Cross-Language Weaving Approach Targeting Software Security Hardening

Azzam Mourad, Dima Alhadidi and Mourad Debbabi
Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{mourad,dm_alhad,debbabi}@encs.concordia.ca *

Abstract

In this paper, we propose an approach for systematic security hardening of software based on aspect-oriented programming and Gimple language. We also present the first steps towards a formal specification for Gimple weaving together with the implementation methodology of the proposed weaving semantics. The primary contribution of these propositions is providing the software architects with the capabilities to perform systematic security hardening by applying well-defined solutions and without the need to have expertise in the security solution domain. We explore the viability of our propositions by realizing the weaving semantics for Gimple by implementing it into the GCC compiler and applying our methodologies for systematic security hardening to develop a case study for securing the connections of client applications together with experimental results.

1. Introduction

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 legacy systems must be adapted to networked/web environments, while they are not originally designed to fit into such high-risk environments. For example, in the case of Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided. As a result, integrating security into software becomes a very challenging and interesting domain of research.

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

On the other hand, securing software is a difficult procedure. If it is done manually, it requires high security expertise and lot of time to be tackled. It may also create other vulnerabilities. Moreover, there is always a difficulty in finding the software engineers or 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 companies' managers. 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. Aspect-oriented programming (AOP) is an appealing approach that allows the separation of crosscutting concerns. This paradigm seems to be very promising to integrate security into software. The process of merging the security concerns into the original program is called weaving.

This paper provides new contributions towards developing practical and formal framework for systematic security hardening of software. The initial approach [11, 12] is composed of the following components: Security Hardening Language (*SHL*), plans, patterns and their equivalent aspects. Their combination allows the developers to perform systematic security hardening of software by applying well-defined solutions and without the need to have expertise in the security solution domain. However, the current AOP technologies lack several features needed for security hardening concerns. Accordingly, the approach still has some limitations due to the fact that it uses the current AOP languages (e.g. AspectC++, AspectJ) to weave the security components into the code. We are addressing these shortcomings by elaborating a new approach that allows to apply the hardening on the *Gimple* representation (tree) of software and avoid in some cases the use of the current AOP

technologies. *Gimple* is an intermediate representation of a program. It is a language-independent and a tree-based representation generated by GNU Compiler Collection (*GCC*) during compilation. We propose in this paper novel weaving capabilities for *Gimple* to be integrated into the *GCC* compiler. These features allow to compile the security hardening pattern and inject it into the *Gimple* tree of a program during the *GCC* compilation procedures. Beside, exploiting *Gimple* intermediate representation enables to advise an application written in a specific language with code written in a different one.

Moreover, we elaborate *Gimple* partial syntax, *SHL* syntax, and first steps towards formal semantics for *Gimple* weaving. These first steps constitute an initial attempt and a guide toward developing a complete weaver for *Gimple*. We also demonstrate the feasibility of our propositions by providing the implementation methodology and results of the proposed semantics into *GCC*. This is followed by a case study for securing the connections of client applications, where the hardening is applied and compiled using our extended *GCC*.

The remainder of this paper is organized as follows. In Section 2, we review the contributions in the field of AOP, AOP for securing software, and AOP weaving semantics. Afterwards, in Section 3, we summarize our initial approach for systematic security hardening and illustrate the new proposition where weaving is performed on the *Gimple* representation of a software by adopting an aspect-oriented style. In Section 4, we present the syntax of *SHL* and *Gimple* and provide the first steps towards operational semantics for *Gimple* weaving. After that, we explain briefly in Section 5 the implementation methodology and results of the *Gimple* weaving capabilities in the *GCC* compiler. Finally, we illustrate in Section 6 a security hardening case study and offer in Section 7 concluding remarks.

2. Background and Related Work

We present in the sequel an overview of the current literature on the approaches related to the contribution of this paper.

2.1. Aspect-oriented Programming

AOP depends on the principle of "Separation of Concerns", where issues that crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed. However, they are programming language-dependent and cannot be used to specify abstract security hardening patterns, which is a requirement in the defined approach. AspectJ [7] built on top of the Java programming language and AspectC++ [15] built on top of the C/C++ programming languages are the most prominent ones. The approach, which is adopted by most of the AOP languages, is called the pointcut-advice model. The fundamental concepts of this model are: join points, pointcuts, and advices.

Each atomic unit of code to be injected is called an advice. It is necessary to formulate where to inject the advice into the program. This is done by the use of a pointcut expression, which its matching criteria restricts the set of the join points of a program for which the advice will be injected. A join point is a principled point in the execution of a program. The pointcut expressions typically allow to pick out function calls, function executions, join points on the control flow ulterior to a given join point, etc. At the heart of this model, is the concept of an aspect, which embodies all these elements. Examples of implemented aspects are presented in Section 6. 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.

2.2. AOP Approaches for Security Injection

Most of the contributions [2, 4, 6, 14] that explore the usability of AOP for integrating security code into applications are presented as case studies that show the relevance of AOP languages for application security. They have focused on exploring the usefulness of AOP for securing software by security experts who know exactly where each piece of code should be manually injected. None of them have proposed an approach or methodology for systematic security hardening with features similar to our proposition. We present in the following an overview on these contributions.

Cigital labs has proposed an AOP language called CSAW [14], which is a small superset of C programming language dedicated to improve the security of C programs. De Win, in his Ph.D. thesis [4], has discussed an aspect-oriented approach that allows the integration of security aspects within applications. It uses AOP concepts to specify the behavior code to be merged in the application and the location where this code should be injected. In [2], Ron Bodkin has surveyed the security requirements for enterprise applications and described examples of security crosscutting concerns, with a focus on authentication and authorization. Another contribution in AOP and security is the Java Security Aspect Library (JSAL), in which Huang et al. [6] has introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. Masuhara and Kawauchi [8] have defined the dataflow pointcut, which identifies join points based on the origin of values.

2.3. Semantics for AOP Weaving

The main related work that addresses AOP weaving semantics is presented in this subsection. None of them have 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. [16] where the authors have defined the semantics of MinAML, an aspect-oriented language. They have used labels to mark points where advices are going to be injected. Advices are applied to the argument or to the result of a function.

Tatsuzawa et al. [9] 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.

Wang et.al. [17] have provided seamless integration of AOP paradigm and strongly-typed functional languages paradigm through a static weaving process, which deals with around advices and type-scoped pointcuts in the presence of higher-order functions.

It is noticeable that all the previous contributions target AOP with functional programming. As a new idea, a name-based calculus μABC [3] has been introduced in which aspects are the primitive computational entity. 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.

3. *Gimple* Weaving Approach for Hardening

This section summarizes the whole approach for systematic security hardening and presents an extension to it based on *Gimple* weaving that is needed to achieve our objectives.

Software security hardening has been defined in [10] as *any process, methodology, product or combination thereof that is used to add security functionalities and/or remove vulnerabilities or prevent their exploitation in existing software*. Security hardening practices are usually applied manually by injecting security code into the software [1, 5, 13]. This task entails that the software engineers have deep knowledge of the inner working of the software code, which is not available all the time. In this context, we have elaborated an aspect-oriented approach to perform security hardening in a systematic way. The whole approach architecture is illustrated in Figure 1.

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 security solution domain. At the same time, the security hardening should be applied in an organized and systematic way in order not to alter the original functionalities of a software. This is done by providing an abstraction over the actions required to improve the security of a 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 an elaborated language called Security Hardening Language (*SHL*). We define security hardening patterns as well-defined solutions to known security problems, together with detailed information on how and where to inject each component of the solution into application.

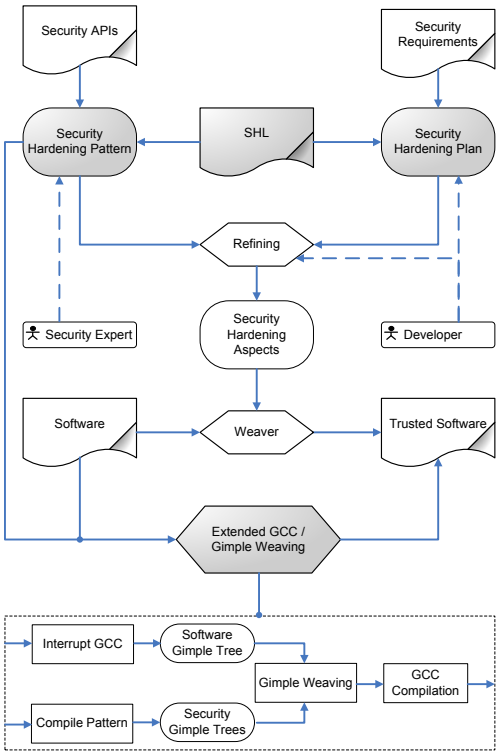

Figure 1. Approach Architecture

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

In the original approach, once the security hardening solutions are built, the refinement of the solutions into aspects or low level code can be performed by programmers that do not need to have any security expertise. Afterwards, an AOP weaver (e.g. AspectJ, AspectC++) can be executed to harden the aspects into the original source code, which at this stage can be inspected for correctness. However, this task still requires human interaction to provide some parameters needed for the concrete implementation, and it is somehow hard to elaborate an automatic translator to perform it. Moreover, the current AOP technologies lack some features needed for security hardening.

In this paper, we first provide 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. This may also provide more systematization and automation to the approach, since the refinement process is performed manually by programmers. Moreover, 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 hardening.

This is done by passing the *SHL* patterns and the original software 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* in Figure 2) 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* specified into each *Behavior* of the pattern. Finally, the resulted *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 [18] 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 software. In this context, we provide in this paper the syntax of *SHL* and *Gimple*, together with the first steps towards 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 the implementation of the weaving capabilities for *Gimple*. Moreover, it may allow to verify the correctness of applying security hardening patterns into applications.

Beside the fact that the contributions presented in this paper improve significantly the approach for systematic security hardening, it also constitutes by itself 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.

We have 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. We have also applied these security hardening solutions on large scale software (e.g. mysql and APT) and several applications. In the sequel, we present a case study showing first the use of AOP (AspectC++) to secure the connections of an application implemented in C, then exploring its equivalence using 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.

4. Towards Formal Weaving Semantics

In this this section, we present the syntax of *SHL* and part of the syntax of *Gimple* that serves our goals. Besides the first steps towards a formal semantics are provided. This semantics describes how to inject security-related code at specific locations in the *Gimple* representation of programs. In the beginning, we need to define the notations that are used along this section.

Notations

- Given two sets *A* and *B*, we will write $A \underset{m}{\longrightarrow} B$ to denote the set of all mappings from *A* to *B*. A mapping (map for short) $m \in A \underset{m}{\longrightarrow} B$ could be defined by extension as $[a_1 \mapsto b_1, ..., a_n \mapsto b_n]$ to denote the association of the elements b_i's to a_i's. We will write Dom(m) to denote the domain of the map m. Given two maps m and m', we will write $m \dagger m'$ to denote the overwriting of the map m by the associations of the map m', i.e., the domain of $m \dagger m'$ is Dom(m) $\cup$ Dom(m'), and we have $(m \dagger m')(a) = m'(a)$ if $a \in$ Dom(m') and $m(a)$ otherwise.
- 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 category τ.

- Given a type τ, we write τ-`list` to denote the type of lists having elements of category τ.
- The type *Identifier* classifies identifiers.

4.1. *SHL* and *Gimple* Syntax

In this subsection, we present the syntax of *SHL* and part of the syntax of *Gimple*. We define an environment that is built from a *Gimple* program *Program* and a pattern *Pattern*. *SHL* syntax is presented in Fig. 2 where the main part of it is a pattern. A hardening pattern is based on the pointcut-advice model of AOP. A *Pattern* includes a list of behaviors. Each *Behavior* specifies where *insertionPoint* and what *code* to insert at specific location *location* in addition to a set of primitives *primitive*. Behavior insertion point specifies the point of code insertion after identifying the location. The behavior insertion point can be `before` or `after`. `before` or `after` means keep the old code at the identified location and insert the new code before or after it respectively. *Location* identifies the joint points in the program where behavior code should be applied. The list of constructs used in *Location* has not yet been completed and left 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:

- `call`: picks out the join points where we call a specific function.
- `execution`: picks out the join points referring to the implementation of a specific function.
- `withincode`: 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 base locations *BaseLocation* can be combined using logical operators to produce more complex ones. The code *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, a library, or left to be implemented by the user.

Since *Gimple* contains a lot of constructs, the needed ones to express the weaving semantics are chosen and presented in Fig. 4. 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

```
Environment ::= ⟨program: Program, pattern: Pattern⟩ (Environment)
Pattern ::= Behavior -list (Pattern)
Behavior ::= ⟨insertionPoint: before | after
               location: Location,
               code: Code ⟩ (Behavior)
Location ::= BaseLocation | BooleanLocation (location)
BaseLocation ::= ⟨kind: call | execution | withincode,
                  signature: Fname⟩
               | ⟨kind: set | get,
                  signature: Vname⟩
BooleanLocation ::= Location and Location
                  | Location or Location
                  | not Location
Code ::= ⟨iRetType: integer_type | real_type | boolean_type | void_type,
           iName: Fname ⟩ (Code)
Fname ::= Identifer
Vname ::= Identifer
```

Figure 2. *SHL* **Syntax**

block *Block* represented by `bind_expr` contains the 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` and call statement *CallStmt* represented by `call_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 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

Program	::=	⟨*funs*: *FunDecl* -set,	(**Program**)
		types: *Type* -set,	
		const: *Const* -set ⟩	
FunDecl	::=	⟨*kind*: function_decl,	(**Function**)
		fname: *Fname*,	
		ftype: *FunType*,	
		args: *ParmDecl* -list,	
		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* -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* ⟩	
IndirectRef	::=	⟨*kind*: indirect_ref,	
		op: *VarDecl* ⟩	
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* ⟩	

Figure 3. *Gimple* **Partial Syntax-A**

Const	::=	*Nat*		(**Constant**)
Pname	::=	*Identifer*	*Rname*::= *Identifer*	
Vname	::=	*Identifer*	*Lname*::= *Identifer*	

Figure 4. *Gimple* **Partial Syntax-B**

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 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. Besides, there are two complex types: function type *FuncType* represented by function_type and pointer type *PoniterType* represented by pointer_type. 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.

4.2. Weaving Semantics

In this subsection, we provide the first steps towards an operation semantics for *Gimple* weaving. The utility functions that are used in the weaving rules are defined informally to be defined in the future work. We restrict ourselves describing only the weaving semantics inside a block body of a specific function declaration because this illustrates the weaving process for all the declared functions inside a *Gimple* representation. To facilitate the semantics of weaving, we give numbers in a sequential manner to statements inside a block body of a function declaration according to their order in the list. This is done by creating a map from integers to statements that reflects this order. The map represents the index of a list. The map contains two extra dummy statements ($\langle$*kind* : dumpf_expr$\rangle$, $\langle$*kind* : dumpl_expr$\rangle$) as a first and a last statements to delimit the boundary of a function body. Any changes in the order of statements by weaving will be done in the map and then reflected in the block body but without the dummy statements. We consider just call, set, get, withincode, and execution locations and postpone the other locations for future work. Every base location has a corresponding shadow in the *Gimple* representation. Shadows are ei-

ther a single statement or a bounded region of statements. The call statement is a shadow for the `call` location while the assignment statement is a shadow for the `set` and `get` locations. The shadow that corresponds to a function execution `execution` is the entire method body. Consequently, we need to delimit the beginning and the end of this shadow. For this reason, we introduce the notions of "before shadow" and "after shadow". The call statement and the assignment statement are considered as both after and before shadows. The first statement in a function definition is considered as a before shadow whereas the last statement (dummy) in a function definition is considered as an after shadow. The `withincode` location does not define new shadows by itself but it uses the shadows that are defined by the other pointcuts.

The operational semantics is based on the evolution of configurations that are defined hereafter. The configuration has the following form:

$$\langle \mathcal{E}, fd, map, n, next, pat \rangle$$

where

- $\mathcal{E}$ represents the environment, which constitutes a *Gimple* program and a pattern.
- fd represents the current function declaration.
- map represents the map to consider. The map maps integers to statements. At the beginning, the map is generated from the block body of the function declaration fd by calling the function createMap with the block body (which is a list of statements) and the length of this block body as parameters.

  $$\mathsf{createMap}(fd.\mathit{block.body}, \mathsf{listLength}(fd.\mathit{block.body}))$$

  This map is then changed during the weaving process to reflect code injection. The function listLength takes a list and returns the length of it.
- n represents the counter in the mapping map that is mapped to the statement, which is going to be inspected for behavior weaving.
- $next$ represents the counter in the mapping map that is mapped to the next statement to be considered for behavior weaving.
- pat represents the behaviors to consider.

Notice that the initial configuration when considering advice weaving within a function whose declaration is fd is: $\langle \mathcal{E}, fd, \mathsf{createMap}(fd.\mathit{block.body}, \mathsf{listLength}(fd.\mathit{block.body})), 0, 1, \mathcal{E}.\mathit{pattern} \rangle$. The first rule of the semantics describes the case where the current statement is not a shadow or the pattern is empty. If the current statement is not a shadow this means that it is not: a call statement, an assignment statement, a first statement in a function definition, or a last statement in a function definition. If the pattern list is empty this means that all the behaviors have been treated for this statement. In such cases, the current statement is skipped and the list of behaviors is reset to its initial value (all the defined behaviors).

$$\frac{\neg \mathsf{isStmtShadow}(map, n) \vee pat = [\,]}{\langle \mathcal{E}, fd, map, n, next, pat \rangle \rightarrow \langle \mathcal{E}, fd, map, next, next + 1, \mathcal{E}.\mathit{pattern} \rangle}$$

The second rule fires when the current statement is a before shadow but does not match with the before behavior in the head of the pattern list. Accordingly, this behavior is removed from the pattern list and the weaving process continues with the remaining list of behaviors.

$$\frac{\begin{array}{c}\mathsf{isBeforeShadow}(map, n) \quad pat \neq [\,] \\ \mathsf{head}(pat).\mathit{insertionPoint} = \texttt{before} \\ \neg \mathsf{isBeforeApplicable}(fd, map, n, \mathsf{head}(pat).\mathit{location})\end{array}}{\langle \mathcal{E}, fd, map, n, next, pat \rangle \rightarrow \langle \mathcal{E}, fd, map, n, next, \mathsf{tail}(pat) \rangle}$$

The third rule represents the case where the head of the pattern list is a before-behavior and it is applicable to the current statement which represents a before shadow. In this case, the function declaration and the environment are changed because of the before-behavior merging. The function declaration is changed because the body of its block is changed as a result of inserting a call statement to the weaved function before the statement shadow. Consequently, this will be reflected in the map by adding the counter of the statement shadow and all its subsequent statements by one. In the case of the (`dummyf_expr`), the call to the weaved function is inserted after it because this implies inserting it before the first real statement in the function. The environment is changed because its program is changed as a result of the modification in its function declaration. This before-behavior is then removed from the pattern list and the weaving process continues with the remaining list of behaviors.

$$\frac{\begin{array}{c}\mathsf{isBeforeShadow}(map, n) \quad pat \neq [\,] \\ \mathsf{head}(pat).\mathit{insertionPoint} = \texttt{before} \\ \mathsf{isBeforeApplicable}(fd, map, n, \mathsf{head}(pat).\mathit{location}) \\ (\mathcal{E}', fd', map', n', next') = \mathsf{insertBefore}(\mathcal{E}, fd, map, n, next, \mathsf{head}(pat))\end{array}}{\langle \mathcal{E}, fd, map, n, next, pat \rangle \rightarrow \langle \mathcal{E}', fd', map', n', next', \mathsf{tail}(pat) \rangle}$$

The fourth rule fires when the current statement is an after shadow but does not match with the after behavior in the head of the pattern list. Accordingly, this behavior is removed from the pattern list and the weaving process continues with the remaining list of behaviors.

$$\frac{\begin{array}{c}\mathsf{isAfterShadow}(map, n) \quad pat \neq [\,] \\ \mathsf{head}(pat).\mathit{insertionPoint} = \texttt{after} \\ \neg \mathsf{isAfterApplicable}(fd, map, n, \mathsf{head}(pat).\mathit{location})\end{array}}{\langle \mathcal{E}, fd, map, n, next, pat \rangle \rightarrow \langle \mathcal{E}, fd, map, n, next, \mathsf{tail}(pat) \rangle}$$

The fifth rule represents the case where the head of the pattern list is an after-behavior and it is applicable to the current

statement which represents an after shadow. In this case, the function declaration and the environment are changed because of the after-behavior merging. The function declaration is changed because the body of its block is changed as a result of inserting a call statement to the weaved function after the statement shadow. Consequently, this will be reflected in the map by adding the counter for all statements after the statement shadow by one. In the case of the (`dummyl_expr`), the call to the weaved function is inserted before it because this implies inserting it after the last real statement in the function. The environment is changed because its program is changed as a result of the modification in its function declaration. This after-behavior is then removed from the pattern list and the weaving process continues with the remaining list of behaviors.

$$\frac{\begin{array}{c}\mathsf{isAfterShadow}(map, n) \quad pat \neq [\,] \\ \mathsf{head}(pat).\mathit{insertionPoint} = \texttt{after} \\ \mathsf{isAfterApplicable}(fd, map, n, \mathsf{head}(pat).\mathit{location}) \\ (\mathcal{E}', fd', map', n', next') = \mathsf{insertAfter}(\mathcal{E}, fd, map, n, next, \mathsf{head}(pat))\end{array}}{\langle \mathcal{E}, fd, map, n, next, pat\rangle \rightarrow \langle \mathcal{E}', fd', map', n', next', \mathsf{tail}(pat)\rangle}$$

To insert a function call before, after, or instead a statement shadow in *Gimple* trees, the following steps are needed before doing the weaving: 1) Build a result type for the weaved function and add it to the defined types in the program. 2) Build a function type for the weaved function and add it to the defined types in the program. 3) Build a function declaration for the weaved function and add it to the declared functions in the program. 4) Build a pointer type for the weaved function and add it to the defined types in the program. 5) Build an address expression for the weaved function. 6) Build a call statement to the weaved function based on the address expression. 7) Insert the call statement before, after, or instead the statement shadow.

All the utility functions used in the semantics are described informally as follows:

- The function createMap takes a statement list and an integer. It returns a mapping that maps integers to statements to reflect the order of statements in a function definition.
- The function listLength takes a list and returns its length.
- isStmtShadow takes a map and an integer. It returns true if the statement mapped to the integer in the map is a call statement, an assignment statement, a first statement in a function definition, or a last statement in a function definition.
- isBeforeShadow takes a map and an integer. It returns true if the statement mapped to the integer in the map is a call statement, an assignment statement, or a first statement in a function definition.
- isAfterShadow takes a map and an integer. It returns true if the statement mapped to the integer in the map is a call statement, an assignment statement, or a last statement in a function definition.
- head takes a list and returns its first element.
- tail takes a list and removes its first element.
- isBeforeApplicable takes a function definition, a map, an integer, and a location in a before behavior, It returns true if the location match the current statement mapped to the integer in the map. Regarding matching, the call statement matches a `call` location if the location signature equals the name of the called function. The assignment statement matches a `get` or a `set` location if the location signature equals the name of the used variable or the defined variable respectively. An assignment or a call statement matches a `withincode` location if the location signature equals the name of the function where the corresponding statement exists. An `execution` location in a before behavior matches the first statement in a function body represented by $\langle$*kind* : `dumpf_expr`$\rangle$ while An `execution` location in an after behavior matches the last statement in a function body represented by $\langle$*kind* : `dumpl_expr`$\rangle$
- isAfterApplicable takes a function definition, a map, an integer, and a location in an after behavior, It returns true if the location match the current statement mapped to the integer in the map.
- insertBefore takes an environment, a function definition, a map, two integers, and a behavior. It returns an environment, a function definition, a map, and two integers. It inserts a call statement to the weaved function before the statement shadow. In the case of the (`dummyf_expr`), the call to the weaved function is inserted after it because this implies inserting it before the first real statement in a function body.
- insertAfter takes an environment, a function definition, a map, two integers, and a behavior. It returns an environment, a function definition, a map, and two integers. It inserts a call statement to the weaved function after the statement shadow. In the case of the (`dummyl_expr`), the call to the weaved function is inserted before it because this implies inserting it after the last real statement in a function body.

5. Implementation of *Gimple* Weaving Capabilities into *GCC*

The proposed semantics for *Gimple* weaving have been implemented into the *GCC* compiler, which is able now

to apply several hardening practices on the *Gimple* representation (tree) of a program before generating the corresponding executable. The work on the implementation of the whole weaving features is still in progress. 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 4.1 for more details on *SHL* syntax). Additional components have been developed in order to parse the pattern and gather the information (e.g. function name, return type, etc.) from the *Behavior*(s) and pass them as parameters to specific functions provided by *GCC* that are responsible of building *Gimple* trees (e.g. build_decl(...)). Afterwards, each generated tree is injected in the program tree depending on the *insertionPoint* and *location* specified in each *Behavior*. Besides, new components have been elaborated to gather the required information from the pattern parser and call the *GCC* functions responsible of modifying the *Gimple* tree to inject the new generated ones at specific points (e.g. before the call to the function Hello). Once this weaving procedure is done, *GCC* takes over and continues the classical compilation of the modified tree to generate the executable of the hardened program.

6. 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. 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. Then, we have applied our initial approach for hardening, where we have 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.

6.1. Hardening Pattern for Securing the Connections of Client Application

An important issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission, or session hijacking. In this section, we illustrate our elaborated solutions for securing the connections of client applications by following the approach methodology and using the proposed *SHL* language. 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.

Listing 1 presents the pattern elaborated in *SHL* for securing the connection of the aforementioned application using GnuTLS/SSL. The code of the functions used in the *Code* of the pattern's *Behavior*(s) is illustrated in Listing 2. It is expressed in C/C++ because the application is implemented in this programming language. However, other syntax and programming languages can also be used depending on the abstraction required and the implementation language of the application to harden. To generalize our solution and make it applicable on wider range of applications, we assume that not all the connections are secured, since many programs have different local interprocess communications via sockets. In this case, all the functions responsible of sending/receiving data on the secure channels are replaced by the ones providing TLS. On the other hand, the other functions that operate on the non-secure channels are kept untouched. Moreover, we suppose that the connection processes and the functions that send and receive data are implemented in different components. This required additional effort to develop additional components that distinguish between the functions that operate on secure and non secure channels and export parameters between different places in the applications.

Listing 1. *SHL* Hardening Pattern for Securing Connection

```
BeginPattern

Before Execution <main> //Starting Point BeginBehavior
    // Initialize the TLS library
    InitializeTLSLibrary;
EndBehavior

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

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

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

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

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

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

EndPattern
```

Listing 2. Functions Used in Secure Connection Pattern

```
  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();
```

6.2. Hardening the Source Code using AspectC++

We have refined and implemented (using AspectC++) in Listing 3 the corresponding aspect of the pattern that is presented in Listing 1. The reader will notice the appearance of `hardening_sockinfo_t`. These are the data structures and functions that we have developed to distinguish between secure and non secure channels. Besides, they are used to export the parameters between the application's components at runtime (since the primitives `ImportParamter` and `ExportParameter` have not yet deployed into the weavers). Parameter passing between functions that initialize the connection and those that use it for sending and receiving data is a major problem that we have faced. In order to avoid using shared memory directly, we have 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 have stored is whether the socket is secure or not. In this manner, all calls to a `send()` and `recv()` are modified for a runtime check that uses the proper sending/receiving function.

Listing 3. Excerpt of Aspect for Securing Connections

```
aspect SecureConnection {

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

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

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

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

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

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

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

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

```
}
};
```

6.3. Hardening the *Gimple* Representation of Code

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 in Listing 1 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 compile the hardening pattern for securing connection. The pattern is compiled *Behavior* by *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 *Behavior*. Afterwards, *GCC* continues its classical compilation on the modified tree and generates finally the executable of the hardened client application. The resulted application is able now to connect securely using HTTPS.

6.4. 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 5 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). 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 in Listing 3 with the different variants 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 capture. Due to this and to avoid duplication, we present in Figure 6 only the run of the application

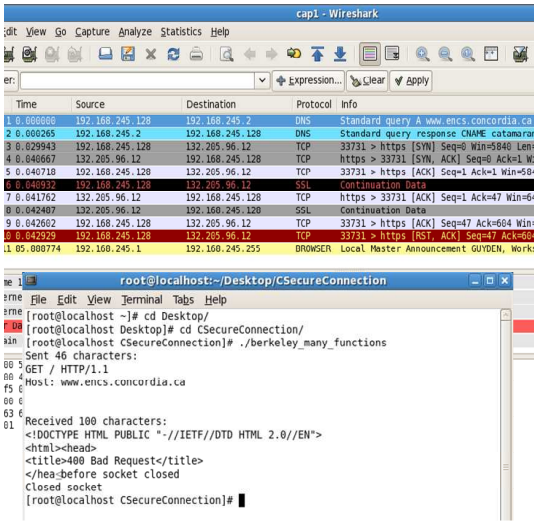

Figure 5. Capture of Connection

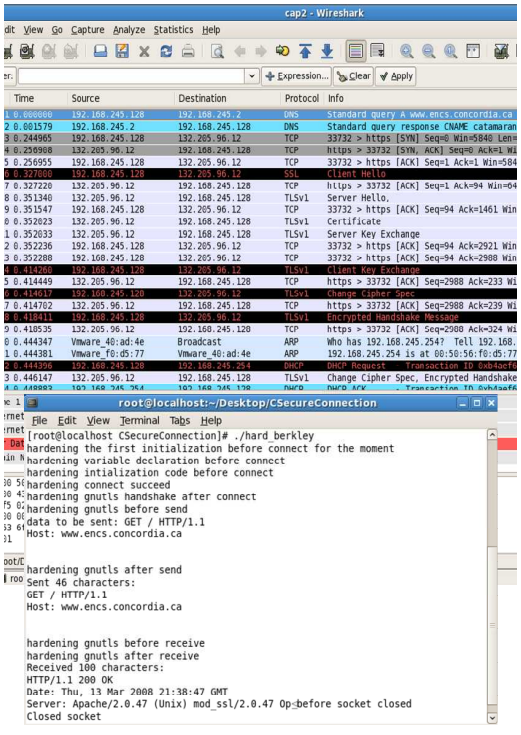

Figure 6. Capture of Hardened Connection

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

7. Conclusion

We have presented in this paper our accomplishment towards developing a formal and practical framework for systematic security hardening. This framework, which is based on AOP and *Gimple* weaving, illustrates the propositions and methods that allow developers to perform the security hardening of software in a systematic way and without the need to have expertise in the security solution domain. At the same time, it allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them. This is done by adopting an approach that provides an abstraction over the actions required to improve the security of programs and building *Gimple-GCC* weaving features for security hardening. Moreover, we have elaborated the syntax of *SHL* and *Gimple* and the first steps towards operational semantics for *Gimple* weaving together with its implementation methodology. We have also explored the feasibility and relevance of our propositions into practical implementation and security hardening case studies.

Regarding our future work, we are currently working on extending the *Gimple* weaving semantics and completing its implementation into *GCC*.

References

[1] M. Bishop. How Attackers Break Programs, and How to Write More Secure Programs, 2005. `http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html` (accessed 2007/04/19).

[2] R. Bodkin. Enterprise security aspects. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.

[3] G. Bruns, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In *Proc. Concur*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, 2004.

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

[5] M. Howard and D. E. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[6] M. Huang, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Overview of aspectj. In *Proceedings of the 15th European Conference ECOOP 2001, Budapest, Hungary*. Springer Verlag, 2001.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 105–121, 2003.

[9] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 320–330, New York, NY, USA, 2005. ACM.

[10] A. Mourad, M.-A. Laverdière, and M. Debbabi. Security hardening of open source software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM Press, 2006.

[11] A. Mourad, M.-A. Laverdière, and M. Debbabi. A high-level aspect-oriented based language for software security hardening. In *Proceedings of the International Conference on Security and Cryptography*. Secrypt, 2007.

[12] A. Mourad, M.-A. Laverdière, and M. Debbabi. Towards an aspect oriented approach for the security hardening of code. In *Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems*. IEEE Press, 2007.

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

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

[15] O. Spinczyk, A. Gal, and W. chroder Preikschat. Aspectc++: An aspect-oriented extension to c++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia*, 2002.

[16] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2003.

[17] M. Wang, K. Chen, and S.-C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 78–87, New York, NY, USA, 2006. ACM.

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