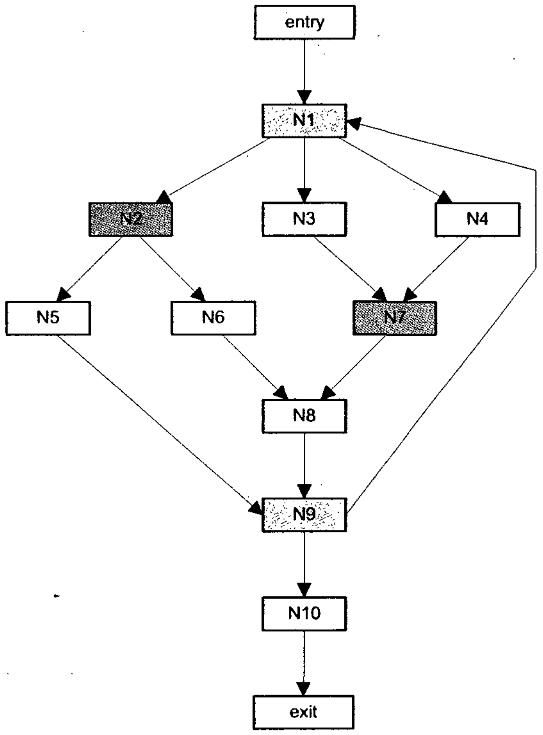


Figure 10: Graph Illustrating the *GAFlow* and *GDFlow* of N2 and N7

Selected Nodes	**Common Dominator Set**	***GAFlow***
N2, N7	entry, N1	N1
N5, N6	entry, N1, N2	N2
N4, N6, N10	entry, N1	N1
N8, N9	entry, N1	N1

Table 2: Results of the Execution of Algorithm 2 on the Graph of Figure 10 (a)

Selected Nodes	N4, N6, N10
DOM(N4)	entry, N1
DOM(N6)	entry, N1, N2
DOM(10)	entry, N1, N9
CommonDominatorSet (N4, N6, N10)	entry, N1
GAFLow	N1

Table 3: Results of the Execution of Algorithm 2 on the Graph of Figure 10 (b)

Pointcut GAFLow

In order to compute the *GAFlow*, we developed a mechanism built on top of the dominator algorithm. First, we calculate the common dominator set of all the selected nodes specified in the parameter of *GAFlow*. Then we remove the selected nodes from the calculated set. The last node in this set will be returned by Algorithm 2 as the closest guaranteed ancestor.

Algorithm 2 Algorithm to determine *GAFlow* using dominator

```
Require: $SelectedNodes$ is initialized with the contents of the pointcut match
1: GAFlow(NodeSet SelectedNodes):
2: $CommonDomSet \leftarrow \emptyset$
3: for all $node \in SelectedNodes$ do
4:     $CommonDomSet \leftarrow CommonDomSet \cup (DOM(node) - node)$
5: end for
6: return $GetLastNode(CommonDomSet)$
```

We implemented Algorithm 1 to calculate the dominator set of a particular node. Then, we implemented on top of it Algorithm 2 and applied this implementation into several case studies, one of them is illustrated in Figure 10. The result and steps of calculating the *GAFlow* of some selected nodes is illustrated in Tables 2 and 3.

Selected Nodes	**Common Post-Dominator Set**	***GDFlow***
N2, N7	N9, N10, exit	N9
N4, N5, N6	N9, N10, exit	N9
N6, N7	N8, N9, N10, exit	N8
N8, N9	N10, exit	N10

Table 4: Results of the Execution of Algorithm 3 on the Graph of Figure 10 (a)

Pointcut GDFLow

The closest guaranteed descendant is determined by elaborating a mechanism built on top of the post-dominator algorithms. First, we calculate the common post-dominator set of all the selected nodes specified in the parameter of *GDFlow*. Then we remove the selected nodes from the calculated set. The first node in this set will be returned by the Algorithm 3 as the closest guaranteed descendant.

Algorithm 3 Algorithm to determine *GDFlow* using post-dominator

```
Require: $SelectedNodes$ is initialized with the contents of the pointcut match
1: GDFlow(NodeSet SelectedNodes):
2: $CommonPostDomSet \leftarrow \emptyset$
3: for all $node \in SelectedNodes$ do
4:     $CommonPostDomSet \leftarrow CommonPostDomSet \cup (PDOM(node) - node)$
5: end for
6: return $GetFirstNode(CommonPostDomSet)$
```

Similarly to Algorithm 2, we implemented Algorithm 3 by reversing the edge direction of the CFG, starting from the exit node and applying Algorithm 1 to calculate the post-dominator set of a particular node [49]. Then, we applied this implementation on several case studies, one of them illustrated in Figure 10. The result and steps of calculating the *GDFlow* of some selected nodes is illustrated in Tables 4 and 5.

SelectedNodes	N4, N5, N6
PDOM(N4)	N7, N8, N9, N10, exit
PDOM(N5)	N9, N10, exit
PDOM(N6)	N8, N9, N10, exit
CommonPostDominatorSet (N4, N5, N6)	N9, N10, exit
GDFLow	N9

Table 5: Results of the Execution of Algorithm 3 on the Graph of Figure 10 (b)

5.6.2 GAFlow and GDFlow using Labeled Graph

As an alternate solution to determine the *GAFlow* and *GDFlow,* we also chose to use a graph labeling algorithm developed by our colleagues that we slightly modified in order to meet our requirements. This algorithm allows to associate a label to each node of a graph as depicted in Figure 11. Algorithm 4 describes the graph labeling method.

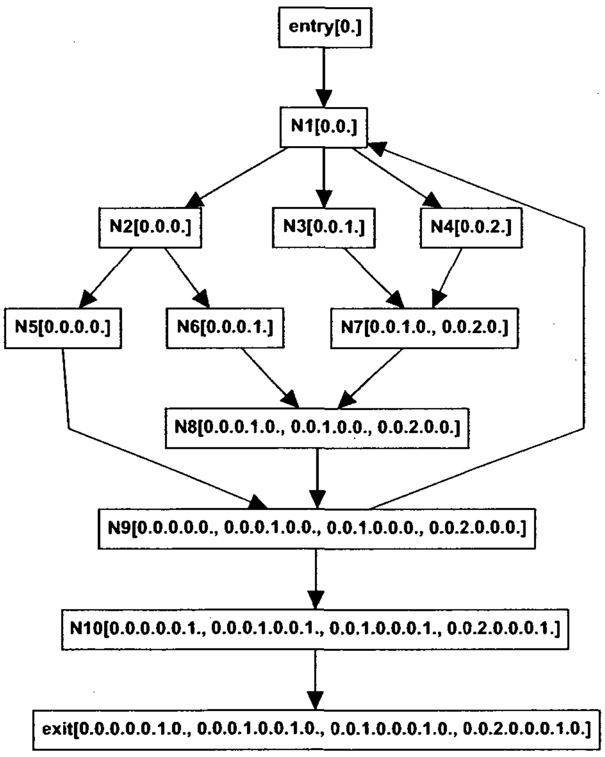

entry[0.]
N1[0.0.]
N2[0.0.0.]
N3[0.0.1.]
N4[0.0.2.]
N5[0.0.0.0.]
N6[0.0.0.1.]
N7[0.0.1.0., 0.0.2.0.]
N8[0.0.0.1.0., 0.0.1.0.0., 0.0.2.0.0.]
N9[0.0.0.0.0., 0.0.0.1.0.0., 0.0.1.0.0.0., 0.0.2.0.0.0.]
N10[0.0.0.0.0.1., 0.0.0.1.0.0.1., 0.0.1.0.0.0.1., 0.0.2.0.0.0.1.]
exit[0.0.0.0.0.1.0., 0.0.0.1.0.0.1.0., 0.0.1.0.0.0.1.0., 0.0.2.0.0.0.1.0.]

Figure 11: Sample Labeled Graph

Each node down the hierarchy is labeled in the same manner as the table of contents

Algorithm 4 Hierarchical Graph Labeling Algorithm

```
labelNode(Node $s$, Label $l$):
$s.labels \leftarrow s.labels \cup \{l\}$
$NodeSequence\ children = s.children()$
for $k = 0$ to $|children| - 1$ do
    $child \leftarrow children[k]$
    if $\neg hasProperPrefix(child, s.labels)$ then
        $labelNode(child, l +_c k +_c "."$);
    end if
end for

hasProperPrefix(Node $s$, LabelSet $parentLabels$):
if $s.label = \epsilon$ then
    return false
end if
if $\exists s \in Prefixes(s.label) : s \in parentLabels$ then
    return true
else
    return false
end if

Prefixes(Label $l$):
$LabelSet\ labels \leftarrow \emptyset$
$Label\ current \leftarrow$""
for $i \leftarrow 0$ to $l.length()$ do
    $current.append(l.charAt(i))$
    if $Label1.charAt(i) =$ '.' then
        $labels.add(current.clone())$
    end if
end for
```

Selected Nodes	***GAFlow***
N2, N7	N1
N5, N6	N2
N4, N6, N10	N1
N8, N9	N1

Table 6: Results of the Execution of Algorithm 5 on the Labeled Graph of Figure 11

of a book (e.g., 1., 1.1., 1.2., 1.2.1., ...), as depicted by Algorithm 4, where the operator $+_c$ denotes string concatenation (with implicit operand type conversion). To that effect, the labeling is done by executing Algorithm 4 on the *start* node with label "0.", thus recursively labeling all nodes.

We implemented Algorithm 4 and tested it on a hypothetical CFG. The result is displayed in Figure 11. This example will be used throughout the rest of this chapter.

Pointcut GAFlow

In order to compute the *GAFlow*, we developed a mechanism that operates on the labeled graph. We compare all the hierarchical labels of the selected nodes in the input set and find the largest common prefix they share. The node labeled with this largest common prefix is the closest guaranteed ancestor. We insure that the *GAFlow* result is a node through which all the paths that reach the selected nodes pass by considering all the labels of each node. This is elaborated in Algorithm 5. Please note that the `FindCommonPrefix` function was specified recursively for the sake of simplicity and understanding.

We implemented Algorithm 5 and we applied it on the labeled graph of Figure 11. We selected, as case study, some nodes in the graph with various combinations. Few results are summarized in Table 6 and Figure 12.

Algorithm 5 Algorithm to determine *GAFlow* using labeled graph

```
Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled
 1: GAFlow(NodeSet SelectedNodes):
 2: $LabelSequence\ Labels \leftarrow \emptyset$
 3: for all $node \in SelectedNodes$ do
 4:    $Labels \leftarrow Labels \cup node.labels()$
 5: end for
 6: return $GetNodeByLabel(FindCommonPrefix(Labels))$
 7:
 8: FindCommonPrefix (LabelSequence Labels):
 9: if $|Labels| = 0$ then
10:    return error
11: else if $|Labels| = 1$ then
12:    return $Labels.removeHead()$
13: else
14:    $Label\ Label1 \leftarrow Labels.removeHead()$
15:    $Label\ Label2 \leftarrow Labels.removeHead()$
16:    if $|Labels| = 2$ then
17:       for $i \leftarrow 0$ to $min(Label.length(), Label2.length()$ do
18:          if $Label1.charAt(i) \neq Label2.charAt(i)$ then
19:             return $Label1.substring(0, i - 1)$
20:          end if
21:       end for
22:       return $Label1.substring(0, min(Label.length(), Label2.length())$
23:    else
24:       $Label\ PartialSolution \leftarrow FindCommonPrefix(Label1, Label2)$
25:       $Labels.append(PartialSolution)$
26:       return $FindCommonPrefix(Labels)$
27:    end if
28: end if
```

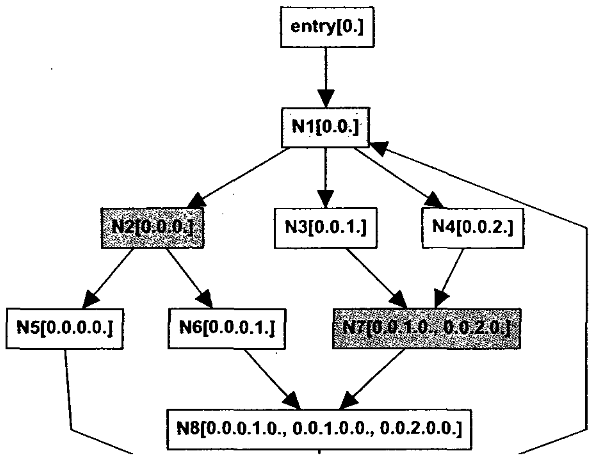


Figure 12: Excerpt of Labeled Graph Illustrating the *GAFlow* of N2 and N7

Pointcut GDFlow

The same mechanism for reversing the edge direction of the CFG [49], that calculates the post-dominator set by using the dominator algorithm, can also be applied to determine the closest guaranteed descendant on a labeled graph (see Section 5.6.1 for more detail). Once the edge directions are reversed, labeling the CFG can be performed and then the same algorithm used for calculating the *GAFLow* (Algorithm 5) can be applied to determine the *GDFLow*.

We used the same implementation of Algorithm 4 and case study illustrated in Figure 11. Then, we applied the aforementioned mechanism and implemented Algorithm 5 to calculate the *GDFlow* for the selected nodes. Table 7 contains few results along with Figure 13.

Selected Nodes	*GDFlow*
N2, N7	N9
N4, N5, N6	N9
N6, N7	N8
N8, N9	N10

Table 7: Results of the Execution of Reverse Edge Direction and Algorithm 5 on the Labeled Graph of Figure 11

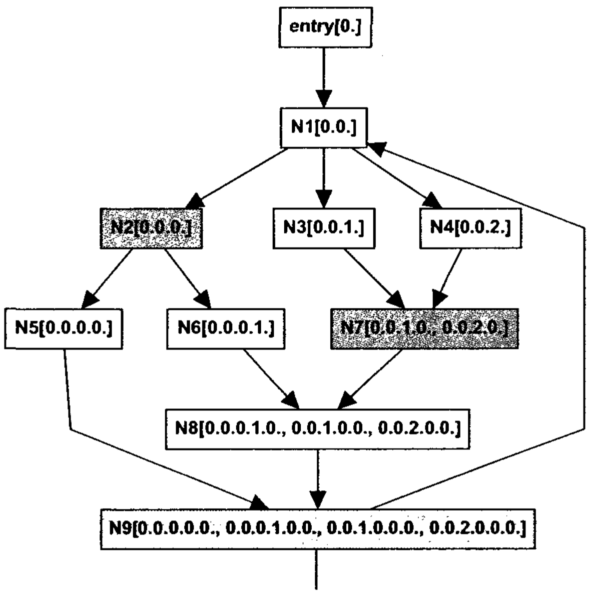

entry[0.]
N1[0.0.]
N2[0.0.0.]
N3[0.0.1.]
N4[0.0.2.]
N5[0.0.0.0.]
N6[0.0.0.1.]
N7[0.0.1.0., 0.0.2.0.]
N8[0.0.0.1.0., 0.0.1.0.0., 0.0.2.0.0.]
N9[0.0.0.0.0., 0.0.0.1.0.0., 0.0.1.0.0.0., 0.0.2.0.0.0.]

Figure 13: Excerpt of Labeled Graph Illustrating the *GDFlow* of N2 and N7

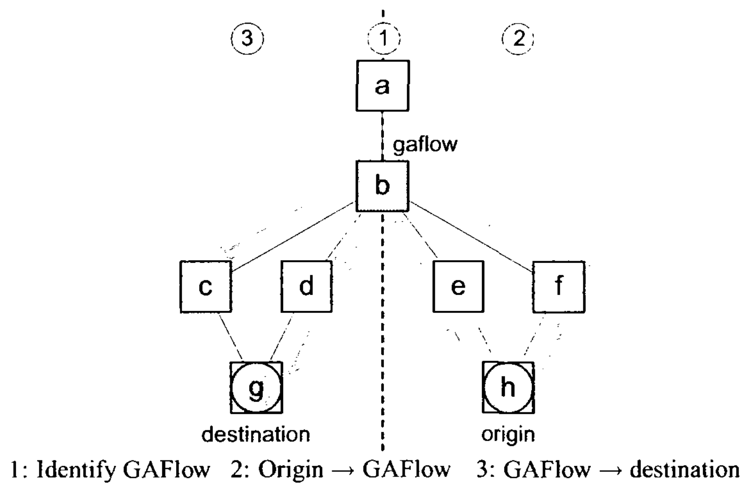
5.6.3 Primitives ExportParameter and ImportParameter

This section presents the implementation methodology and algorithms of the proposed primitives responsible of passing parameters, together with the experimental results. These primitives are the *ExportParameter* and *ImportParameter.* The *ExportParameter* is used in the advice of the origin pointcut to make the parameters available, while the *ImportParameter* is used in the advice of the destination pointcut to import the needed parameters.

Algorithm 7 allows parameter passing between two nodes of the context-insensitive call graph of a program [44], with each node representing a function and each arrow representing a call site. To ensure the declaration and initialization of the passed parameter all the time, whatever the selected execution path, we elaborated on top of this algorithm a mechanism based on the *GAFlow*. This mechanism exports the parameter from the origin to the destination nodes.

This is achieved by performing the following steps: (1) Calculating, using the CFG, the closest guaranteed ancestor (*GAFlow*) of the origin (*ExportParameter*) and destination join points (*ImportParameter*), (2) localizing the three nodes representing the origin, destination and *GAFlow* in the call graph, (3) declaring and initializing the parameter in the node representing the *GAFlow* in the call graph, (4) executing Algorithm 7 to pass the parameter from the origin node to the *GAFlow* node, and (5) executing again the same algorithm to pass the parameter from the *GAFlow* node to the destination node. This procedure is described in Algorithm 6 and operates on one parameter at a time.

The *GAFlow* of a set of points is always called before the points themselves (*GAFlow* criteria). By passing the parameter from the origin to *GAFlow* and then to the destination, we ensure that the parameter will be definitely declared and initialized, even if the destination is called before the origin. Otherwise, the parameter could be communicated without initialization, which would create software errors and affect the correctness of the solution. However, in all the security hardening cases we have treated, the origin is always called before the destination. For instance, in the case study of securing the connection of applications, the functions responsible for establishing the connections are always called before the functions responsible for exchanging data, otherwise there will be an execution error. This also apply on all the cases where a sequence of operations should be executed in order



3
1
2
a
gaflow
b
c
d
e
f
g
h
destination
origin
1: Identify GAFlow 2: Origin → GAFlow 3: GAFlow → destination

Figure 14: Parameter Passing in a Call Graph

to provide a particular functionality (indeed, this is the only cases where we need to pass parameters between two points in a program).

Figure 14 shows an illustration of Algorithm 6 on a call graph example. To pass the parameter from *h* to *g*, their *GAFlow*, which is *b* in this case, is first identified. Afterwards, the parameter is passed over all the call sites (paths) from *h* to *b*, then from *b* to *g* again over all the call sites.

Algorithm 6 Algorithm to Pass the Parameter between two pointcuts

```
1: function passParameter(Node origin, Node end, Parameter param):
2: if $origin = destination$ then
3:    return $success$
4: end if
5: $start \leftarrow GuaranteedAncestor(origin, end)$
6: $passParamOnBranch(start, origin, param)$
7: $node.addLocalVariable(param)$
8: $passParamOnBranch(start, end, param)$
```

The proposed methodology presented in Algorithm 7 allows to modify the function signatures and calls in a way that would preserve the program syntactical correctness and intent (i.e., would still compile and behave the same). It finds all the call sites (paths) between the origin node and the destination node in the call graph. For each one, it propagates the parameter from the called function to the caller, starting from the end of the path. In other words, the signatures of all the functions involved in the call graph between the exporting and importing join points are augmented by a parameter `inout`. All calls to these functions are modified to pass the parameter as is, in the case of the functions involved in this transmission path (e.g., nodes *b*, *c*,*d*,*e* and *f* of Figure 14). In order to be optimal in the presence of loops, it modifies all the callers only one time and keeps track of the modified nodes.

We implemented a program similar to the scenario of the call graph illustrated in Figure 14. This program, which is presented in Listing 5.31, is essentially a client application that establishes a connection, sends a request and receives a response from the server. Then, we simulated the execution of the proposed primitive algorithms and applied manually the aspects presented in Listings 5.30 on this application in order to secure its communication channels, producing the programme in Listing 5.32. We successfully tested the correctness of the hardened applications with SSL enabled web server by capturing the exchange of data packets, demonstrating that the communication was effectively encrypted.

5.7 Conclusion

AOP is a very promising paradigm for software security hardening. However, this technology was not initially designed to address security issues and many research initiatives

```
const char * HTTPrequest = "GET / HTTP/1.1 \nHost: localhost\n\n";

int dosend(int sd, char * buffer, unsigned int bufSize){
  return send(sd, buffer, bufSize, 0);
}

int doreceive(int sd, char * buffer, unsigned int bufSize){
    return recv(sd, buffer, bufSize, 0);
}

int doConnect(int sd, struct sockaddr_in servAddr){
    return connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
}

int main (int argc, char *argv[]) {
    /* ...   */
      /* create socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* connect to server */
    rc= doConnect(sd, servAddr);

    /*send/receive*/
    rc = dosend(sd);
    fprintf(stderr,"Sent %u characters:\n%s\n", rc, HTTPrequest);
    memset((void *)buf, 0, MAX_MSG);
    rc=doreceive(sd, buf, MAX_MSG);
    fprintf(stderr,"Received %u characters:\n%s", rc, buf);

    /* Shutdown */
    close(sd);

    /* ...   */
}
```

Listing 5.31: Excerpt of a Program to be Hardened

```
const char * HTTPrequest = "GET / HTTP/1.1 \nHost: localhost\n\n";
int dosend(int sd, char * buffer, unsigned int bufSize, gnutls_session_t
    * session){
  if(session != NULL) return gnutls_record_send(*session, buffer,
     bufSize);
  else return send(sd, buffer, bufSize, 0);
}
int doreceive(int sd, char * buffer, unsigned int bufSize,
   gnutls_session_t * session){
  if(session != NULL) return gnutls_record_recv(*session, buffer,
     bufSize);
  else return recv(sd, buffer, bufSize, 0);
}
int doConnect(int sd, struct sockaddr_in servAddr, gnutls_session_t *
   session, gnutls_certificate_credentials_t * xcred){
  static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
     GNUTLS_CRT_OPENPGP, 0};
  int rc;
  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);
  rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
  if(rc >= 0){
    gnutls_transport_set_ptr (*session, (gnutls_transport_ptr) sd);
    rc = gnutls_handshake (*session);
  }
  return rc;
}
int main (int argc, char *argv[]) {
  gnutls_global_init ();
  /* ... */
  /* create socket */
  sd = socket(AF_INET, SOCK_STREAM, 0);
  if(sd<0) {
    perror("cannot open socket");
    exit(1);  }
  doConnect(sd, servAddr,&session,&xcred);
  dosend(sd,HTTPrequest,strlen(HTTPrequest) + 1,&session);
  memset((void *)buf, 0, MAX_MSG);
  doreceive(sd, buf, MAX_MSG,&session);
  /* Shutdown */
  close(sd);
  gnutls_bye(session, GNUTLS_SHUT_RDWR);
  gnutls_deinit(session);
  gnutls_certificate_free_credentials(xcred);
  gnutls_global_deinit();
  return 0;
}
```

Listing 5.32: Resulting Hardened Program

Algorithm 7 Algorithm to Pass a Parameter Between Two Nodes of a Call Graph

```
function passParamOnBranch(Node origin, Node destination, Parameter param):
if $origin = destination$ then
    return $success$
end if
$paths \leftarrow findPathsBetween(origin, destination)$
for all $path \in paths$ do
    $path.remove(origin)$
    while $\neg path.isEmpty()$ do
        $currnode \leftarrow path.tail()$
        $path.remove(currnode)$
        if $\neg node.signature.isModified() \wedge$
        $\neg\exists parameter \in currnode.signature() : parameter = param$ then
            $node.signature.addParameter(param)$
            $node.signature.markModified()$
            $modifyFunctionsCallsTo$
            $(currNode, param)$
        end if
    end while
end for
return $success$

function modifyFunctionsCallsTo(Node currnode, Parameter param):
for all $caller \in currnode.getCallers()$ do
    for all $call \in caller.getCallsTo(node) : \neg call.modified$ do
        $call.parameters.add(param)$
        $call.modified =$ true
    end for
end for
```

showed its limitations in such domain. Similarly, we explored in this chapter the limitations of AOP in applying some security hardening practices. Consequently, this imposes restrictions to the proposed security hardening framework, from which the need to extend this technology with new pointcuts and primitives. In this context, we elaborated the following AOP pointcuts and primitives that enrich our proposed framework and AOP languages and provide features needed for systematic security hardening concerns: *GAFlow*, *GDFlow*, *ExportParameter* and *ImportParameter*. The *GAFlow* returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The *GDFlow* returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. The two primitives pass parameters from one advice to the other through the program call graph. We explored the viability of the proposed pointcuts and primitives by (1) exploring their advantages for security hardening, (2) developing their corresponding algorithms and (3) presenting the results of explanatory case studies.

Chapter 6

Formal Semantics of *SHL* Weaving

6.1 Introduction

The main intent of this chapter is to ascribe a formal semantics of the *SHL* language, and hence for the whole security hardening solutions performed by the proposed framework. The work performed to reach this objective results in two main contributions:

- Elaborating a novel approach for applying aspect-oriented weaving on the *Gimple* representation of software.
- Elaborating a formal semantics of *SHL* weaving based on the *Gimple* representation of software.

The initial security hardening approach discussed in Chapter 4 is based on the following components: Security Hardening Language (*SHL*), plans, patterns and their equivalent aspects. The combination of these components allows the developers to perform systematic security hardening of software by applying well-defined solutions and without the need to have security expertise. In this approach, the security hardening solutions need to be refined manually into the current AOP languages (e.g., AspectC++, AspectJ) before weaving the security components into the code.

In this chapter, we extend our proposition by elaborating a new approach that allows to apply the hardening on the *Gimple* representation (tree) of software and avoid in some cases the refinement of pattern to the current AOP technologies. *Gimple* is an intermediate representation of a program. It is language-independent and tree-based representation generated by GNU Compiler Collection (*GCC*) during the compilation. We propose in this chapter novel weaving capabilities for *Gimple* to be integrated into the *GCC* compiler. These features allow to compile the security hardening patterns and inject them into the *Gimple* tree of a program during the *GCC* compilation. Beside, exploiting Gimple intermediate representation enables to advise an application written in a specific language with code written in a different one.

Regarding the formal specification of *SHL*, we present in this chapter a core syntax for *Gimple*, a core syntax for *SHL* syntax, and formal semantics for *Gimple* weaving. This formal specification constitutes an initial attempt and a guide toward developing a complete weaver for *Gimple*. It also constitutes a base for applying formal verification on the performed security hardening solutions. We demonstrate the feasibility of our propositions by providing the methodology and results of implementing into *GCC* some weaving features illustrated in the proposed semantics. This is followed by a case study for securing the connections of client applications, where the hardening is applied on the *Gimple* representation and compiled using our extended *GCC*.

The remainder of this chapter is organized as follows. We provide in Section 6.2 a brief background on formal description and semantics and summarize in Section 6.3 the related work on AOP weaving semantics. Afterwards, in Section 6.4, we illustrate the new proposition for systematic security hardening where weaving is performed on the *Gimple* representation of a software by adopting an aspect-oriented style. Then, in Section 6.5, we present the syntax of *SHL* and *Gimple* and provide the operational semantics for *Gimple* weaving. After that, we explain briefly in Section 6.6 the methodology and results of implementing several *Gimple* weaving capabilities into the *GCC* compiler. Finally, we illustrate in Section 6.7 a security hardening case study and offer in Section 6.8 some concluding remarks.

6.2 Formal Semantics

Formal semantics constitutes of rigorous mathematical study of the meaning of languages and models of computation [65, 77]. It allows to prove the properties of a program. The formal semantics of a language is specified by a mathematical model that illustrates the possible computations described by the language. There are many approaches to formal semantics that belong to three major classes: Operational semantics, denotational semantics and axiomatic semantics. These three classes are presented in the increasing order of abstraction with respect to the concepts of meaning underlying them. The following is a brief description for each one of them:

- Operational semantics describes the execution of the language directly rather than by translation. It somehow corresponds to interpretation, where the implementation language of the interpreter is a mathematical formalism. The operational semantics may define an abstract machine and give meaning to the transitions between its states. It may also be defined via syntactic transformations on phrases of the language itself.

- Denotational semantics translates each phrase in the language to another phrase in another language. It somehow corresponds to compilation, where the target language is a mathematical formalism.

- Axiomatic semantics gives meaning to phrases by expressing the logical axioms that apply to them. Axiomatic semantics does not distinguish between a phrase meaning and the logical formulas describing it. A phrase means exactly what can be proven about it in some logic.

Since this chapter presents an operational semantics for *SHL* weaving, in the sequel we elaborate more about this approach and introduce the used structural operational semantics. Operational semantics is considered as a method to give meaning to programs in a mathematically rigorous way. It describes how a valid program is interpreted as sequences of computational steps, which then constitute the meaning of the whole program. The final step in the terminating sequence returns the value of the program in the case of a functional program. A program could be also nondeterministic, in this context there may be many computation sequences and many return values.

Structural operational semantics is an approach proposed to give logical means in defining operational semantics [67]. It consists of defining the behavior of a program in terms of the behavior of its parts. Hence, it provides a structural, a syntax oriented and an inductive view on operational semantics. Computation is represented by means of deductive systems that turn the abstract machine into a system of logical inferences. This allows to apply formal analysis on the behavior of programs. The proofs of program properties are derived directly from the definitions of the language constructs because the semantics descriptions are based on deductive logic.

With structural operational semantics, the behavior of a program is defined in terms of a set of transition relations. Such specifications take the form of inference rules. The valid transitions of a composite piece of syntax is defined into these rules in terms of the transitions of its components. Definitions are given by inference rules, which consist of a conclusion that follows from a set of premises, possibly under control of some conditions. An inference rule has a general form consisting of the premises listed above a horizontal line, the conclusion below, and the condition, if present, to the right, as follows [77]:

$$\frac{premise_1 \quad premise_2 \quad \dots \quad premise_n}{conclusion}$$

If n=0, i.e., the number of premises is zero, then the line containing the premises is omitted, and we refer to the rule as an axiom.

6.3 Related Work on AOP Weaving Semantics

The related work that addresses AOP weaving semantics is presented in this subsection. None of them has defined a semantics that demonstrates how to weave in *Gimple* trees.

The most prominent research proposals in this area are the contribution of Walker et al. [84] where the authors have defined the semantics of the aspect-oriented language MinAML, and the contribution of Dantas et. al. [30] where the authors have defined

PolyAML, a typed functional and aspect-oriented language. They have used labels to mark points where advices are going to be injected. Advices are applied to the arguments or to the result of a function.

Tatsuzawa et al. [59] have implemented an aspect-oriented version of core O'Caml called Aspectual Caml. Aspectual Caml carries out type inference on advices without consulting the types of the functions designated by the pointcuts. In addition, there are no formal definitions for Aspectual Caml.

Wand et al. [85] have presented a denotational semantics for pointcuts and advices of an AOP language defined in the Aspect Sand Box (ASB) project [36]. The language is untyped. The language of the pointcuts includes designators for procedure calls and control flows, but not for variable access or update.

Wang et.al. [86] have provided seamless integration of AOP paradigm and strongly-typed functional language paradigm through a static weaving process, which deals with around advices and type-scoped pointcuts in the presence of higher-order functions. However, their advice is scoped such that it is not possible to install advice that will affect already defined functions.

It is noticeable that all the previous contributions target AOP with functional programming. As a new idea, a name-based calculus μABC [23] has been introduced in which aspects are the primitive computational entities. The authors have demonstrated its expressiveness by presenting encodings of various other languages into μABC. In μABC, computational events are messages sent from a source to a target.

6.4 *Gimple* Weaving Approach

This section summarizes the approach for systematic security hardening and presents an extension to it based on *Gimple* weaving and needed to achieve our objectives. The whole approach architecture is illustrated in Fig. 15.

In the original approach presented in Chapter 4, once the security hardening solutions are built, the refinement of the solutions into aspects or low level code can be performed by programmers who 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. This task still requires human interaction to refine the patterns into aspects by providing some parameters needed for the implementation.

We first provide in this chapter an extension to this approach, which allows bypassing the refinement step from pattern into aspect, and consequently not using the current AOP weavers to harden the software. The hardening tasks specified into the patterns are abstract and programming language-independent, which makes the *Gimple* representation (i.e., *Gimple* Tree) of software a relevant target to apply the security hardening.

In this approach, the *SHL* patterns and the original software are passed to an extended version of the *GCC* compiler, which generates the executable of the trusted software. An additional pass has been added to *GCC* in order to interrupt the compilation once the *Gimple* representation of the code is completed. In parallel, the hardening pattern is compiled and a *Gimple* tree is built for each *Behavior* (Please see *SHL* syntax in Fig. 16) using the routines of *GCC* provided for this purpose. Afterwards, the generated security trees will be integrated in the tree of the original code with respect to the location(s) *location*

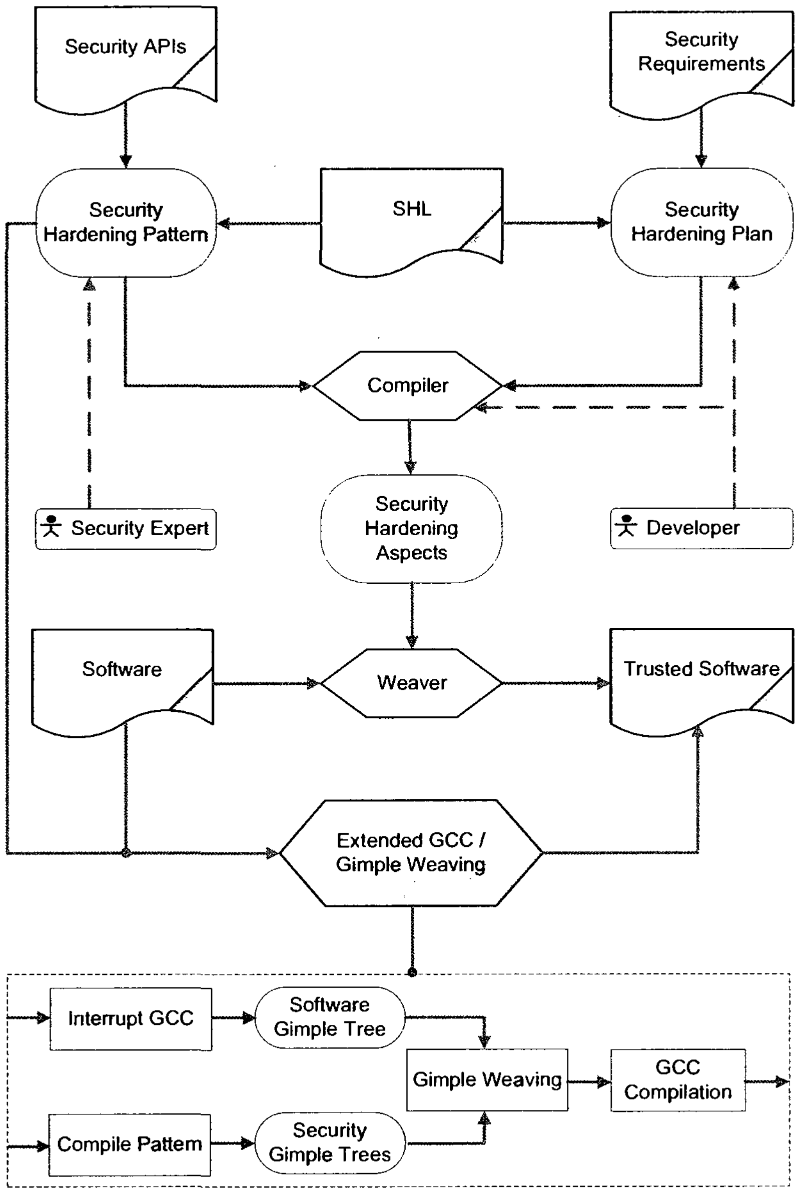


Figure 15: Approach Architecture

specified into each *Location_Behavior* of the pattern. Finally, the resulting *Gimple* tree is passed again to *GCC* in order to continue the regular compilation process and produce the executable of the secure software. The added features was originally implemented by our colleagues [89] in order to insert code for monitoring. We have modified it in order to inject the security functionalities specified in the hardening pattern.

Moreover, we have elaborated the formal specification of weaving an *SHL* pattern into the *Gimple* representation of a software. In this context, we provide in this chapter the syntax of *SHL* and *Gimple*, together with a formal operational semantics of the weaving capabilities. Providing such semantics allows to understand the inner working of *Gimple* procedures, and hence leads to complete implementation of the weaving capabilities for *Gimple*. Moreover, it may allow to formally verify the effect of applying the security hardening patterns and solutions into applications.

Beside the fact that the contributions presented in this chapter improve the approach for systematic security hardening, it also constitutes by itself the first attempt towards adopting aspect-oriented programming on *Gimple*, exploring it into a formal operational semantics and exploiting *Gimple* intermediate representation to weave an application written in a specific programming language with code written in a different one.

We have illustrated the feasibility of our propositions by developing several *Gimple* weaving features into *GCC* and elaborating a case study showing first the use of AOP (AspectC++) to secure the connections of an application implemented in C++ , then exploring the *Gimple* weaving of the extended *GCC* to integrate the same security code in the *Gimple* tree. The experimental results explore the relevance of applying both methods to harden security.

6.5 Formal Weaving Description

In this section, we present part of the syntax of *SHL* and *Gimple* that serves our goals. Beside, the weaving semantics is provided. This semantics describes how to inject security-related code at specific locations in the *Gimple* representation of programs. We first define the notations that are used along this section.

Notations

- Given a record space $D = \langle f_1 : D_1, f_2 : D_2, \ldots, f_n : D_n \rangle$ and an element e of type D, the access to the field f_i of an element e is written as $e.f_i$.
- Given a type τ, we write τ`-set` to denote the type of sets having elements of type τ.
- Given a type τ, we write τ`-list` to denote the type of lists having elements of type τ.
- The type *Identifier* classifies identifiers.

6.5.1 *SHL* and *Gimple* Syntax

In this subsection, we present only the parts of the syntax of *SHL* and *Gimple* necessary to ascribe the proposed weaving semantics. An environment is built from a *Gimple* program (*Program*) and a pattern (*SH_Pattern*). The *SHL* syntax describing a security hardening pattern is presented in Figure 16. We added labels to the syntax in order to use them in the semantics rules. A hardening pattern is based on the pointcut-advice model of AOP.

```
Environment          ::=  ⟨program:        Program,                                   (Environment)
                           pattern:        SH_Pattern
SH_Pattern           ::=  Pattern Pattern_Name                                         (Pattern)
                          Matching_Criteria?
                          SH_Pattern_Body
SH_Pattern_Body      ::=  Location_Behavior -list
Location_Behavior    ::=  ⟨insertionPoint:  before | after | replace,                  (Behavior)
                           location:        Location,
                           primitive:       Primitive -set,
                           code:            Behavior_Code ⟩
Location             ::=  Location_Identifier | Boolean_Location                       (location)
Location_Identifier  ::=  ⟨kind:            FunctionCall |
                          FunctionExecution | WithinFunction,
                           signature:       Fname⟩
                      |   ⟨kind:            set | get,
                           signature:       Vname⟩
                          ...
Boolean_Location     ::=  Location and Location
                      |   Location or Location
                      |   not Location
Behavior_Code        ::=  ⟨iRetType:        integer_type | real_type                   (Code)
                      |   boolean_type | void_type,
                           iName:           Fname ⟩
Fname                ::=  Identifer
Vname                ::=  Identifer
```

Figure 16: *SHL* Syntax

A *SH_Pattern* includes a list of behaviors (*Location_Behavior*). Each *Location_Behavior* specifies where (*insertionPoint*) and what (*code*) to insert at specific location *location*. The behavior *insertionPoint* specifies the point of code insertion after identifying the location. The behavior *insertionPoint* can have the following three values: `Before`, `After` or `Replace`. The insertion point `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* is composed of one or more *Location_Identifier* that identify the joint points in the program where the *Behavior_Code* should be integrated. The list of constructs used in *Location_Identifier* is left open for future extensions. Depending on the need of the security hardening solutions, a developer can define his own constructs. We consider the following base locations:

- `FunctionCall`: picks out the join points where we call a specific function.
- `FunctionExecution`: picks out the join points referring to the implementation of a specific function.
- `WithinFunction`: picks out the join points within a specific function.
- `set`: picks out the join points where we set a method local variable.
- `get`: picks out the join points where we get a method local variable.

The locations *Location* can be combined using logical operators to produce more complex ones. The code *Behavior_Code* that is going to be weaved is specified by its name and its return type. Actually this code could be provided as an interface or a library, or left to be implemented by the user.

Since *Gimple* contains a lot of constructs, only the ones needed to express the weaving semantics are chosen and presented in Figures 17 and 18. A *Gimple* program *Program* consists of the following main parts: a set of function declarations *funs*, a set of types *types*, and a set of constants *const*. A function declaration specifies the function name *fname*, the function type *ftype*, the argument declarations *args*, the result declaration *result*, and the function block *block*. The function block *Block* represented by `bind_expr` contains the

Program	::=	⟨*funs*:	*FunDecl* -set,	**(Program)**
		types:	*Type* -set,	
		const:	*Const* -set ⟩	
FunDecl	::=	⟨*kind*:	function_decl,	**(Function)**
		fname:	*Fname*,	
		ftype:	*FunType*,	
		args:	*ParmDecl* -set,	
		result:	*ResDecl*,	
		block:	*Block* ⟩	
Block	::=	⟨*ekind*:	bind_expr,	**(Block)**
		decl:	*VLDecl* -set,	
		body:	*Stmt*-list ⟩	
Stmt	::=	*ModifyStmt* | *CallStmt* | *Block*		**(Statement)**
ModifyStmt	::=	⟨*kind*:	modify_expr,	**(Assignment)**
		lhs:	*Lhs*,	
		rhs:	*Rhs* ⟩	
CallStmt	::=	⟨*kind*:	call_expr,	**(Function Call)**
		addrExpr:	*AddrExpr*,	
		arglist:	*VPDecl* -set ⟩	
IfStmt	::=	⟨*kind*:	cond_expr, **(If)**	
		condition:	*Condition*,	
		op$_1$:	*Stmt*-list,	
		op$_2$:	*Stmt*-list ⟩	
Lhs	::=	*ParmDecl* | *VarDecl* | *IndirectRef*		
Rhs	::=	*Const* | *Lhs* | *CallStmt* | *UnStmt* | *BinStmt* | *AddrStmt*		
UnStmt	::=	⟨*op*:	*Const* |*ParmDecl* | *VarDecl* | *AddrStmt* ⟩	
BinStmt	::=	⟨*op*$_1$:	*Const* |*ParmDecl* | *VarDecl* | *AddrStmt*,	
		op$_2$:	*Const* |*ParmDecl* | *VarDecl* | *AddrStmt* ⟩	
AddrExpr	::=	⟨*kind*:	addr_expr,	
		type:	*PointerType*,	
		op:	*VarDecl* | *FunDecl* ⟩	

Figure 17: *Gimple* Partial Syntax (Part 1)

IndirectRef	::=	⟨*kind*:	`indirect_ref,`	
		op:	*VarDecl* ⟩	
Condition	::=	*Const* | *ParmDecl* | *VarDecl* | *RelStmt*		
RelStmt	::=	⟨op_1:	*Const* | *ParmDecl* | *VarDecl* | *AddrStmt,*	
		op_2:	*Const* | *ParmDecl* | *VarDecl* | *AddrStmt* ⟩	
Type	::=	*IntType* | *RealType* | *BoolType*		**(Type)**
	|	*VoidType* | *PointerType* | *FunType*		
IntType	::=	⟨*kind*:	`integer_type` ⟩	
RealType	::=	⟨*kind*:	`real_type` ⟩	
BoolType	::=	⟨*kind*:	`boolean_type` ⟩	
VoidType	::=	⟨*kind*:	`void_type` ⟩	
PointerType	::=	⟨*kind*:	`pointer_type,`	
		type:	*FunType* | *IntType* | *RealType* ⟩	
FuncType	::=	⟨*kind*:	`function_type,`	
		type:	*IntType* | *RealType* | *BoolType* | *VoidType* ⟩	
VLDecl	::=	*LabelDecl* | *VarDecl*		**(Declaration)**
VPDecl	::=	*ParmDecl* | *VarDecl*		
ParmDecl	::=	⟨*kind*:	`parm_decl,`	
		name:	*Pname,*	
		type:	*IntType* | *RealType* | *BoolType* | *VoidType* ⟩	
ResDecl	::=	⟨*kind*:	`result_decl,`	
		name:	*Rname,*	
		type:	*IntType* | *RealType* | *BoolType* | *VoidType* ⟩	
VarDecl	::=	⟨*kind*:	`var_decl,`	
		name:	*Vname,*	
		type:	*IntType* | *RealType* | *BoolType* | *VoidType* ⟩	
LabelDecl	::=	⟨*kind*:	`label_decl,`	
		name:	*Lname,*	
		type:	*VoidType* ⟩	
Const	::=	*Nat*		**(Constant)**
Pname	::=	*Identifer*	*Rname*::= *Identifer*	
Vname	::=	*Identifer*	*Lname*::= *Identifer*	

Figure 18: *Gimple* Partial Syntax (Part 2)

declaration of the function variables and the function labels. In addition, multiple statements at the same nesting level are collected into a list of statements as the body *body* of a block.

There are several varieties of complex statements in *Gimple*. We consider statements that are shared between well-known programming languages such as assignment statement *ModifyStmt* represented by `modify_expr`, call statement *CallStmt* represented by `call_expr`, and conditional statements *IfStmt* represented by `cond_expr`. The modify statement has two parts: the left-hand side statement *Lhs* and the right-hand side statement *Rhs*. The left-hand side can be a variable declaration *VarDecl*, a parameter declaration *ParmDecl*, or an indirect reference *IndirectRef*, whereas the right-hand side can be one of the kinds of the left-hand side statements, a constant *Const*, a call statement *CallStmt*, a unary statement *UnStmt*, a binary statement *BinStmt*, or an address expression *AddrExpr*. Unary statements represent unary operations that have one operand. Binary statements represent binary operations that have two operands. An indirect reference represents a pointer variable defined using the indirect operator (*) in the `C` programming language and specified by `indirect_ref` and a variable declaration in *Gimple*. The address expression represents the operator (&) in `C` programming language and specified by `addr_expr`, a pointer type, and a variable declaration or a function declaration in *Gimple*. The call statement has two parts: the address expression *AddrStmt* and the function arguments *VPDecl* `-set`. The conditional statement has tree parts: the condition *Condition* and two statement lists *Stmt*`-list`. The condition can be either a constant *Const*, a variable declaration *VarDecl*, a parameter declaration *ParmDecl*, or a relational statement *RelStmt*. Relational statements represent relational operations that have two operands.

The considered base types are integer type represented by `integer_type`, real type represented by `real_type`, boolean type represented by `boolean_type`, and void type represented by `void_type`. Beside, there are two complex types: function type *FuncType* represented by `function_type` and pointer type *PoniterType* represented by `pointer_type`. A pointer type can specify an integer type, a real type, or a function type, which in its turn specifies the function return type.

Any declaration is specified by a kind, a name, and a type. The following declarations are considered: parameter declaration *ParmDecl* represented by `parm_decl`, variable declaration *VarDecl* represented by `var_decl`, result declaration *ResDecl* represented by `result_decl`, and label declaration *LabelDecl* represented by `label_decl`. Finally constants *Const* are represented by natural numbers.

6.5.2 Weaving Semantics

In this subsection, we provide the rules that describe the weaving semantics. First, we begin with the matching and then we continue with the weaving. Notice that *cs* $\in$ *CallStmt*, *loc* $\in$ *Location*, fd $\in$ *FunDecl*, bfd $\in$ *FunDecl*, ms $\in$ *ModifyStmt*, s $\in$ *Stmt*, t $\in$ *Type*, ft $\in$ *FuncType*, pt $\in$ *PointerType*, *ae* $\in$ *AddrExpr*, *beh* $\in$ *Location_Behavior*, and $\mathcal{E}$ $\in$ *Environment*.

Matching Rules

Rule *1* describes the case where the current statement in a function body is a call statement, the current location (*Location_Identifier*) in a pattern is a function call location (*FunctionCall*), and the location signature is equal to the called function specified in the call statement. In such a case, the call statement matches the function call location.

$$\frac{cs.kind = \texttt{call\_expr} \qquad loc.kind = \texttt{FunctionCall} \quad cs.addrExpr.op.fname = loc.signature}{fd, cs \vdash_{match} loc} \qquad \textbf{(1)}$$

Rule *2* describes the case where the current statement in a function body is a call statement, the current location (*Location_Identifier*) in a pattern is a *WithinFunction* location, and the location signature is equal to the name of the function where the call statement exists. In such a case, the call statement matches the *WithinFunction* location.

$$\frac{cs.kind = \texttt{call\_expr} \qquad loc.kind = \texttt{WithinFunction} \quad fd.fname = loc.signature}{fd, cs \vdash_{match} loc} \qquad \textbf{(2)}$$

Rule *3* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *set* location, and the location signature is equal to the name of the variable being set. In such a case, the assignment statement matches the *set* location.

$$\frac{ms.kind = \texttt{modify\_expr} \quad loc.kind = \texttt{set} \qquad ms.lhs.kind = \texttt{var\_decl} \quad ms.lhs.name = loc.signature}{fd, ms \vdash_{match} loc} \qquad \textbf{(3)}$$

Rule *4* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *get* location, and the location signature is equal to the name of the variable being get by a unary operation. In such a case, the assignment statement matches the *get* location.

$$\frac{ms.kind = \texttt{modify\_expr} \quad loc.kind = \texttt{get} \qquad ms.rhs.kind = \texttt{var\_decl} \quad ms.rhs.name = loc.signature}{fd, ms \vdash_{match} loc} \tag{4}$$

Rule *5* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *get* location, and the location signature is equal to the name of the variable being get. In such a case, the assignment statement matches the *get* location.

$$\frac{ms.kind = \texttt{modify\_expr} \quad loc.kind = \texttt{get} \qquad ms.rhs.op.kind = \texttt{var\_decl} \quad ms.rhs.op.name = loc.signature}{fd, ms \vdash_{match} loc} \tag{5}$$

Rule *6* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *get* location, and the location signature is equal to the name of the first variable being get by a binary operation. In such a case, the assignment statement matches the *get* location.

$$\frac{s.kind = \texttt{modify\_expr} \quad loc.kind = \texttt{get} \qquad ms.rhs.op_1.kind = \texttt{var\_decl} \quad ms.rhs.op_1.name = loc.signature}{fd, ms \vdash_{match} loc} \tag{6}$$

Rule *7* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *get* location, and the location signature is equal to the name of the second variable being get by a binary operation. In such a case, the assignment statement matches the *get* location.

$$\frac{\begin{array}{c} ms.kind = \texttt{modify\_expr} \qquad loc.kind = \texttt{get} \\ ms.rhs.op_2.kind = \texttt{var\_decl} \qquad ms.rhs.op_2.name = loc.signature \end{array}}{fd, ms \vdash_{match} loc} \quad \textbf{(7)}$$

Rule *8* describes the case where the current statement in a function body is an assignment statement, the current location (*Location_Identifier*) in a pattern is a *WithinFunction* location, and the location signature is equal to the name of the function where the assignment statement exists. In such a case, the assignment statement matches the *WithinFunction* location.

$$\frac{\begin{array}{c} ms.kind = \texttt{modify\_expr} \\ loc.kind = \texttt{WithinFunction} \qquad fd.fname = loc.signature \end{array}}{fd, ms \vdash_{match} loc} \quad \textbf{(8)}$$

Rule *9* describes the case where the locations in a pattern are combined using the and logical operators. In such a case, the current statement in a function body should match both locations in order to match their and combination.

$$\frac{fd, s \vdash_{match} loc_1 \qquad fd, s \vdash_{match} loc_2}{fd, s \vdash_{match} loc_1 \ \texttt{and} \ loc_2} \quad \textbf{(9)}$$

Rule *10* describes the case where the locations in a pattern are combined using the or logical operators. In such a case, the current statement in a function body should match only one of the locations (e.g., the first location) in order to match their or combination.

$$\frac{fd, s \vdash_{match} loc_1}{fd, s \vdash_{match} loc_1 \texttt{ or } loc_2} \quad \textbf{(10)}$$

Rule *11* describes the case where the locations in a pattern are combined using the `or` logical operators. In such a case, the current statement in a function body should match only one of the locations (e.g., the second location) in order to match their `or` combination.

$$\frac{fd, s \vdash_{match} loc_2}{fd, s \vdash_{match} loc_1 \texttt{ or } loc_2} \quad \textbf{(11)}$$

Rule *12* describes the case where the current statement in a function body does not match the current location (*Location_Identifier*) in a pattern. This can be expressed using the unary operator `not`.

$$\frac{fd, s \nvdash_{match} loc}{fd, s \vdash_{match} \texttt{not } loc} \quad \textbf{(12)}$$

Statement Creation Rule

Rule *13* describes how to create a call statement from a given behavior. The environment is changed as a result of such a creation.

$$\frac{\begin{array}{c} t = \mathsf{buildRetType}(beh.code) \quad ft = \mathsf{buildFunType}(t) \\ bfd = \mathsf{buildFunDecl}(beh.code, ft) \quad pt = \mathsf{buildFunPtr}(ft) \\ ae = \mathsf{buildAddrExpr}(pt, pfd) \quad cs = \mathsf{buildCallStmt}(ae) \\ \mathcal{E}'.program.types = \mathcal{E}.program.types \cup t \cup ft \cup pt \\ \mathcal{E}'.program.funs = \mathcal{E}.program.funs \cup bfd \end{array}}{\mathcal{E}, beh \vdash_{build} \mathcal{E}', cs} \quad \textbf{(13)}$$

In the sequel, we describe the utility functions used in Rule **(13)** and required for the statement creation:

- The function buildRetType builds a result type for the weaved function and adds it to the defined types in the program. It takes a behavior code and returns a type.

  buildRetType : $Code \rightarrow Type$

  buildRetType(c)=t where

$$\begin{cases} t.kind = \texttt{integer\_type} \quad \text{if } c.iRetType{=}\texttt{integer\_type}; \\ t.kind = \texttt{real\_type} \quad \text{if } c.iRetType{=}\texttt{real\_type}; \\ t.kind = \texttt{boolean\_type} \quad \text{if } c.iRetType{=}\texttt{boolean\_type}; \\ t.kind = \texttt{void\_type} \quad \text{if } c.iRetType{=}\texttt{void\_type}. \end{cases}$$

- The function buildFunType builds a function type for the weaved function and adds it to the defined types in the program. It takes a type and returns a function type.

buildFunType : *Type* $\rightarrow$ *FuncType*

buildFunType(t)=ft where $(ft.kind = \texttt{function\_type}) \wedge (ft.type = t)$

- The function buildFunDecl builds a function declaration for the weaved function and adds it to the declared functions in the program. It takes a behavior code and a function type. It returns a function declaration.

  buildFunDecl : *Code* $\times$ *FuncType* $\rightarrow$ *FuncDecl*

  buildFunDecl(c, ft)=fd

  where $(fd.kind = \texttt{function\_decl}) \wedge (fd.fname = c.iName) \wedge (fd.ftype = ft)$

- The function buildFunPtr builds a pointer type for the weaved function and add it to the defined types in the program. It takes a function type and returns a pointer type.

  buildFunPtr : *FuncType* $\rightarrow$ *PointerType*

  buildFunPtr(ft)=pt where $(pt.kind = \texttt{pointer\_type}) \wedge (pt.type = ft)$

- The function buildAddrExpr builds an address expression for the weaved function. It takes a pointer type and a function declaration. It returns an address expression.

  buildAddrExpr : *PointerType* $\times$ *FunDecl* $\rightarrow$ *AddrExpr*

  buildAddrExpr(pt, fd)=ae

  where $(ae.kind = \texttt{addr\_expr}) \wedge (ae.type = pt) \wedge (ae.op = fd)$

- The function buildCallStmt builds a call statement to the weaved function based on the address expression. It takes an address expression and returns a call statement.

  buildCallStmt : *AddrExpr* $\rightarrow$ *CallStmt*

  buildCallStmt(ae)=cs where $(cs.kind = \texttt{call\_expr}) \wedge (cs.addrExpr = ae)$

Weaving Rules

A function body is composed of a list of statements *l*, followed by the current statement *s*, which is followed by another list of statements *l'*. Rule *14* describes the case where the current statement in a function body matches the current location in a pattern and the insertion point corresponding to the current location is *Before*. In such a case, the call statement *cs* corresponding to the location behavior is built and inserted before the matched statement in the function body.

$$\frac{fd.block.body = l@(s :: l') \quad fd, s \vdash_{match} beh.loc \qquad \mathcal{E}, beh \vdash_{build} \mathcal{E}', cs \quad beh.insertionPoint = \texttt{Before}}{\langle \mathcal{E}, fd.block.body \rangle \rightarrow \langle \mathcal{E}', l@(cs :: s :: l') \rangle} \quad \textbf{(14)}$$

A function body is composed of a list of statements *l*, followed by the current statement *s*, which is followed by another list of statements *l'*. Rule *14* describes the case where the current statement in a function body matches the current location in a pattern and the insertion point corresponding to the current location is *After*. In such a case, the call statement *cs* corresponding to the location behavior is built and inserted after the matched statement in the function body.

$$\frac{fd.block.body = l@(s :: l') \quad fd, s \vdash_{match} beh.loc \qquad \mathcal{E}, beh \vdash_{build} \mathcal{E}', cs \quad beh.insertionPoint = \texttt{After}}{\langle \mathcal{E}, fd.block.body \rangle \rightarrow \langle \mathcal{E}', l@(s :: cs :: l') \rangle} \quad \textbf{(15)}$$

A function body is composed of a list of statements *l*, followed by the current statement *s*, which is followed by another list of statements *l'*. Rule *14* describes the case where the current statement in a function body matches the current location in a pattern and the insertion point corresponding to the current location is *Replace*. In such a case, the call statement *cs* corresponding to the location behavior is built and supersedes the matched statement in the function body.

$$\frac{\begin{array}{c} fd.block.body = l@(s :: l') \quad fd, s \vdash_{match} beh.loc \\ \mathcal{E}, beh \vdash_{build} \mathcal{E}', cs \quad beh.insertionPoint = \texttt{Replace} \end{array}}{\langle \mathcal{E}, fd.block.body \rangle \rightarrow \langle \mathcal{E}', l@(cs :: l') \rangle} \quad \textbf{(16)}$$

Rule *17* propagates the weaving changes applied to a specific function to higher levels, i.e., program and environment where the corresponding function exists.

$$\frac{\begin{array}{c} fd' = fd \quad \langle \mathcal{E}, fd.block.body \rangle \rightarrow \langle \mathcal{E}', l \rangle \quad fun = \mathcal{E}'.program.funs \quad fd' \in fun \\ \mathcal{E}'.program.funs = (fun - fd') \cup fd \end{array}}{\langle \mathcal{E} \rangle \rightarrow \langle \mathcal{E}' \rangle} \quad \textbf{(17)}$$

6.6 Implementation of *Gimple* Weaving Capabilities into *GCC*

Few weaving capabilities of the proposed semantics for *Gimple* weaving have been implemented into the *GCC* compiler. As a result, we are able now to apply several hardening practices on the *Gimple* representation (tree) of a program before generating the corresponding executable. Here is the implementation methodology.

First, the extended *GCC* is interrupted once the *Gimple* tree of the compiled program is built. This is done by adding a new pass to *GCC* that can be called by selecting an option when performing the compilation (e.g., `gcc -Weaving SecureConnectionPattern.shl -c Connection.c ...`). Then, the selected hardening pattern is compiled and a *Gimple* tree is built for the *Code* of each one of its *Behavior*(s) (Please see Section 6.5.1 for more details on *SHL* syntax). The needed information of the pattern *Behavior*(s) (e.g., function name, return type, etc.) is gathered from the *SHL* parser and passed as parameters to specific functions provided by *GCC* and responsible of building and modifying the *Gimple* trees (e.g., `buildFunctionDeclarationTree(...)`). Afterwards, each link to a generated tree is injected in the original program tree with respect to the *insertionPoint* and *location* specified in each *Location_Behavior.* Once this weaving procedure is done, *GCC* takes over and continues the classical compilation of the modified *Gimple* tree to generate the executable of the hardened program.

6.7 Case Study: Performing Security Hardening in the *Gimple* Representation of Software

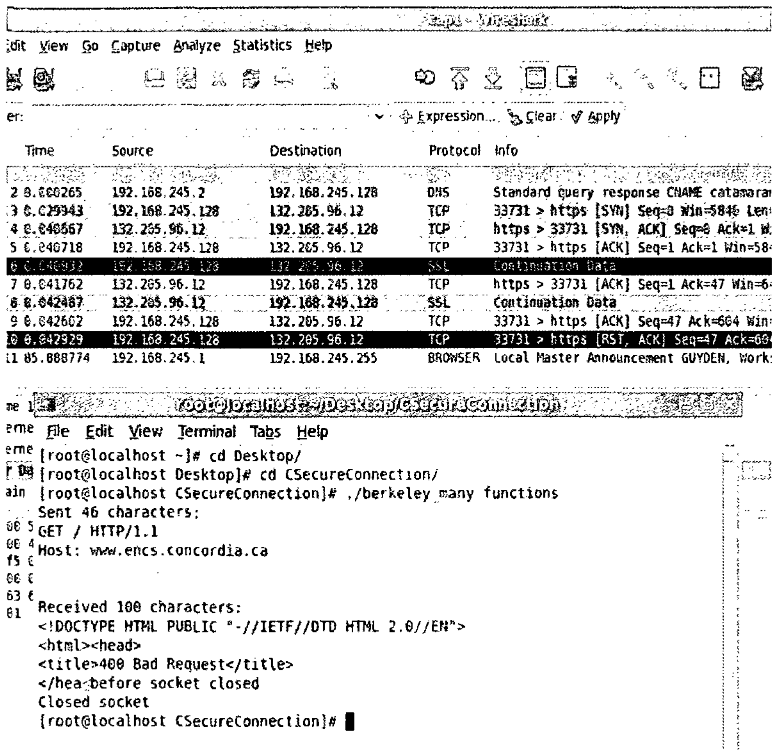
In this section, we present a case study for securing the connections of client applications. Securing channels between two communicating parties allow to avoid eavesdropping, tampering with the transmission, or session hijacking. In this context, we have selected a client application implemented in C++ which allows to connect and exchange data with a server through HTTP requests. To demonstrate the feasibility of our proposition, we have elaborated first, using *SHL*, the security hardening pattern needed to secure the connections of a selected client application. Listings 4.11 and 4.12 in Chapter 4 presents the pattern elaborated in *SHL* for securing the connection of similar application using GnuTLS/SSL. The code of the functions used in the *Code* of the pattern *Behavior*(s) is illustrated in Listing 4.13. Then, we have applied our initial methodology for hardening, where we refined the pattern into AspectC++ aspect and weaved it into the selected application. Afterwards, we have repeated the hardening using our new proposition, where we have compiled directly the same application and the hardening pattern using the extended *GCC* and applied the weaving on the *Gimple* representation of the application. Indeed, this case study explores also the relevance of elaborating the operational semantics for *Gimple* weaving, as initial attempt toward full implementation of a *Gimple* weaver.

Applying the hardening on the *Gimple* representation of code does not require anymore refining the hardening pattern into aspect. Compiling the selected client application, by using our extended *GCC*, specifying the weaving option and selecting the hardening pattern for securing connection to be weaved into the application, is enough to perform the hardening and generate the executable of the hardened application. In the sequel, we provide the compilation steps. *GCC* compiles first the client application and is interrupted once the *Gimple* tree is generated. Then, the developed weaving capabilities take over and the needed information of the hardening pattern for securing connection are gathered. The pattern is treated *Location_Behavior* by *Location_Behavior*, where a *Gimple* tree is built for the *Code* of each one of them and weaved into the application *Gimple* tree at the place specified in the *insertionPoint* and *location* of the *Location_Behavior*. Afterwards, *GCC* continues its classical compilation of the modified tree and generates finally the executable of the hardened client application. The resulted application is able now to connect securely through HTTPS.

Experimental Results

In order to verify the hardening correctness, we have set first in the original application the server port number to 443, which means the client and the server can only communicate through HTTPS (ssl-mode). Any communication through HTTP won't be understood and will fail. Then, we have compiled and run the client application and made it connect to the server (www.encs.concordia.ca) to retrieve information. The experimental results in Figure 19 show that the application failed to retrieve successfully the information. The server replies with a bad request because it is not able to understand the message content (Please see the run in the terminal of Figure 19). The highlighted lines in the Wireshark capture of the traffic show that the communication fails and stops after exchanging few undetermined messages.

Afterwards, we have applied our both approaches to harden this client application. First, we have weaved and compiled (using AspectC++ weaver and g++) the elaborated aspect (Listing 5.29 in Chapter 5 shows an excerpt of this aspect) with the different components of the application. Then, we have compiled the same original application using the extended *GCC* and enabling the *Gimple* weaving option. Running the two generated executables gives exactly the same results on the terminal and in the Wireshark packet captures. Due to this and to avoid duplication, we present in Figure 20 only the run of the application



```
Time        Source            Destination       Protocol Info
2 0.000265  192.168.245.2     192.168.245.128   DNS      Standard query response CNAME catamaran
3 0.029943  192.168.245.128   132.205.96.12     TCP      33731 > https [SYN] Seq=0 Win=5840 Len
4 0.040567  132.205.96.12     192.168.245.128   TCP      https > 33731 [SYN, ACK] Seq=0 Ack=1 W
5 0.040718  192.168.245.128   132.205.96.12     TCP      33731 > https [ACK] Seq=1 Ack=1 Win=58
6 0.040932  192.168.245.128   132.205.96.12     SSL      Continuation Data
7 0.041762  132.205.96.12     192.168.245.128   TCP      https > 33731 [ACK] Seq=1 Ack=47 Win=6
8 0.042487  132.205.96.12     192.168.245.128   SSL      Continuation Data
9 0.042602  192.168.245.128   132.205.96.12     TCP      33731 > https [ACK] Seq=47 Ack=604 Win
10 0.042929 192.168.245.128   132.205.96.12     TCP      33731 > https [RST, ACK] Seq=47 Ack=60
11 85.888774 192.168.245.1    192.168.245.255   BROWSER  Local Master Announcement GUYDEN, Work
```

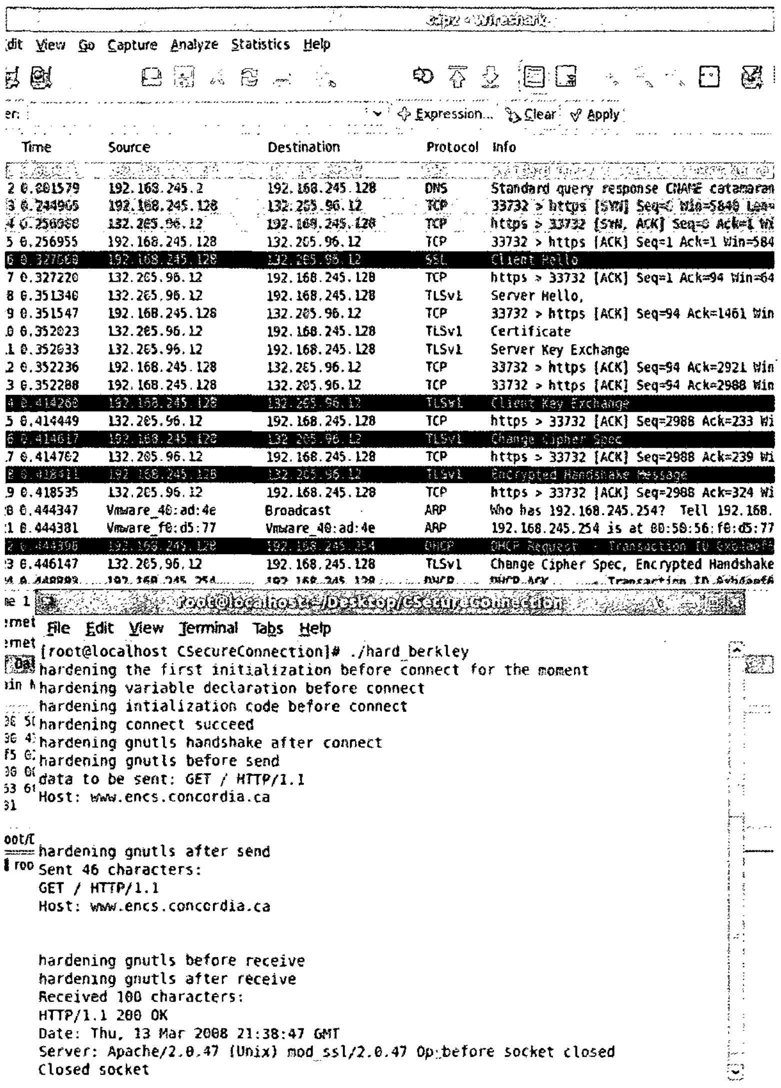
```
[root@localhost ~]# cd Desktop/
[root@localhost Desktop]# cd CSecureConnection/
[root@localhost CSecureConnection]# ./berkeley many functions
Sent 46 characters:
GET / HTTP/1.1
Host: www.encs.concordia.ca



Received 100 characters:
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</hea before socket closed
Closed socket
[root@localhost CSecureConnection]#
```

Figure 19: Capture of Connection

hardened by the *Gimple* weaving capabilities. The experimental results (Please see the run in the terminal and the highlighted lines in the Wireshark capture of Figure 20) explore that the new secure application is able to connect through HTTPS connections. It is also able to exchange successfully the data from the server in ssl-mode and encrypted form, exploring the feasibility and correctness of the security hardening process.

6.8 Conclusion

We presented in this chapter our accomplishment towards ascribing the formal specification of the proposed framework for systematic security hardening. In this context, we enriched



```
 Time       Source            Destination       Protocol Info
2 0.201579  192.168.245.2     192.168.245.128   DNS      Standard query response CNAME catamaran
3 0.244965  192.168.245.128   132.205.96.12     TCP      33732 > https [SYN] Seq=0 Win=5840 Len=
4 0.256908  132.205.96.12     192.168.245.128   TCP      https > 33732 [SYN, ACK] Seq=0 Ack=1 Wi
5 0.256955  192.168.245.128   132.205.96.12     TCP      33732 > https [ACK] Seq=1 Ack=1 Win=584
6 0.327069  192.168.245.128   132.205.96.12     SSL      Client Hello
7 0.327220  132.205.96.12     192.168.245.128   TCP      https > 33732 [ACK] Seq=1 Ack=94 Win=64
8 0.351340  132.205.96.12     192.168.245.128   TLSv1    Server Hello,
9 0.351547  192.168.245.128   132.205.96.12     TCP      33732 > https [ACK] Seq=94 Ack=1461 Win
0 0.352023  132.205.96.12     192.168.245.128   TLSv1    Certificate
1 0.352033  132.205.96.12     192.168.245.128   TLSv1    Server Key Exchange
2 0.352236  192.168.245.128   132.205.96.12     TCP      33732 > https [ACK] Seq=94 Ack=2921 Win
3 0.352288  192.168.245.128   132.205.96.12     TCP      33732 > https [ACK] Seq=94 Ack=2988 Win
4 0.414260  192.168.245.128   132.205.96.12     TLSv1    Client Key Exchange
5 0.414449  132.205.96.12     192.168.245.128   TCP      https > 33732 [ACK] Seq=2988 Ack=233 Wi
6 0.414617  192.168.245.128   132.205.96.12     TLSv1    Change Cipher Spec
7 0.414762  132.205.96.12     192.168.245.128   TCP      https > 33732 [ACK] Seq=2988 Ack=239 Wi
8 0.418411  192.168.245.128   132.205.96.12     TLSv1    Encrypted Handshake Message
9 0.418535  132.205.96.12     192.168.245.128   TCP      https > 33732 [ACK] Seq=2988 Ack=324 Wi
0 0.444347  Vmware_40:ad:4e   Broadcast         ARP      Who has 192.168.245.254?  Tell 192.168.
1 0.444381  Vmware_f0:d5:77   Vmware_40:ad:4e   ARP      192.168.245.254 is at 00:50:56:f0:d5:77
2 0.444396  192.168.245.128   192.168.245.254   DHCP     DHCP Request  - Transaction ID 0x0dae
3 0.446147  132.205.96.12     192.168.245.128   TLSv1    Change Cipher Spec, Encrypted Handshake
```

```
[root@localhost CSecureConnection]# ./hard_berkley
hardening the first initialization before connect for the moment
hardening variable declaration before connect
hardening intialization code before connect
hardening connect succeed
hardening gnutls handshake after connect
hardening gnutls before send
data to be sent: GET / HTTP/1.1
Host: www.encs.concordia.ca


hardening gnutls after send
Sent 46 characters:
GET / HTTP/1.1
Host: www.encs.concordia.ca



hardening gnutls before receive
hardening gnutls after receive
Received 100 characters:
HTTP/1.1 200 OK
Date: Thu, 13 Mar 2008 21:38:47 GMT
Server: Apache/2.0.47 (Unix) mod_ssl/2.0.47 Op:before socket closed
Closed socket
```

Figure 20: Capture of Hardened Connection

our proposition presented in Chapter 4 by elaborating a new approach for applying security hardening on the *Gimple* representation of software. This approach allows to avoid in some cases the manual refinement of the security hardening solution into the current AOP languages, and hence weave the security concerns during the compilation into the *Gimple* tree instead of the code. It also enables to weave an application written in a specific language with code written in a different one by exploiting the *Gimple* intermediate representation. Then, we provided a formal specification of the security hardening solutions developed by the proposed framework. This has been done through the elaboration of the formal syntax of *SHL* and *Gimple* and the operational semantics of *SHL* weaving based on the *Gimple* depiction of software. This formal specification constitutes an initial attempt and a guide toward developing a complete weaver for *Gimple*. It also provides support for the whole framework to eventually apply formal verification on the security hardening solutions. We realized and demonstrate the feasibility of our propositions by: (1) Implementing into *GCC* several *Gimple* weaving features described in the formal semantics and (2) developing a case study where the hardening is applied on the *Gimple* representation of the application and compiled using the extended *GCC*.

Chapter 7

Conclusion

Security is taking an increasingly predominant role in today computing world. On the other hand, plethora of high quality open source software have been designed and implemented without having security in mind. Such software are currently used into high-risk network/web environments. This leads to the discovery of several flaws and vulnerabilities that have been eventually exploited by different attacks. It also required security mechanisms to be added and integrated into the software for protection. In this context, the security hardening of open source software, which is addressed thoroughly in this thesis, becomes a very challenging and interesting problem.

In the sequel, we summarize briefly the main thesis contributions:

- Aspect-Oriented and pattern-based approach for systematic security hardening of software without the need to high security expertise.

- Programming language independent and aspect-oriented language for security hardening called *SHL*.

- New Aspect-Oriented pointcut and primitive constructs for security hardening concerns.
- Approach for weaving the security hardening concerns on the *Gimple* representation of software.
- Formal operational semantics of *SHL* weaving.

Technical Summary

Although the current approaches for software security hardening target security during the development of new software, as is the case of security design patterns and secure programming techniques, they may still be relevant and give resolutions for several security problems and requirements. However, they have major shortcomings regarding their methodologies for applying the security solutions into software. They are all based on performing security hardening in ad-hoc and manual manners and require high security expertise. On the other hand, the procedures of security hardening are difficult and critical. If they are applied manually, they need lot of time to be tackled and may create other vulnerabilities, especially when dealing with large scale software (e.g., thousands, millions lines of code). They also require the developers to take important and significant decisions that entail high expertise in both the security and the software functionality domains. This causes another problem consisting in the difficulty of finding the developers specialized in the both aforementioned domains.

These issues have been addressed in the proposed AOP and pattern-based approach for systematic security hardening. Adopting AOP allows performing the security hardening procedures in a systematic way and avoiding the manual integration of security components and code into software. However, using only AOP for security hardening still requires high expertise in both the security solutions applied and the software functionality domain. The developers still need to specify into the aspects all the steps needed to implement the security solutions and the target where to apply them. To solve this problem, we opted in the proposed approach to increase the abstraction of the security hardening solution and separate totaly the roles and duties of the security experts from the ones of the developers. The security experts are able to provide into hardening patterns well-defined solutions to particular security problems with all the details on why, how and where to apply them. On the other side, the developers are able to use these solutions to harden open source software by refining the hardening patterns into AOP aspects and specifying high-level security hardening plans. The developers do not need to have expertise in the applied security solutions.

The realization of the proposed approach has been achieved by elaborating a programming independent and aspect-oriented based language for security hardening called *SHL*, developing its corresponding parser, compiler and facilities and integrating all of them into a framework for software security hardening. The resulting framework allows the description of security hardening plans and patterns using *SHL* and the execution of all the required steps for systematic security hardening of software. We illustrated the feasibility of the elaborated framework by developing several security hardening solutions that are dealing with security requirements and vulnerabilities (e.g., securing connections, adding authorization, encrypting some information in the memory, and remedying low level security vulnerabilities) and applying them on large scale software (e.g., APT and MySQL).

The proposed approach requires to refine the *SHL* security hardening solution into an AOP aspect (e.g., in AspectC++, in AspectJ) before applying and weaving it into the software. On the other hand, the current AOP technologies were not initially designed to target security, and hence they have some related limitations. Consequently some security hardening activities requires manual intervention and addition of components and code, which may affect the systematic target of the proposed approach. For instance, in our experiment for securing the connections of client applications, we faced the problem of passing needed parameters related to the security library between the application components. Since the current AOP languages miss such feature, we opted to integrate additional modules and changing some internal functions to do so. However, such solution may not work with other software that have complex dependencies and relations between its components.

These issues have been addressed by elaborating new pointcut and primitive constructs to *SHL* and AOP languages that provide features needed for systematic security hardening concerns. The two proposed pointcuts, *GAFlow* and *GDFlow*, return particular join points in a program CFG where security features common between a set of join points can be added. The *GAFlow* and *GDFlow* allow to analyze the CFG execution paths to identify respectively the closest guaranteed ancestor and closest guaranteed descendant join points according to the pointcuts of interest. The two proposed primitives, *ExportParameter* and *ImportParameter*, are used to pass parameters between two pointcuts. They allow to analyze a program call graph in order to determine how to change function signatures for passing the parameters associated with a given security hardening functionality. We explored the viability of the proposed pointcuts and primitives by elaborating and implementing their methodologies and algorithms and presenting the result of explanatory case studies.

In the original proposed approach for security hardening, the refinement of the patterns into AOP aspects or low level code does not require security expertise. However, this task still requires human interactions to perform the refinement and provide the parameters needed for the concrete implementation. Another concern consists in the need to ascribe a formal description of the *SHL* language, and hence for the whole security hardening solutions performed by the proposed framework.

These issues have been addressed first by enriching the proposed framework with a new approach for applying security hardening on the *Gimple* representation of software. This approach allows the weaving of security concerns into the *Gimple* tree of the software during the *GCC* compilation. Accordingly, it provides more systematization to the initial proposition by bypassing in some cases the manual refinement of the security hardening solution into AOP aspects. Then, the formal specification of the security hardening solution has been provided through the elaboration of formal syntax for *SHL* and *Gimple* and an operational semantics of *SHL* weaving based on the *Gimple* depiction of software. Targeting *Gimple* for such formal description is relevant because *SHL* and the hardening solution described using *SHL* are abstract and programming language independent. The elaborated operational semantics allow to understand the inner working of *Gimple* procedures and constitutes a guide for developing a complete aspect-oriented weaver for *Gimple*. Eventually, it may also allow to perform formal verification on the framework security hardening solutions.

The realization of these propositions has been achieved by implementing into *GCC* few features described in the *SHL* weaving semantics and developing a case study, in which the security hardening concerns are applied and weaved using the extended *GCC* into the *Gimple* representation of an application. These contributions improve significantly the approach for systematic security hardening. Besides, they also constitutes by themselves the first attempts towards adopting aspect-oriented programming on *Gimple*, exploring it into a formal operational semantics and exploiting *Gimple* intermediate representation to advise an application written in a specific programming language with code written in a different one.

Future Work

Currently, we are working on adapting our approach for the systematic security hardening of software at the design level. This direction will allow us to reuse the advanced literature in the domain of security engineering (i.e., security design patterns), and consequently provide a methodology for applying them while designing software. We will also benefit from the achievements and experiments of our framework for security hardening of code, build on top of it and/or reuse some of its components to reach our goal.

As future directions, we are planning to

- Address the problems related to security engineering and security design patterns.
- Provide methodologies for applying security patterns during the different life cycle of software development.
- Adapt and apply our approach on more specific security domain and provide AOP solutions for more security issues.

- Deploy the proposed pointcut and primitive constructs into the current AOP technologies (i.e., AspectC++ and AspectJ).

- Address other limitations of the AOP technologies for security hardening concerns through elaborating new pointcut and primitive constructs.

- Build more *Gimple* weaving capabilities towards developing a complete weaver for *Gimple*.

- Apply formal verification on the *SHL* security hardening solutions using the proposed weaving semantics.

List of Publications

The following is the list of publications derived from the thesis work:

Book Chapters

1. A Security Hardening Language Based On Aspect-Orientation. To appear in E-Business and Telecommunication Networks Book, 2009, Springer.

2. Middleware Security in Wireless Applications. In the Encyclopedia of Wireless and Mobile Communications Book, CRC Press, 2008, Taylor & Francis Group.

Journal Papers

1. An Aspect Oriented Approach for the Systematic Security Hardening of Code. In the Journal of Computers and Security, Volume 27, Issues 3-4, Pages 101–114, May-June 2008, Elsevier.

2. A High-Level Aspect-oriented Based Framework for Software Security Hardening. In the Information Security Journal: A Global Perspective, Volume 17, Issue 2, Pages 56 – 74, May 2008, Taylor & Francis Group.

3. Nouveaux Points de coupure et Primitives pour les préoccupations de renforcement de sécurité. To appear in the Technique et Science Informatiques (TSI) Journal, 2009, Hermes/Lavoisier.

4. Security Hardening of Open Source Software. In the Open Source Business Resource Journal, June 2008, Open Journal Systems.

Conference Papers

1. Cross-Language Weaving Approach Targeting Software Security Hardening. In Proceedings of the Sixth Annual Conference on Privacy, Security and Trust (PST 2008), October 1-3, 2008, Fredericton, New Brunswick, Canada, IEEE.

2. Towards Language-Independent Approach for Security Concerns Weaving. In Proceedings of the International Conference on Security and Cryptography (ICETE-Secrypt 2008), July 26-29, 2008, Porto, Portugal.

3. Control Flow Based Pointcuts for Security Hardening Concerns. In Proceedings of the Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM 2007), July 30 - August 2, 2007, Moncton, New Brunswick, Canada, IFIP - Springer.

4. A High-Level Aspect-Oriented Based Language for Software Security Hardening. In Proceedings of the International Conference on Security and Cryptography (ICETE-Secrypt 2007), July 28-31, Barcelona, Spain (Winner of the Best Student Paper Award).

5. New Primitives to AOP Weaving Capabilities for Security Hardening Concerns. In Proceedings of the 9th International Conference on Enterprise Information Systems, Security in Information Systems Symposium (ICEIS-WOSIS 2007), June 12-13, 2007, Funchal, Madeira, Portugal.

6. Towards an Aspect Oriented Approach for the Security Hardening of Code. In Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications, Security in Networks and Distributed Systems Symposium (AINA-SSNDS), May 21-23, 2007, Niagara Falls, Canada, IEEE.

7. Points de coupure pour les préoccupations de renforcement de sécurité utilisant le flot de contrôle. In Proceedings of the 3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007, AOSD), March 26, 2007, Toulouse, France.

8. Security Hardening for Open Source Software. In Proceeding of the ACM Conference on the Privacy, Security, Trust 2006, PST 2006, Oct 30 - Nov 1, 2006, Markham, Ontario, Canada, ACM/McGraw-Hill.

9. Security Design Patterns: Survey and Evaluation. In Proceeding of the IEEE Canadian Conference on Electrical and Computer Engineering, CCECE 2006, May 7-10, 2006, Ottawa, Ontario, Canada, IEEE.

Bibliography

[1] Advanced packaging tool. Available at `http://www.debian.org/doc/manuals/apt-howto/` (accessed on 2008/11/11).

[2] Debian apache-ssl package. Available at `http://packages.debian.org/etch/apache-ssl` (accessed on 2008/11/11).

[3] Documentation: Aspectc++ language reference. Availabe at `http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf` (accessed on 2008/11/11).

[4] Gnu transport layer security library. Available at `http://www.gnu.org/software/gnutls/` (accessed on 2008/11/11).

[5] Java authentication and authorization service. Available at `http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html` (accessed on 2008/11/11).

[6] Bastille linux, 2006. Available at `http://www.bastille-linux.org/` `(accessed on 2008/11/11).`

[7] ISO/IEC 10181-1:1996. Security frameworks for open systems: Overview, 1996.

[8] ISO/IEC 7498-2:1989. Information processing systems – open systems interconnection – basic reference model – part 2: Security architecture, 1989.

[9] Rickard A. Aberg, Julia L. Lawall, Mario Sudholt, Gilles Muller, and Anne-Francoise Le Meury. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE03)*. IEEE, 2003.

[10] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.

[11] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University, 1977.

[12] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[13] S Bhatkar, DC DuVarne, and R Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[14] Matt Bishop. Writing safe secure programs, 1997. Available at `http://nob.cs.ucdavis.edu/bishop/secprog/ns1997.pdf` (accessed on 2008/11/11).

[15] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[16] Matt Bishop. How attackers break programs, and how to write more secure programs, 2005. Available at `http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html` (accessed on 2008/11/11).

[17] Matt Bishop and Dave Bailey. A critical analysis of vulnerability taxonomies. Technical Report CSE-96-11, Department of Computer Science, University of California at Davis, 1996.

[18] Bob Blakley, Craig Heath, and members of The Open Group Security Forum. Security design patterns. Technical Report G031, Open Group, 2004.

[19] Kai Böllert. On weaving aspects. In *Proceeding of the International Workshop on Aspect-Oriented Programming at ECOOP99*, 1999.

[20] Ron Bodkin. Enterprise security aspects. In *Proceedings of the Workshop on AOSD Technology for Application-level Security (AOSD04:AOSDSEC)*, 2004.

[21] Jonas Bonér. Semantics for a synchronized block join point, 2005. Available at `http://www.jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-join-point/` (accessed on 2008/11/11).

[22] 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.

[23] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. muABC: A minimal aspect calculus. In *Proceedings of the Fifteenth International Conference on Concurrency Theory (CONCUR 2004)*, volume 3170 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[24] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[25] David Callahan, Alan Carle, Mary Wolcott Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, 1990.

[26] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD07:Proceedings of the 6th international conference on Aspect-oriented software development*. ACM, 2007.

[27] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of Foundations of Software Engineering*. ACM, 2001.

[28] Keith Cooper, Timothy Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 4(1-10), 2001.

[29] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Stateful aspects: The case for aspect-oriented modeling. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*. ACM, 2007.

[30] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ACM SIGPLAN, 2005.

[31] Bart DeWin. *Engineering Application Level Security through Aspect Oriented Software Development*. PhD thesis, Katholieke Universiteit Leuven, 2004.

[32] Bart DeWin, Bart Vanhaute, and Bart 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*, 2001.

[33] Bart DeWin, Bart Vanhaute, and Bart De Decker. How aspect-oriented programming can help to build secure software. *Informatica (Slovenia)*, 26(2), 2002.

[34] Edsger Dijkstra. Dijkstra's algorithm. Available at `http://en.wikipedia.org/wiki/Dijkstra_algorithm` (accessed on 2008/11/11).

[35] 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 *AOSD05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005.

[36] Christopher Dutchyn, Gregor Kiczales, and Hidehiko Masuhara. Aspect Sand Box Project, 2002. Available at `http://www.cs.ubc.ca/labs/spl/projects/asb.html` (accessed on 2008/11/11).

[37] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005.

[38] 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.

[39] Eduardo B. Fernandez and Reghu Warrier. Remote authenticator/authorizer. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP 2003)*, 2003.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[41] Ernesto Gomez. Cs624- notes on control flow graph, 2003. Available at `http://www.csci.csusb.edu/egomez/cs624/cfg.pdf` (accessed on 2008/11/11).

[42] Mark G. Graff and Ken v. Wyk. *Secure Coding: Principles and Practices*. O'Reilly & Associates, 2003.

[43] Demeter Group. Available at `http://www.ccs.neu.edu/research/demeter/` (accessed on 2008/11/11).

[44] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.

[45] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[46] Dima Hadidi, Nadia Belblidia, and Mourad Debbabi. Security crosscutting concerns and AspectJ. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM, 2006.

[47] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March*, 2005.

[48] Gabriel Hermosillo, Roberto Gomez, Lionel Seinturier, and Laurence Duchien. Aprosec: an aspect for programming secure web applications. In *Proceedings of the Second International Conference on Availability, Reliability and Security (ARES07)*. IEEE, 2007.

[49] Glenn Holloway and Michael D. Smith. The machine-suif control flow analysis library. Harvard University, 1998. Available at `http://www.eecs.harvard.edu/machsuif/software/nci/cfa.html` (accessed on 2008/11/11).

[50] Michael Howard and David E. LeBlanc. *Writing Secure Code*. Microsoft, Redmond, WA, USA, 2002.

[51] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft, Redmond, WA, USA, 2006.

[52] Minhuan Huang, Chunlei Wang, and Lufeng Zhang. Toward a reusable and generic security aspect library. In *Proceedings of the Workshop on AOSD Technology for Application-level Security (AOSD04:AOSDSEC)*, 2004.

[53] Norman L. Kerth and Ward Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1):53–59, 1997.

[54] Gregor Kiczales. The fun has just begun, keynote talk at AOSD 2003, 2003. Available at `http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt` (accessed on 2008/11/11).

[55] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Overview of AspectJ. In *Proceedings of the 15th European Conference ECOOP 2001*. Springer Verlag, 2001.

[56] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository, 2002. Available at `http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf` (accessed on 2008/11/11).

[57] Howard Kim. An AOSD implementation for C#. Technical Report TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[58] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS03)*, 2003.

[59] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *ICFP05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. ACM, 2005.

[60] John McCormick. OpenBSD declares war on buffer overruns. *TechRepublic*, 2003. Available at `http://techrepublic.com.com/5100-1035_11-5034831.html` (accessed on 2008/11/11).

[61] Andrew Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL99)*. ACM, 1999.

[62] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *AOSD06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*. ACM, 2006.

[63] Doug Orleans. Adaptive programming with traversals and visitors. Available at `http://www.ccs.neu.edu/research/demeter/posters/introDemeter-Java/poster.html` (accessed on 2008/11/11).

[64] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer*, 2000.

[65] Frank G. Pagan. *Formal Specification of Programming Languages*. Prentice-Hall, Inc., 1981.

[66] Terence Parr. Antlr. Available at `http://www.antlr.org` (accessed on 2007/11/11).

[67] G.D. Plotkin. A structural approach to operational semantics. *Logic and Algebraic Programming*, 60-61:17–139, 2004.

[68] Sasha Romanosky. Security design patterns part 1, 2001. Available at `http://www.romanosky.net/` (accessed on 2008/11/11).

[69] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216– 226, 1979.

[70] Markus Schumacher. *Security Engineering with Patterns*. Springer, 2003.

[71] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[72] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire Linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE, 2005.

[73] Robert C. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.

[74] Viren Shah. An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, 2003.

[75] 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)*, 2002.

[76] Viren Shah and Frank Hill. Aspect-oriented programming security framework. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 03)*. IEEE, 2003.

[77] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Language: A Laboratory Based Approach*. Addison-Wesley Publishing Company, Inc., 1995.

[78] 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.

[79] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, 2002.

[80] Peri Tarr and Harold Ossher. HyperJ user and installation manual, 2000. IBM T. J.Watson Research Center, Yorktown Heights, NY, USA.

[81] Bart Vanhaute and Bart DeWin. Security and genericity. In *Proceedings of the 1st Belgian AOSD Workshop*, 2001.

[82] John Viega, J.T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[83] John Viega and Matt Messier. *Secure Programming Cookbook For C and C++*. O'Reilly Media, Inc., 2003.

[84] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM, 2003.

[85] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.

[86] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. In *PEPM06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2006.

[87] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. Available at `http://www.dwheeler.com/secure-programs/` (accessed on 2008/11/11).

[88] 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)*. IEEE, 2006.

[89] Zhenrong Yang. On building a dynamic vulnerability detection system. Master's thesis, Concordia University, 2007.

[90] 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.

[91] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++: A survey of vulnerabilities and countermeasures. Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, July 2004.