

Common Weaving Approach in Mainstream Languages for Software Security Hardening^{*}

Dima Alhadidi¹, Azzam Mourad², Hakim Idrissi Kaitouni¹, and Mourad Debbabi¹

¹ Computer Security Laboratory (CSL)
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, Quebec, Canada
{dm_alhad,h_idri,debbabi}@encs.concordia.ca

² Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon
azzam.mourad@lau.edu.lb

Abstract. In this paper, we propose a novel aspect-oriented approach based on GIMPLE, a language-independent and a tree-based representation generated by the GNU Compiler Collection (GCC), for the systemization of application security hardening. The security solutions are woven into GIMPLE representations in a systematic way, eliminating the need for manual hardening that might generate a considerable number of errors. To achieve this goal, we present a formal specification for GIMPLE weaving and the implementation strategies of the proposed weaving semantics. Syntax for a common aspect-oriented language that is abstract and multi-language support together with syntax for a core set for GIMPLE constructs are presented to express the weaving semantics. GIMPLE weaving accompanied by a common aspect-oriented language (1) allows security experts providing security solutions using this common language, (2) lets developers focus on the main functionality of programs by relieving them from the burden of security issues, (3) unifies the matching and the weaving processes for mainstream languages, and (4) facilitates introducing new security features in AOP languages. We handle the correctness and the completeness of GIMPLE weaving in two different ways. In the first approach, we prove them according to the rules and algorithms provided in this paper. In the second approach, we accommodate Kniesel's discipline that ensures that security solutions specified by our approach are applied at all and only the required points in source code, taking into consideration weaving interactions and interferences. Finally, we explore the viability and the relevance of our propositions by applying the defined approach for systematic security hardening to develop case studies.

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

Keywords: Application Security Hardening, Aspect-oriented Programming (AOP), GIMPLE, Weaving, Operational Semantics, Security Hardening Patterns.

1 Introduction

Security takes an increasingly predominant role in today's computing world. The computer industry faces challenges in public confidence as vulnerabilities are discovered, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. The challenge is even greater when such systems must be adapted to networked/web environments, when they were not originally designed to fit into such high-risk environments. In some cases, and whenever the source code is available, as is the case for Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided [23,25,35,61]. As a result, integrating security into software is a very challenging and interesting domain of research.

Securing software is a difficult and a critical procedure. If done manually, it may require a lot of time and create other vulnerabilities. In fact, one should find all the code involved and change it consistently everywhere. Moreover, it is always difficult to find software engineers or developers who are specialized in both security solutions and software functionality. This is an open problem raised by managers of different companies [2]. One way to overcome these difficulties is by separating security concerns from the rest of the application so they can be addressed independently and applied globally. A methodology that would encompass separation of security concerns and consistent implementation of security solutions would pave the road towards secure applications, enable a security expert to enforce security properties, and facilitate the correctness verification of security solutions.

In this respect, Aspect-oriented Programming (AOP) [45] is an appealing approach that allows the separation of crosscutting concerns. There are many AOP languages developed that are language-dependent. We distinguish from them AspectJ [46], built on top of the Java programming language, AspectC [31], built on top of the C programming language, AspectC++ [67] built on top of the C++ programming language, AspectC# [47], built on top of the C# programming language, and the AOP version, addressed for the Smalltalk programming language [29]. The approach adopted by most of these languages is the pointcut-advice model [54]. The fundamental concepts of this model are: join points, pointcuts, and advice pieces. A join point is a location in the execution of a program. A pointcut is a concept that classifies join points in the same way a type classifies values. Advice is a code fragment that is executed when join points satisfying a particular pointcut are reached. This execution can be done before, after, or around a specific join point. Finally, advice is composed and merged with the core functionality modules into one single program. The process of composition is called weaving, and the tools that perform such a process are called weavers.

More recently, several proposals have been advanced for code injection via AOP into source code for improving its security [28,33,42,63]. However, AOP has not been initially engineered with security in mind. Consequently, new security-related constructs and primitives have been proposed as AOP extensions, e.g., pointcuts for dataflow [53], integer overflow [19], loop [40], local variable setting and getting [18], and synchronized block [18]. Extending each AOP language with new security features addresses time waste and effort repetition. In this paper, we present new contributions towards developing a practical and a formal framework for systematic security hardening of software. The defined approach allows application of the hardening on the GIMPLE representation of software [14]. GIMPLE is an intermediate representation of programs. It is a language-independent and a tree-based representation generated by the GNU Compiler Collection (GCC) [8] during compilation. GCC is a compiler system supporting various programming languages, e.g., C, C++, Objective-C, Fortran, Java, and Ada. In transforming the source code to GIMPLE, complex expressions are split into three-address forms using temporary variables.

Exploiting the intermediate representation of GIMPLE enables us to define common weaving semantics that supports the mainstream languages and facilitates introducing new security-related AOP extensions. The importance of this stems from the fact that aspect-oriented languages are language-dependent. Accordingly, GIMPLE weaving allows defining common weaving semantics and implementation for all programming languages supported by the GCC compiler instead of defining them for each AOP language. For example, instead of having a specific compiler for every aspect-oriented programming language that tries to match join points in code and then does the weaving, the matching and the weaving are done on GIMPLE trees without focusing on a specific programming language.

The defined approach in this paper is based on the Security Hardening Language (SHL) that is defined in [57,58]. This language is expressive, human-readable, and intermediate between English and programming languages. Moreover, it is aspect-oriented and supports multiple languages. The authors [57,58] have performed systematic security hardening of software by applying well-defined solutions that are defined by security experts using SHL. Afterward, developers refine these security solutions manually into one of the existing AOP languages. To bypass this manual refinement and to facilitate future extensions for new security features in AOP languages, the proposed approach in this paper allows applying the hardening on the GIMPLE representation of software. The main contributions of this paper can be summarized as follows:

- Semantics and algorithms for matching and weaving in GIMPLE are formalized. For this reason, syntax for a common aspect-oriented language that is abstract and multi-language support and syntax for GIMPLE constructs are defined. Providing a formal framework for GIMPLE weaving helps to better understand the underpinnings of this approach and capture its steps in a rigorous way. This can serve both as a reference for programmers wishing to understand subtle points about the GIMPLE weaving and as a touchstone

for implementers by providing a standard against which the correctness of an implementation may be judged. Additionally, it establishes the theoretical properties of the approach and provides a foundation for mathematical proofs.

- Correctness and completeness of GIMPLE weaving are explored from two different views. In the first approach, we address them according to the provided formal matching and weaving rules and the defined algorithms in this paper. On the other hand, we accommodate in the second approach Kniesel’s discipline to prove that GIMPLE weaving is correct and complete only in some specific cases because of behavior interactions and interferences. This ensures that security code is injected in all specified locations and only at these specified locations. Accordingly, security properties that are represented by this code are satisfied after weaving.
- Implementation strategies of the proposed semantics are introduced. To explore the viability and the relevance of the defined approach, case studies are developed. Case studies of small programs are developed to solve the problems of unsafe creation of temporary files and use of deprecated functions. These problems are inspired from CERT coding rules [4,5,12] and U.S. Department of Homeland Security coding rules [15] where they are representations of knowledge gained from real-world experiences about potential vulnerabilities that exist in programming languages. For scalability reasons, we target large-size software (*gzip* version 1.2.4) for monitoring purposes.

The remainder of this paper is organized as follows. Section 2 provides an overview of the GCC compiler. In Section 3 we summarize the new proposition where weaving is performed on the GIMPLE representation of software by adopting an aspect-oriented style. In Section 4 we present SHL syntax and GIMPLE syntax together with the weaving semantics for matching and weaving in GIMPLE trees. We analyze the correctness and the completeness of the GIMPLE weaving in Section 5. We explain in Section 6 the implementation strategies of the GIMPLE weaving capabilities in the GCC compiler. We present in Section 7 security hardening case studies for validation and illustration purposes. The paper discusses the related work in Section 8. Concluding remarks as well as a discussion of future work are represented in Section 9.

2 Overview of GCC

GCC is composed from three components: a front end, a back end, and a middle end [27,73], as shown in Fig. 1. The compilation of a source file can be viewed as a pipeline that converts one program representation into another. Source code enters the front end and flows through the pipeline, being converted at each stage into successively lower-level representation forms until final code generation in the form of assembly code. There are two intermediate languages that are used on the interfaces between the three “ends” of GCC. The higher level one, used between front end and middle end, is called GENERIC. The lower level one, used between middle end and back end, is called Register Transfer

Language (RTL). Both middle end and back end do various optimizations on their intermediate representation before they turn it into a yet lower-level one. Each language supported by GNU Compiler Collection (GCC) has its specific front end that produces the syntax tree abstraction of a given source code. The meaning of a tree was somewhat different for different language front ends. This was simplified with the introduction of GENERIC and GIMPLE, two new forms of language-independent trees that were introduced with the advent of GCC 4.0.

GENERIC provides a language-independent way of representing an entire function in trees. For optimization purposes, GENERIC is still a too high-level representation. At the GENERIC level, many language-specific structures are preserved, e.g., loops. During the compilation, it is lowered into GIMPLE. The process of lowering is called *gimplification*. In transforming code to GIMPLE, complex expressions are split into a three-address code using temporary variables. GIMPLE is a subset of GENERIC and it is the common language for a large number of new powerful language- and architecture-independent global (function scope) optimizations. Our approach in this paper is based on GIMPLE, a language-independent representation. Accordingly, the defined weaving approach in this paper is multi-language support. Whatever the programming language used for software development, we can do the weaving as long as this programming language is supported by the GCC compiler. These languages currently include C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada.

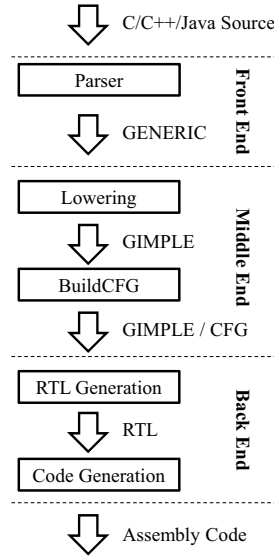


Fig. 1. GCC Architecture [73]

3 GIMPLE Weaving Approach for Security Hardening

Software security hardening [56] is defined as *any process, methodology, product, or combination that is used to add security functionalities, remove vulnerabilities, or prevent their exploitation in existing software*. Security hardening practices are usually manually applied by injecting security code into software [26, 41, 62]. In this paper, we propose a novel and an aspect-oriented approach based on the GIMPLE representation for the systemization of application security hardening. The security solutions are woven into GIMPLE representations in a systematic way eliminating the need for manual hardening that may generate a considerable number of errors. GIMPLE [14] is a language-independent and an intermediate representation generated by the GCC [8] compiler. It is based on a tree data structure. In GIMPLE, expressions are broken down into a three-address form, using temporary variables to hold intermediate values.

The proposed approach in this paper is based on the Security Hardening Language (SHL) that is defined in [57, 58]. The authors have elaborated an aspect-oriented approach to perform security hardening in a systematic way. In their approach, security experts provide security solutions using an abstract and a general aspect-oriented language called SHL that is expressive, human-readable, multi-language support and intermediate between English and programming languages. This relieves developers from the burden of security issues and allows them to focus on the main functionality of programs. The approach provides an abstraction over the actions that are required to improve the security of programs and adopts an aspect-oriented approach to build and develop the required security solutions.

SHL is built on top of the current AOP technologies that are based on the pointcut-advice model [54]. The solutions elaborated in SHL are expressed by plans and patterns and can be refined into a selected AOP language. Security hardening patterns are high-level and well-defined solutions to known security problems, together with detailed information on how and where to inject each component of the solution into an application. A pattern includes a list of behaviors, each of which specifies the insertion point and the code to be inserted at a specific location. Security hardening plans instantiate security hardening patterns with parameters regarding platforms, libraries, and languages. The combination of hardening plans and patterns constitutes a bridge that allows security experts to provide the best solutions to particular security problems and allows developers to use these solutions to harden applications by developing security hardening patterns. The development implies manual refinement of solutions into one of the existing AOP languages (e.g., AspectJ, AspectC++).

Fig. 2 illustrates the architecture of the GIMPLE weaving approach presented in this paper together with the architecture presented in [57, 58]. The GIMPLE weaving approach bypasses the refinement step from patterns into AOP languages. This provides more systematization and automation because the refinement process in [57, 58] is performed manually by programmers. The hardening tasks specified in patterns are abstract and support multiple languages, which makes the GIMPLE representation a relevant target to apply the

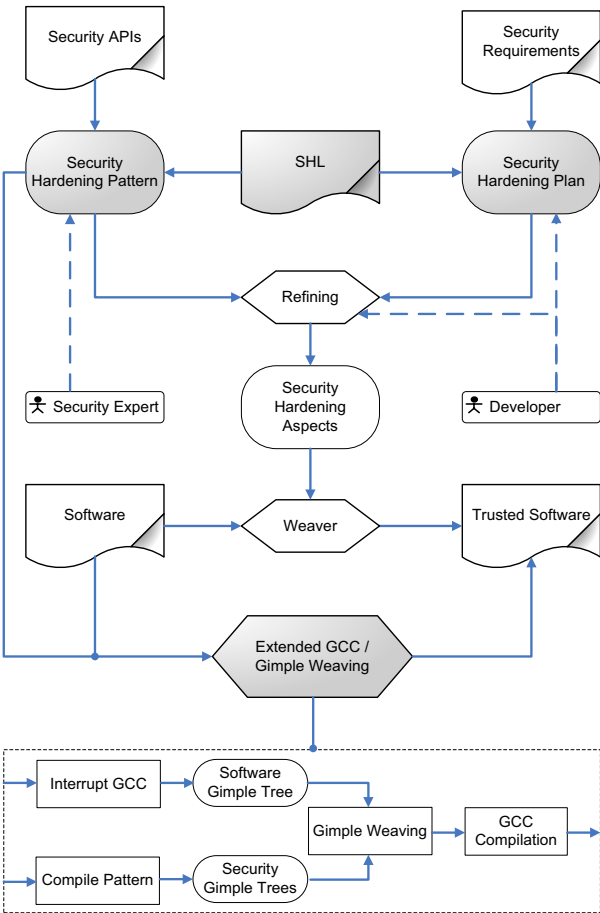


Fig. 2. Approach Architecture

hardening. This is done by passing the SHL patterns and the original software to an extended version of the GCC compiler that in the end generates the executable of the trusted software. For this purpose, an additional pass is added to the GCC compiler in order to interrupt the compilation once the GIMPLE representation of code is completed. In parallel, the hardening pattern is compiled and a GIMPLE tree is built for each behavior using the routines of the GCC compiler that are provided for this purpose. Afterward, the GIMPLE trees generated from the hardening patterns are integrated in the GIMPLE tree of the original code with respect to the location(s) specified in each behavior of the hardening pattern. Finally, the resultant GIMPLE tree is passed again to the GCC compiler in order to continue the regular compilation process and produce the executable of the secure software. The extended GCC was originally implemented by our colleagues [76] for the purposes of dynamic vulnerability detection. We modify it in order to inject the security functionalities specified in the hardening patterns.

In the following, we present the formal specification of weaving an SHL pattern into the GIMPLE representation. In this context, we present GIMPLE and SHL syntax with matching and weaving semantics in GIMPLE. Providing such semantics leads to the implementation of the GIMPLE weaving capabilities. Moreover, it allows us establishing results regarding the correctness and the completeness of weaving security hardening patterns into the GIMPLE representation of applications.

4 Formal Weaving Specification

In this section, we present SHL and GIMPLE syntax together with the corresponding matching and weaving semantics. The GIMPLE syntax covers the required constructs for the weaving semantics. The weaving semantics describes how to inject security-related code at specific locations in the GIMPLE representation of programs. First, we need to introduce the following notation.

Notation

- The algorithms and notations are written with respect to the OCaml notations [10].
- Given a record space $D = \langle f_1 : D_1, f_2 : D_2, \dots, f_n : D_n \rangle$ 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 $\tau\text{-set}$ to denote the type of sets having elements of type τ .
- Given a type τ , we write $\tau\text{-list}$ to denote the type of lists having elements of type τ .
- The type *Identifier* classifies identifiers.

4.1 SHL and GIMPLE Syntax

In this subsection, we present SHL and GIMPLE syntax. We define an environment as the combination of a GIMPLE program and a SHL pattern. The syntax

<i>Environment</i>	::= <program: <i>Program</i> , pattern: <i>Pattern</i> >	(Environment)
<i>Pattern</i>	::= <i>Behavior</i> -list	(Pattern)
<i>Behavior</i>	::= <insertionPoint: before after replace , location: <i>Location</i> , code: <i>Code</i> >	(Behavior)
<i>Location</i>	::= <i>BaseLocation</i> <i>BooleanLocation</i>	(location)
<i>BaseLocation</i>	::= true <kind: call execution withincode , signature: <i>Fname</i> > <kind: set get , signature: <i>Vname</i> > <kind: cflow , signature: <i>Location</i> >	
<i>BooleanLocation</i>	::= <i>Location</i> and <i>Location</i> <i>Location</i> or <i>Location</i> not <i>Location</i>	
<i>Code</i>	::= <iRetType: integer_type real_type boolean_type void_type , iName: <i>Fname</i> >	(Code)
<i>Fname</i>	::= <i>Identifier</i>	
<i>Vname</i>	::= <i>Identifier</i>	

Fig. 3. SHL Syntax

of SHL is presented in Fig. 3, where the main part is a pattern. A hardening pattern is based on the pointcut-advice model of AOP. A pattern includes a list of behaviors, each of which specifies the insertion point and the code to be inserted at a specific location. The insertion point of a behavior specifies where the code should be injected according to a specific location. It can have the following three values: **before**, **after** or **replace**. The value **replace** means remove the code at the identified location and replace it with the new code, while the values **before** and **after** mean keep the old code at the identified location and insert the new code before and after it, respectively. A location identifies joint points in the program where behavior code should be applied. In the following, we present a list of constructs used in location. This list can be extended depending on the need for security hardening solutions.

- **call**: picks out the join points where we call a specific function.
- **execution**: picks out the join points that refer to the entire code segment of a specific function.
- **withincode**: picks out the join points within a specific function.
- **set**: picks out the join points where a variable is set.
- **get**: picks out the join points where a variable is gotten.
- **cflow**: picks out the join points occurring in the dynamic execution context of the join points captured by a specific location.

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

The GIMPLE syntax presented in Fig. 4 and Fig. 5 is based on the so-called rough GIMPLE grammar [13]. A GIMPLE program consists of the following main parts: a set of function declarations, a set of types, and a set of constants. A function declaration specifies the function name, the function type, the argument declarations, the result declaration, and the function block. The function 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 of a block.

There are a variety of statements in GIMPLE. Most are assignment statements represented by `modify_expr`, call statements represented by `call_expr`, and conditional statements represented by `cond_expr`. The modify statement has two parts: the left-hand side and the right-hand side. The left-hand side can be a variable declaration, a parameter declaration, or an indirect reference, whereas the right-hand side can be one of the kinds of the left-hand side statements, a constant, a call statement, a unary statement, a binary statement, a relational statement, or an address expression. We refer to the kind of unary statements as `unary_expr` to represent `!`, `~`, etc., whereas we refer to the kind of binary statements as `binary_expr` to represent `+`, `-`, `*`, etc. Similarly, we refer to the kind of relational statements as `rel_expr` to represent `<`, `>`, etc. 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 the 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 and the function arguments. The conditional statement has three parts: the condition and two statement lists. The condition can be either a constant, a variable declaration, a parameter declaration, or a relational statement.

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`. In addition there are two complex types: function type, represented by `function_type`, and pointer type, represented by `pointer_type`. Pointer type can specify a function type, which in 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, represented by `parm_decl`; variable declaration, represented by `var_decl`; result declaration, represented by `result_decl`; and label declaration, represented by `label_decl`. Finally, constants are represented by natural numbers.

<i>Program</i>	::=	{ <i>fun</i> : <i>FunDecl-set</i> , <i>types</i> : <i>Type-set</i> , <i>const</i> : <i>Const-set</i> }	(Program)
<i>FunDecl</i>	::=	{ <i>kind</i> : function_decl , <i>fname</i> : <i>Fname</i> , <i>ftype</i> : <i>FunType</i> , <i>args</i> : <i>ParmDecl-list</i> , <i>result</i> : <i>ResDecl</i> , <i>block</i> : <i>Block</i> }	(Function)
<i>Block</i>	::=	{ <i>ekind</i> : bind_expr , <i>decl</i> : <i>VLDecl-set</i> , <i>body</i> : <i>Stmt-list</i> }	(Block)
<i>Stmt</i>	::=	<i>ModifyStmt</i> <i>CallStmt</i> <i>IfStmt</i> <i>Block</i>	(Statement)
<i>ModifyStmt</i>	::=	{ <i>kind</i> : modify_expr , <i>lhs</i> : <i>Lhs</i> , <i>rhs</i> : <i>Rhs</i> }	(Assignment)
<i>CallStmt</i>	::=	{ <i>kind</i> : call_expr , <i>addrExpr</i> : <i>AddrExpr</i> , <i>arglist</i> : <i>VPDecl -list</i> }	(Function Call)
<i>IfStmt</i>	::=	{ <i>kind</i> : cond_expr , <i>condition</i> : <i>Condition</i> , <i>op1</i> : <i>Stmt-list</i> , <i>op2</i> : <i>Stmt-list</i> }	(If)
<i>Lhs</i>	::=	<i>ParmDecl</i> <i>VarDecl</i> <i>IndirectRef</i>	
<i>Rhs</i>	::=	<i>Const</i> <i>Lhs</i> <i>CallStmt</i> <i>UnStmt</i> <i>BinStmt</i> <i>RelStmt</i> <i>AddrStmt</i>	
<i>UnStmt</i>	::=	{ <i>kind</i> : unary_expr , <i>op</i> : <i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>AddrStmt</i> }	
<i>BinStmt</i>	::=	{ <i>kind</i> : binary_expr , <i>op1</i> : <i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>AddrStmt</i> , <i>op2</i> : <i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>AddrStmt</i> }	
<i>AddrExpr</i>	::=	{ <i>kind</i> : addr_expr , <i>type</i> : <i>PointerType</i> , <i>opp</i> : <i>VarDecl</i> <i>FunDecl</i> }	
<i>IndirectRef</i>	::=	{ <i>kind</i> : indirect_ref , <i>opp</i> : <i>VarDecl</i> }	
<i>Condition</i>	::=	<i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>RelStmt</i>	

Fig. 4. GIMPLE Syntax - Part 1

<i>RelStmt</i>	::=	<kind:	<i>rel_expr</i> ,	
		<op ₁ :	<i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>AddrStmt</i> ,	
		op ₂ :	<i>Const</i> <i>ParmDecl</i> <i>VarDecl</i> <i>AddrStmt</i> >	
<i>Type</i>	::=	<i>integer_type</i> <i>real_type</i> <i>boolean_type</i> <i>void_type</i>		(Type)
		<i>PointerType</i> <i>FunType</i>		
<i>PointerType</i>	::=	<kind:	<i>pointer_type</i> ,	
		type:	<i>FunType</i> <i>integer_type</i> <i>real_type</i> >	
<i>FuncType</i>	::=	<kind:	<i>function_type</i> ,	
		type:	<i>integer_type</i> <i>real_type</i>	
			<i>boolean_type</i> <i>void_type</i> >	
<i>VLDecl</i>	::=	<i>LabelDecl</i> <i>VarDecl</i>		(Declaration)
<i>VPDecl</i>	::=	<i>ParmDecl</i> <i>VarDecl</i>		
<i>ParmDecl</i>	::=	<kind:	<i>parm_decl</i> ,	
		name:	<i>Identifer</i> ,	
		type:	<i>integer_type</i> <i>real_type</i>	
			<i>boolean_type</i> <i>void_type</i> >	
<i>ResDecl</i>	::=	<kind:	<i>result_decl</i> ,	
		name:	<i>Identifer</i> ,	
		type:	<i>integer_type</i> <i>real_type</i>	
			<i>boolean_type</i> <i>void_type</i> >	
<i>VarDecl</i>	::=	<kind:	<i>var_decl</i> ,	
		name:	<i>Identifer</i> ,	
		type:	<i>integer_type</i> <i>real_type</i>	
			<i>boolean_type</i> <i>void_type</i> >	
<i>LabelDecl</i>	::=	<kind:	<i>label_decl</i> ,	
		name:	<i>Identifer</i> ,	
		type:	<i>VoidType</i> >	
<i>Const</i>	::=	<kind:	<i>cst</i> ,	(Constant)
		op:	<i>Nat</i> >	

Fig. 5. GIMPLE Syntax - Part 2

4.2 Matching and Weaving Semantics

In this subsection, we provide the matching and the weaving semantics. For information about GIMPLE semantics in general, we refer the reader to the contribution of Löding [51].

Matching

We define the judgment $fd, s \vdash_{match} loc$, which is used in the matching rules presented in Fig. 6 to describe that a statement s within the function declaration fd matches the location loc . A statement s can be a call statement cs , a modify statement ms , or either of the two statements cms . The rule (Call) describes the case where the current statement is a call statement, the current location is a call location, and the location signature equals the called function specified in the call statement. In such a case, the call statement matches the call location.

$\frac{loc.kind = call \quad cs.addrExpr.op.fname = loc.signature}{fd, cs \vdash_{match} loc}$	(Call)
$\frac{loc.kind = withincode \quad fd.fname = loc.signature}{fd, cms \vdash_{match} loc}$	(Withincode)
$\frac{loc.kind = set \quad ms.lhs.kind = var_decl \quad ms.lhs.name = loc.signature}{fd, ms \vdash_{match} loc}$	(Set)
$\frac{loc.kind = get \quad ms.rhs.kind = var_decl \quad ms.rhs.name = loc.signature}{fd, ms \vdash_{match} loc}$	(Get.1)
$\frac{loc.kind = get \quad ms.rhs.op.kind = var_decl \quad ms.rhs.op.name = loc.signature}{fd, ms \vdash_{match} loc}$	(Get.2)
$\frac{loc.kind = get \quad ms.rhs.op_1.kind = var_decl \quad ms.rhs.op_1.name = loc.signature}{fd, ms \vdash_{match} loc}$	(Get.3)
$\frac{loc.kind = get \quad ms.rhs.op_2.kind = var_decl \quad ms.rhs.op_2.name = loc.signature}{fd, ms \vdash_{match} loc}$	(Get.4)
$\frac{fd, s \vdash_{match} loc_1 \quad fd, s \vdash_{match} loc_2}{fd, s \vdash_{match} loc_1 \text{ and } loc_2}$	(And)
$\frac{fd, s \vdash_{match} loc_1}{fd, s \vdash_{match} loc_1 \text{ or } loc_2}$	(Or.1)
$\frac{fd, s \vdash_{match} loc_2}{fd, s \vdash_{match} loc_1 \text{ or } loc_2}$	(Or.2)
$\frac{fd, s \not\vdash_{match} loc}{fd, s \vdash_{match} \text{not } loc}$	(Not)

Fig. 6. Matching Rules

The rule (Withincode) describes the case where the current statement is a call statement or an assignment statement, the current location is a withincode location, and the location signature equals the name of the function where the call statement or the assignment statement exists. In such a case, the call statement or the assignment statement matches the withincode location.

The rule (Set) describes the case where the current statement is an assignment statement, the current location is a set location, and the location signature equals the name of the variable being set. In such a case, the assignment statement matches the set location.

The rule (Get_1) describes the case where the current statement is an assignment statement, the current location is a get location, and the location signature equals the name of the variable being get. In such a case, the assignment statement matches the get location. The rule (Get_2) describes the case where the current statement is an assignment statement, the current location is a get location, and the location signature equals the name of the variable being get by a unary operation. In such a case, the assignment statement matches the get location. The rule (Get_3) describes the case where the current statement is an assignment statement, the current location is a get location, and the location signature equals the name of the first variable being get by a binary operation. In such a case, the assignment statement matches the get location. The rule (Get_4) describes the case where the current statement is an assignment statement, the current location is a get location, and the location signature equals the name of the second variable being get by a binary operation. In such a case, the assignment statement matches the get location. The rules (And), (Or_1), (Or_2), and (Not) describe cases in which locations are combined using logical operators to produce more complex ones.

Weaving

To accomplish the weaving, we should create a call statement from a given behavior to be woven in the specified locations. The rule (CreateWvStmt) presented in Fig. 7 describes how to do such a creation where t is a type, fd , fd' , bfd are function declarations, ft is a function type, pt is a pointer type, ae is an address expression, beh is a behavior, and \mathcal{E} is an environment. The call statement represents a call to the function specified in the behavior code. In addition, it represents the statement that is going to be woven into the statements that match the location of the behavior. The environment is changed as a result of such a creation. We define the judgment $\mathcal{E}, beh \vdash_{build} \mathcal{E}', cs$ to represent that the call statement cs is built based on the behavior beh . The environment \mathcal{E}' reflects a modified version of the environment \mathcal{E} after creating the call statement cs . The following steps are required to create a call statement from a given behavior:

- Build a result type for the weaved function and add it to the defined types in the program.
- Build a function type for the weaved function and add it to the defined types in the program.
- Build a function declaration for the weaved function and add it to the declared functions in the program.
- Build a pointer type for the weaved function and add it to the defined types in the program.
- Build an address expression for the weaved function.
- Build a call statement to the weaved function based on the address expression.
- Modify the environment to include all the new defined types and all the new defined constants together with the new function declaration.

$$\begin{array}{c}
t = \text{buildRetType}(\text{beh.code}) \quad ft = \text{buildFunType}(t) \quad bfd = \text{buildFunDecl}(\text{beh.code}, ft) \\
pt = \text{buildFunPtr}(ft) \quad ae = \text{buildAddrExpr}(ft, pfd) \quad cs = \text{buildCallStmt}(ae) \\
\mathcal{E}'.\text{program.types} = \mathcal{E}.\text{program.types} \cup t \cup ft \cup pt \quad \mathcal{E}'.\text{program.funs} = \mathcal{E}.\text{program.funs} \cup bfd \\
\mathcal{E}'.\text{program.const} = \mathcal{E}.\text{program.const} \cup \text{extractConst}(cs.\text{AddrExpr.op.block.body}) \\
\hline
\mathcal{E}, \text{beh} \vdash_{\text{build}} \mathcal{E}', cs \\
\hline
(\text{CreateWvStmt})
\end{array}$$

Fig. 7. Weaved Statement Creation

$$\begin{array}{c}
\begin{array}{l}
fd.\text{block.body} = l@(s :: l') \quad fd, s \vdash_{\text{match}} \text{beh.location} \quad \mathcal{E}, \text{beh} \vdash_{\text{build}} \mathcal{E}', cs \\
\text{beh.insertionPoint} = \text{before} \quad fd' = \{fd \text{ with } \text{block.body} = l@(cs :: (s :: l'))\} \\
funSet = \mathcal{E}''.\text{program.funs} \quad \mathcal{E}' = \{\mathcal{E}'' \text{ with } \text{program.funs} = (funSet \setminus \{fd\}) \cup \{fd'\}\} \\
\hline
\langle \mathcal{E}, fd, \text{beh} :: \text{behL}', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', \text{behL}', s, \text{weaving} \rangle
\end{array} \quad (\text{Before}) \\
\\
\begin{array}{l}
fd.\text{block.body} = l@(s :: l') \quad fd, s \vdash_{\text{match}} \text{beh.location} \quad \mathcal{E}, \text{beh} \vdash_{\text{build}} \mathcal{E}', cs \\
\text{beh.insertionPoint} = \text{after} \quad fd' = \{fd \text{ with } \text{block.body} = l@(s :: (cs :: l'))\} \\
funSet = \mathcal{E}''.\text{program.funs} \quad \mathcal{E}' = \{\mathcal{E}'' \text{ with } \text{program.funs} = (funSet \setminus \{fd\}) \cup \{fd'\}\} \\
\hline
\langle \mathcal{E}, fd, \text{beh} :: \text{behL}', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', \text{behL}', s, \text{weaving} \rangle
\end{array} \quad (\text{After}) \\
\\
\begin{array}{l}
fd.\text{block.body} = l@(s :: l') \quad fd, s \vdash_{\text{match}} \text{beh.location} \quad \mathcal{E}, \text{beh} \vdash_{\text{build}} \mathcal{E}', cs \\
\text{beh.insertionPoint} = \text{replace} \quad fd' = \{fd \text{ with } \text{block.body} = l@(cs :: l')\} \\
funSet = \mathcal{E}''.\text{program.funs} \quad \mathcal{E}' = \{\mathcal{E}'' \text{ with } \text{program.funs} = (funSet \setminus \{fd\}) \cup \{fd'\}\} \\
\hline
\langle \mathcal{E}, fd, \text{beh} :: \text{behL}', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', \text{behL}', s, \text{weaving} \rangle
\end{array} \quad (\text{Replace}) \\
\\
\begin{array}{l}
fd.\text{block.body} = l@(s :: l') \quad fd, s \vdash_{\text{match}} \text{not } \text{beh.location} \\
\hline
\langle \mathcal{E}, fd, \text{beh} :: \text{behL}', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, \text{behL}', s, \text{weaving} \rangle
\end{array} \quad (\text{NoMatch}) \\
\\
\begin{array}{l}
\text{behL} = [] \\
\hline
\langle \mathcal{E}, fd, \text{behL}, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \text{end} \rangle
\end{array} \quad (\text{End}_1) \\
\\
\begin{array}{l}
fd.\text{block.body} = [] \\
\hline
\langle \mathcal{E}, fd, \text{behL}, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \text{end} \rangle
\end{array} \quad (\text{End}_2)
\end{array}$$

Fig. 8. Weaving Rules

In the weaving rules presented in Fig. 8, the weaving process is represented by the weaving configuration $\langle \text{Environment}, \text{FunDecl}, \text{Behavior-list}, \text{Stmt}, \text{State} \rangle$. The state *State* is a flag that represents the stage of the behavior weaving process, which is either **weaving** or **end**. The flag is equal to **weaving** when applicable behaviors still have to be woven, whereas it becomes **end** when the weaving is completed. Hence, the transformation $\langle \mathcal{E}, fd, \text{behL}, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', \text{behL}', s', \text{end} \rangle$ means that the environment \mathcal{E}' and the function decla-

ration fd' is the result of weaving all the applicable behaviors in the behavior list $behL$ into the statement s . The rule (Before) describes the case where a before location matches a specific statement. The call statement created from the behavior that contains the before location is inserted before the matched statement. The rule (After) describes the case where an after location matches a specific statement. The call statement created from the behavior that contains the after location is inserted after the matched statement. The rule (Replace) describes the case where a replace location matches a specific statement. The call statement created from the behavior that contains the replace location supersedes the matched statement. Additionally, it is required to propagate the weaving changes that are done to a specific function to higher levels in each weaving rule, i.e., the program and the environment where the corresponding function exists should be changed accordingly. The rule (NoMatch) describes the case where the location does not match a specific statement. In this case, the weaving process continues with the rest of the behaviors. Finally, the rule (End_1) and the rule (End_2) represent the cases where there are no more behaviors to check for weaving and where the body of the function block contains no statements, respectively. In these cases, the weaving process is halted for the corresponding function declaration.

4.3 Utility Functions

In this subsection, we define all the utility functions that are used to express the weaving semantics.

- The function `buildRetType` takes a behavior code and returns a type.
 $buildRetType : Code \rightarrow Type$
 $buildRetType(c)=t$ where

$$\begin{cases} t = integer_type & \text{if } c.iRetType = integer_type; \\ t = real_type & \text{if } c.iRetType = real_type; \\ t = boolean_type & \text{if } c.iRetType = boolean_type; \\ t = void_type & \text{if } c.iRetType = void_type. \end{cases}$$
- The function `buildFunType` takes a type and returns a function type.
 $buildFunType : Type \rightarrow FuncType$
 $buildFunType(t)=ft$ where $(ft.kind = function_type) \wedge (ft.type = t)$
- The function `buildCallStmt` takes an address expression and returns a call statement.
 $buildCallStmt : AddrExpr \rightarrow CallStmt$
 $buildCallStmt(ae)=cs$ where $(cs.kind = call_expr) \wedge (cs.addrExpr = ae)$

- The function `buildFunDecl` takes a behavior code and a function type. It returns a function declaration.

`buildFunDecl : Code \times FuncType \rightarrow FuncDecl`

`buildFunDecl(c, ft) = fd` where $(fd.kind = \text{function_decl}) \wedge (fd.fname = c.iName) \wedge (fd.ftype = ft)$

- The function `extractConst` takes a statement list and returns a constant set.

`extractConst : Stmt-list \rightarrow Const-set`

`extractConst(l) =`

$$\left\{ \begin{array}{l} \text{extractConst}(s.addrExpr.opp.block.body) \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{call_expr}; \\ \\ \text{extractConst}(s.op_1) \cup \text{extractConst}(s.op_2) \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{cond_expr}; \\ \\ s.rhs \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \wedge s.rhs.kind = \text{cst}; \\ \\ \text{extractConst}(s.AddrExpr.opp.block.body) \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \wedge s.rhs.kind = \text{call_expr}; \\ \\ \text{extractConst}(s.rhs.addrExpr.opp.block.body) \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \wedge s.rhs.kind = \text{addr_expr}; \\ \\ s.rhs.op \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \wedge s.rhs.kind = \text{unary_expr} \\ \quad \wedge s.rhs.op.kind = \text{cst}; \\ \\ s.rhs.op_1 \cup s.rhs.op_2 \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \\ \quad \wedge (s.rhs.kind = \text{binary_expr} \vee s.rhs.kind = \text{rel_expr}) \\ \quad \wedge s.rhs.op_1.kind = \text{cst} \wedge s.rhs.op_2.kind = \text{cst}; \\ \\ s.rhs.op_1 \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \\ \quad \wedge (s.rhs.kind = \text{binary_expr} \vee s.rhs.kind = \text{rel_expr}) \\ \quad \wedge s.rhs.op_1.kind = \text{cst}; \\ \\ s.rhs.op_2 \cup \text{extractConst}(l') \\ \quad \text{if } l = s :: l' \wedge s.kind = \text{modify_expr} \\ \quad \wedge (s.rhs.kind = \text{binary_expr} \vee s.rhs.kind = \text{rel_expr}) \\ \quad \wedge s.rhs.op_2.kind = \text{cst}. \end{array} \right.$$

- The function `buildFunPtr` takes a function type and returns a pointer type.

`buildFunPtr : FuncType \rightarrow PointerType`

`buildFunPtr(ft) = pt` where $(pt.kind = \text{pointer_type}) \wedge (pt.type = ft)$

- The function `buildAddrExpr` takes a pointer type and a function declaration. It returns an address expression.
 $\text{buildAddrExpr} : \text{PointerType} \times \text{FunDecl} \rightarrow \text{AddrExpr}$
 $\text{buildAddrExpr}(pt, fd) = ae$ where $(ae.kind = \text{addr_expr}) \wedge (ae.type = pt) \wedge (ae.opp = fd)$

5 Completeness and Correctness of GIMPLE Weaving

In this section we address the completeness and the correctness of GIMPLE weaving from two different views. In the first approach we present algorithms that implement the semantics reported in the rules in Fig. 6, Fig. 7, and Fig. 8. Then, we prove the correctness and the completeness of GIMPLE weaving algorithms with respect to the weaving semantics. In the second approach, we adapt Kniesel's discipline [50] wherein a language-independent correctness and completeness analysis is discussed.

5.1 First Approach

We have three algorithms: \mathcal{WS} in Fig. 9, \mathcal{M} in Fig. 10, and \mathcal{W} in Fig. 11. The first, \mathcal{WS} , is meant to create a call statement from a given behavior. It takes as arguments an environment \mathcal{E} and a behavior beh and yields a modified environment \mathcal{E}' and a call statement cs . The second algorithm, \mathcal{M} , is meant for matching. It takes three arguments: a function declaration fd , a location loc , and a statement s . It returns true if the statement s matches the location loc , and it returns false otherwise. The third algorithm, \mathcal{W} , implements the weaving semantics reported in Fig. 8. It takes four arguments: an environment \mathcal{E} , a function declaration fd , a behavior list $behL$, and a statement s . It yields an environment \mathcal{E}' and a function declaration fd' . The environment \mathcal{E}' and the function declaration fd' reflect a modified version of the environment \mathcal{E} and the function declaration fd , respectively, after the weaving process. In the following, we state and prove results that establish the soundness and completeness of the algorithms \mathcal{WS} , \mathcal{M} , and \mathcal{W} with respect to the semantics reported in Fig. 6, Fig. 7, and Fig. 8. The following lemma states the soundness of the algorithm \mathcal{WS} :

Lemma 1. (Soundness of \mathcal{WS}). *Given an environment \mathcal{E} and a behavior beh , if $\mathcal{WS}(\mathcal{E}, beh) = (\mathcal{E}', cs)$ then $\mathcal{E}, beh \vdash_{build} \mathcal{E}', cs$.*

The proof of Lemma 1 is straightforward since the algorithm \mathcal{WS} results from the rule (CreateWvStmt) presented in Fig. 7.

The following lemma states the completeness of the algorithm \mathcal{WS} :

Lemma 2. (Completeness of \mathcal{WS}). *Given an environment \mathcal{E} and a behavior beh , if $\mathcal{E}, beh \vdash_{build} \mathcal{E}', cs$ then $\mathcal{WS}(\mathcal{E}, beh) = (\mathcal{E}', cs)$.*

```

 $\mathcal{WS}(\mathcal{E}, beh) =$ 
  let  $t = \text{buildRetType}(beh.code)$ 
   $ft = \text{buildFunType}(t)$ 
   $bfd = \text{buildFunDecl}(beh.code, ft)$ 
   $pt = \text{buildFunPtr}(ft)$ 
   $ae = \text{buildAddrExpr}(pt, bfd)$ 
   $cs = \text{buildCallStmt}(ae)$ 
   $tt = \mathcal{E}.program.types$ 
   $fn = \mathcal{E}.program.funs$ 
   $cc = \mathcal{E}.program.const$ 
   $pp = \mathcal{E}.pattern$ 
   $\mathcal{E}' = \{pattern = pp; program.types = tt \cup t \cup ft \cup pt; program.funs = fn \cup bfd;$ 
     $program.const = cc \cup \text{extractConst}(cs.AddrExpr.op.block.body)\}$ 
  in  $(\mathcal{E}', cs)$ 

```

Fig. 9. Weaved Statement Creation Algorithm

```

 $\mathcal{M}(fd, loc, s) = \text{case } (loc.kind, s.kind) \text{ of}$ 
  (call, call_expr)  $\Rightarrow s.addrExpr.op.fname = loc.signature$ 
  (withincode, call_expr)  $\Rightarrow fd.fname = loc.signature$ 
  (withincode, modify_expr)  $\Rightarrow fd.fname = loc.signature$ 
  (set, modify_expr)  $\Rightarrow s.lhs.kind = \text{var\_decl} \ \&\&$ 
     $s.lhs.name = loc.signature$ 
  (get, modify_expr)  $\Rightarrow (s.rhs.kind = \text{var\_decl} \ \&\&$ 
     $s.rhs.name = loc.signature)$ 
    ||  $(s.rhs.op.kind = \text{var\_decl} \ \&\&$ 
     $s.rhs.op.name = loc.signature)$ 
    ||  $(s.rhs.op_1.kind = \text{var\_decl} \ \&\&$ 
     $s.rhs.op_1.name = loc.signature)$ 
    ||  $(s.rhs.op_2.kind = \text{var\_decl} \ \&\&$ 
     $s.rhs.op_2.name = loc.signature)$ 

```

Fig. 10. Matching Algorithm

The proof of Lemma 2 is straightforward since the algorithm \mathcal{WS} results from the rule (CreateWvStmt) presented in Fig. 7.

The following lemma states the soundness of the matching algorithm \mathcal{M} :

Lemma 3. (Soundness of \mathcal{M}). *Given a function declaration fd , a location loc , and a statement s , if $\mathcal{M}(fd, loc, s)$ then $fd, s \vdash_{match} loc$.*

To establish this Lemma, we proceed by case analysis:

Proof. – **Case (call):**

From the algorithm, we have:

$loc.kind = \text{call}$
 $s.kind = \text{call_expr}$
 $s.addrExpr.op.fname = loc.signature$

Since $s.kind = \text{call_expr}$ then s is *cs* statement.

By the rule (Call):

$fd, cs \vdash_{match} loc$

– **Case (withincode):**

From the algorithm \mathcal{M} , we have two possible scenarios for this case. In the first one, we have:

$loc.kind = \text{withincode}$
 $s.kind = \text{call_expr}$
 $fd.fname = loc.signature$

Since $s.kind = \text{call_expr}$ then s is *cs* statement.

By the rule (Withincode):

$fd, cms \vdash_{match} loc$

In the second scenario, we have:

$loc.kind = \text{withincode}$
 $s.kind = \text{modify_expr}$
 $fd.fname = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is *ms* statement.

By the rule (Withincode):

$fd, cms \vdash_{match} loc$

– **Case (set):**

From the algorithm \mathcal{M} , we have:

$loc.kind = \text{set}$
 $s.kind = \text{modify_expr}$
 $s.lhs.kind = \text{var_decl}$
 $s.lhs.name = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is *ms* statement.

By the rule (Set):

$fd, ms \vdash_{match} loc$

– **Case (get):**

From the algorithm \mathcal{M} , we have four possible scenarios for this case. In the first one, we have:

$loc.kind = \text{get}$
 $s.kind = \text{modify_expr}$
 $s.rhs.kind = \text{var_decl}$
 $s.rhs.name = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is *ms* statement.

By the rule (Get₁):

$fd, ms \vdash_{match} loc$

In the second scenario, we have:

$loc.kind = \text{get}$
 $s.kind = \text{modify_expr}$
 $s.rhs.op.kind = \text{var_decl}$
 $s.rhs.op.name = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is ms statement.

By the rule (Get₂):

$fd, ms \vdash_{match} loc$

In the third scenario, we have:

$loc.kind = \text{get}$
 $s.kind = \text{modify_expr}$
 $s.rhs.op_1.kind = \text{var_decl}$
 $s.rhs.op_1.name = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is ms statement.

By the rule (Get₃):

$fd, ms \vdash_{match} loc$

In the fourth scenario, we have:

$loc.kind = \text{get}$
 $s.kind = \text{modify_expr}$
 $s.rhs.op_2.kind = \text{var_decl}$
 $s.rhs.op_2.name = loc.signature$

Since $s.kind = \text{modify_expr}$ then s is ms statement.

By the rule (Get₄):

$fd, ms \vdash_{match} loc$

□

The following lemma states the completeness of the matching algorithm \mathcal{M} .

Lemma 4. (Completeness of \mathcal{M}). *Given a function declaration fd , a location loc , and a statement s , if $fd, s \vdash_{match} loc$ then $\mathcal{M}(fd, loc, s)$.*

Proof. The proof of Lemma 4 is straightforward by propagating the matching rules presented in Fig. 6 from conclusion to premises. Let us take as example the following case:

– **Case (call):**

From the rule (Call):

$loc.kind = \text{call}$
 $cs.addrExpr.op.fname = loc.signature$

Since s is a cs statement, then $s.kind = \text{call_expr}$

By the algorithm \mathcal{M} :

$\mathcal{M}(fd, loc, s)$

□

The following theorem states the soundness of the weaving algorithm \mathcal{W} .

Theorem 1. (Soundness of \mathcal{W}). *Given an environment \mathcal{E} , a function declaration fd , a behavior list $behL$, and a statement s , if $\mathcal{W}(\mathcal{E}, fd, behL, s) = (\mathcal{E}''', fd''')$ then $\langle \mathcal{E}, fd, behL, s, weaving \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', end \rangle$.*

```

 $\mathcal{W}(\mathcal{E}, fd, behL, s) = \text{case } behL \text{ of}$ 
   $beh :: behL' \Rightarrow$  if  $fd.block.body = []$ 
    then  $(\mathcal{E}, fd)$ 
  else
    if  $beh.insertionPoint = \text{before}$  and  $\mathcal{M}(fd, beh.location, s)$  and
       $fd.block.body = l@(s :: l')$ 
    then
      let  $(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$ 
       $funSet = \mathcal{E}'' . program.funs$ 
       $fd' = \{fd \text{ with } block.body = l@(cs :: (s :: l'))\}$ 
       $\mathcal{E}' = \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\}$ 
      in  $\mathcal{W}(\mathcal{E}', fd', behL', s)$ 
    else
      if  $beh.insertionPoint = \text{after}$  and  $\mathcal{M}(fd, beh.location, s)$  and
         $fd.block.body = l@(s :: l')$ 
      then
        let  $(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$ 
         $funSet = \mathcal{E}'' . program.funs$ 
         $fd'' = \{fd \text{ with } block.body = l@(s :: (cs :: l'))\}$ 
         $\mathcal{E}' = \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd''\}\}$ 
        in  $\mathcal{W}(\mathcal{E}', fd'', behL', s)$ 
      else
        if  $beh.insertionPoint = \text{replace}$  and  $\mathcal{M}(fd, beh.location, s)$  and
           $fd.block.body = l@(s :: l')$ 
        then
          let  $(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$ 
           $funSet = \mathcal{E}'' . program.funs$ 
           $fd' = \{fd \text{ with } block.body = l@(cs :: l')\}$ 
           $\mathcal{E}' = \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\}$ 
          in  $\mathcal{W}(\mathcal{E}', fd', behL', cs)$ 
        else  $\mathcal{W}(\mathcal{E}, fd, behL', s)$ 

   $[] \Rightarrow (\mathcal{E}, fd)$ 

```

Fig. 11. Weaving Algorithm

Proof. By the algorithm \mathcal{W} , if $fd.block.body = []$, we have:

$$\mathcal{W}(\mathcal{E}, fd, behL, s) = (\mathcal{E}, fd)$$

From the rule (End₂), we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \text{end} \rangle$$

The rest of the proof is done by induction over the length of $behL$.

1. Induction basis ($behL = []$):

By the algorithm \mathcal{W} , we have:

$$\mathcal{W}(\mathcal{E}, fd, [], s) = (\mathcal{E}, fd)$$

From the algorithm \mathcal{W} , we conclude that $behL = []$.

From the rule (**End1**), we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \text{end} \rangle$$

2. Induction step:

We assume as induction hypothesis:

If $\mathcal{W}(\mathcal{E}, fd, behL', s) = (\mathcal{E}''', fd''', [])$ then $\langle \mathcal{E}, fd, behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$.

Now, let us consider $(behL = beh :: behL')$:

Since $beh.location$ can be a:

– **Case (before-location):**

From the algorithm \mathcal{W} , we have:

$$\begin{aligned} beh.insertionPoint &= \text{before} \\ \mathcal{M}(fd, beh.location, s) \\ fd.block.body &= l@(s :: l') \\ (\mathcal{E}'', cs) &= \mathcal{WS}(\mathcal{E}, beh) \\ funSet &= \mathcal{E}'' . program.funs \\ fd' &= \{fd \text{ with } block.body = l@(cs :: (s :: l'))\} \\ \mathcal{E}' &= \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\} \end{aligned}$$

By the soundness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:

$$\begin{aligned} fd, s &\vdash_{match} loc \\ \mathcal{E}, beh &\vdash_{build} \mathcal{E}'', cs \end{aligned}$$

From the rule (**Before**), we conclude:

$$\langle \mathcal{E}, fd, beh :: behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', behL', s, \text{weaving} \rangle$$

By the hypothesis, we conclude:

$$\langle \mathcal{E}', fd', behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

By the transitivity of \rightarrow , we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

– **Case (after-location):**

From the algorithm \mathcal{W} , we have:

$$\begin{aligned} beh.insertionPoint &= \text{after} \\ \mathcal{M}(fd, beh.location, s) \\ fd.block.body &= l@(s :: l') \\ (\mathcal{E}'', cs) &= \mathcal{WS}(\mathcal{E}, beh) \\ funSet &= \mathcal{E}'' . program.funs \\ fd' &= \{fd \text{ with } block.body = l@(s :: (cs :: l'))\} \\ \mathcal{E}' &= \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\} \end{aligned}$$

By the soundness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:

$$\begin{aligned} fd, s &\vdash_{match} loc \\ \mathcal{E}, beh &\vdash_{build} \mathcal{E}'', cs \end{aligned}$$

From the rule (**After**), we conclude:

$$\langle \mathcal{E}, fd, beh :: behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', behL', s, \text{weaving} \rangle$$

By the hypothesis, we conclude:

$$\langle \mathcal{E}', fd', behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

By the transitivity of \rightarrow , we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

– **Case (replace-location):**

From the algorithm \mathcal{W} , we have:

$beh.insertionPoint = \text{replace}$

$\mathcal{M}(fd, beh.location, s)$

$fd.block.body = l@(s :: l')$

$(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$

$funSet = \mathcal{E}'' . \text{program.funs}$

$fd' = \{fd \text{ with } block.body = l@(s :: l')\}$

$\mathcal{E}' = \{\mathcal{E}'' \text{ with } \text{program.funs} = (funSet \setminus \{fd\}) \cup \{fd'\}\}$

By the soundness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:

$fd, s \vdash_{match} loc$

$\mathcal{E}, beh \vdash_{build} \mathcal{E}'', cs$

From the rule (Replace), we conclude:

$$\langle \mathcal{E}, fd, beh :: behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}', fd', behL', cs, \text{weaving} \rangle$$

By the hypothesis, we conclude:

$$\langle \mathcal{E}', fd', behL', cs, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], cs', \text{end} \rangle$$

By the transitivity of \rightarrow , we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], cs', \text{end} \rangle$$

– **Case (no match):**

From the algorithm \mathcal{W} , we have:

$fd.block.body = l@(s :: l')$

By the soundness and the completeness of the algorithm \mathcal{M} , we conclude:

$fd, s \vdash_{match} \text{not } loc$

From the rule (NoMatch), we conclude:

$$\langle \mathcal{E}, fd, beh :: behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, behL', s, \text{weaving} \rangle$$

By the hypothesis, we conclude:

$$\langle \mathcal{E}, fd, behL', s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

By the transitivity of \rightarrow , we conclude:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \text{end} \rangle$$

□

The following theorem states the completeness of the weaving algorithm \mathcal{W} .

Theorem 2. (Completeness of \mathcal{W}). *Given an environment \mathcal{E} , a function declaration fd , a behavior list $behL$, and a statement s , if $\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}'', fd'', [], s'', \text{end} \rangle$ then $\mathcal{W}(\mathcal{E}, fd, behL, s) = (\mathcal{E}'', fd'')$.*

Proof. By the rule (End_2), we have:

$$\langle \mathcal{E}, fd, behL, s, \text{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \text{end} \rangle$$

From the rule (End_2), we conclude that $fd.block.body = []$.

From the algorithm \mathcal{W} , we conclude:

$$\mathcal{W}(\mathcal{E}, fd, behL, s) = (\mathcal{E}, fd).$$

The rest of the the proof is done by induction over the length of $behL$.

1. Induction basis ($behL = []$):
 By the rule (**End_1**), we have:
 $\langle \mathcal{E}, fd, behL, s, \mathbf{weaving} \rangle \rightarrow \langle \mathcal{E}, fd, [], s, \mathbf{end} \rangle$
 From the rule (**End_1**), we conclude that $behL = []$.
 From the algorithm \mathcal{W} , we conclude:
 $\mathcal{W}(\mathcal{E}, fd, [], s) = (\mathcal{E}, fd)$.
2. Induction step:
 We assume as induction hypothesis:
 If $\langle \mathcal{E}, fd, behL, s, \mathbf{weaving} \rangle \rightarrow \langle \mathcal{E}''', fd''', [], s', \mathbf{end} \rangle$ then $\mathcal{W}(\mathcal{E}, fd, behL, s) = (\mathcal{E}''', fd''')$.
 Now, let us consider ($behL = beh :: behL'$):
 Since $beh.location$ can be a:
 - **Case (before-location):**
 From the the rule (**Before**), we conclude:
 $fd.block.body = l@(s :: l')$
 $fd, s \vdash_{match} beh.location$
 $\mathcal{E}, beh \vdash_{build} \mathcal{E}'', cs$
 $beh.insertionPoint = \mathbf{before}$
 $fd' = \{fd \text{ with } block.body = l@(cs :: (s :: l'))\}$
 $funSet = \mathcal{E}'' \cdot program.funs$
 $\mathcal{E}' = \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\}$
 By the completeness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:
 $\mathcal{M}(fd, beh.location, s)$
 $(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$
 From the algorithm \mathcal{W} , we conclude:
 $\mathcal{W}(\mathcal{E}, fd, beh :: behL', s) = \mathcal{W}(\mathcal{E}', fd', behL', s)$
 By the hypothesis, we conclude:
 $\mathcal{W}(\mathcal{E}', fd', behL', s) = (\mathcal{E}''', fd''')$
 - **Case (after-location):**
 From the the rule (**After**), we conclude:
 $fd.block.body = l@(s :: l')$
 $fd, s \vdash_{match} beh.location$
 $\mathcal{E}, beh \vdash_{build} \mathcal{E}'', cs$
 $beh.insertionPoint = \mathbf{after}$
 $fd' = \{fd \text{ with } block.body = l@(s :: (cs :: l'))\}$
 $funSet = \mathcal{E}'' \cdot program.funs$
 $\mathcal{E}' = \{\mathcal{E}'' \text{ with } program.funs = (funSet \setminus \{fd\}) \cup \{fd'\}\}$
 By the completeness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:
 $\mathcal{M}(fd, beh.location, s)$
 $(\mathcal{E}'', cs) = \mathcal{WS}(\mathcal{E}, beh)$
 From the algorithm \mathcal{W} , we conclude:
 $\mathcal{W}(\mathcal{E}, fd, beh :: behL', s) = \mathcal{W}(\mathcal{E}', fd', behL', s)$
 By the hypothesis, we conclude:

$$\mathcal{W}(\mathcal{E}', fd', behL', s) = (\mathcal{E}''', fd''')$$

– **Case (replace-location):**

From the the rule (Replace), we conclude:

$$\begin{aligned} fd.block.body &= l@(s :: l') \\ fd, s &\vdash_{match} beh.location \\ \mathcal{E}, beh &\vdash_{build} \mathcal{E}'', cs \\ beh.insertionPoint &= \mathbf{replace} \\ fd' &= \{fd \text{ with } block.body = l@(cs :: l')\} \\ funSet &= \mathcal{E}'' \cdot \text{program.funs} \\ \mathcal{E}' &= \{\mathcal{E}'' \text{ with } \text{program.funs} = (funSet \setminus \{fd\}) \cup \{fd'\}\} \end{aligned}$$

By the completeness of the algorithm \mathcal{M} and the algorithm \mathcal{WS} , we conclude:

$$\begin{aligned} \mathcal{M}(fd, beh.location, s) \\ (\mathcal{E}'', cs) &= \mathcal{WS}(\mathcal{E}, beh) \end{aligned}$$

From the algorithm \mathcal{W} , we conclude:

$$\mathcal{W}(\mathcal{E}, fd, beh :: behL', s) = \mathcal{W}(\mathcal{E}', fd', behL', cs)$$

By the hypothesis, we conclude:

$$\mathcal{W}(\mathcal{E}', fd', behL', cs) = (\mathcal{E}''', fd''')$$

– **Case (no Match):**

From the the rule (NoMatch), we conclude:

$$\begin{aligned} fd.block.body &= l@(s :: l') \\ fd, s &\vdash_{match} \mathbf{not} beh.location \end{aligned}$$

By the soundness and the completeness of the algorithm \mathcal{M} , we conclude:

$$\mathbf{not} \mathcal{M}(fd, beh.location, s)$$

From the algorithm \mathcal{W} , we conclude:

$$\mathcal{W}(\mathcal{E}, fd, beh :: behL', s) = \mathcal{W}(\mathcal{E}, fd, behL', s)$$

By the hypothesis, we conclude:

$$\mathcal{W}(\mathcal{E}, fd, behL', s) = (\mathcal{E}''', fd''')$$

□

5.2 Second View

The adopted discipline in this subsection is based on Kniesel's proposal [50]. In the following, we give an overview of this discipline wherein a language-independent correctness and completeness analysis is presented. The discipline of Kniesel does not require programmers to annotate their programs or write semantic specifications of program actions. In addition, it is independent of the aspect language and the program language, and it is independent of the program to which the analyzed aspects are applied. Consequently, it is a perfect fit for GIMPLE weaving. While giving more details to this discipline, we apply the defined notations in this paper to Kniesel's approach, i.e., behavior, location, etc.

Weaving interactions can give rise to interferences that may alter the correctness and the completeness of weaving in general. The completeness constraints state that in the weaving result every behavior code must be applied at *all* the join points matched by the associated location. On the other hand, the correctness constraints state that in the weaving result every behavior code must be applied *only* at the join points matched by the associated location. This is exactly what we need to ensure in our case. Actually, every security property represented with a behavior can be satisfied by applying the corresponding behavior code at all the join points and only the join points matched by the behavior location. Jointly deployed behaviors may interact with each other. If multiple behaviors are used within the same program, correctness or completeness of weaving could be violated by two kinds of weaving interferences:

- Missing behavior code that does not appear where it should have, e.g., behavior *A* could add a method *m* that contains field accesses, and behavior *B* might want to enforce that all field accesses are logged. If the weaving process does not ensure that behavior *B* is executed after behavior *A*, logging code will not be added to the field accesses of *m*.
- Erroneous behavior code that appears at the wrong join points, e.g., behavior *A* code is added at a time when its matching join points have already been removed by a replace behavior *B*.

The definition of correct and complete weaving tells us what is expected from a weaver but not yet how to achieve it. To derive an algorithm that enforces correctness and completeness or otherwise diagnoses the cause of errors according to Kniesel's approach, it is necessary to understand the mechanisms that collaborate in producing errors. *Weaving candidates* are code of behaviors that should be woven jointly. *Selections* generalize the notion of selected join points. An element of a selection is a join point. *Weaving interactions* arise if weaving of a candidate adds, deletes, or modifies program elements, thus modifying the future or past selections of other candidates. The candidate that performs such actions is called the affecting candidate, whereas the one whose selection is changed (extended or reduced) is called the affected candidate. *Weaving interferences* arise if the candidate affected by the weaving interaction is executed before the affecting one. The change of a past selection invalidates the assumption made by the affected candidate. The invalidated assumptions can be positive (it was assumed that a particular program element existed) or negative (it was assumed that an element did not exist). Invalidated assumptions require corrective actions to be performed for the affected candidates. If the selection is extended, correction implies applying their actions at the additional join points. If the selection is reduced, correction undoes actions of the affected candidate at the corresponding join points. *Weaving errors*, i.e., missing code and erroneous code, are caused by weaving interferences that are not repaired. Existing weavers do not support repair. Consequently, it is essential to prevent interferences. This can be achieved by detecting interactions in advance and scheduling the execution of candidates such that the affected ones are always executed after those that affect them.

Kniesel’s approach is based on a representation of programs as logic fact bases and a representation of weaving candidates as conditional transformations (CT) [48, 49] of fact bases. Conditional transformations can be implemented efficiently, as demonstrated by their incarnation in JTransformer [9] and the Conditional Transformation Core (CTC) system [7]. The implementation of the uniformly generic aspect language LogicAJ [60] demonstrates the effectiveness of CTs as a formal model for aspects and a target language for aspect compilation. A CT consists of a precondition and a transformation that is executed on the program elements for which the precondition is true. The assumptions of a CT about the transformed program are captured by its precondition. From the precondition and the transformation, it is possible to derive the CT’s postcondition, which captures the state of the program after the CT execution. A comparison of pre- and postconditions suffices to detect all potential or concrete weaving interactions. Any modification of a CT’s assumptions by the effects of another CT corresponds to a pair of unifiable literals: one in the precondition of the affected CT and one in the postcondition of the affecting CT. If both unifiable literals are positive or if both are negative the affecting CT’s postcondition contributes to making the affected CT’s precondition true on some additional program elements. Thus, it potentially triggers the second one. Otherwise, the violating CT’s postcondition contributes to invalidating the violated CT’s precondition on some program elements. Thus, it potentially inhibits the second one. Accordingly, the following two kinds of weaving interactions can occur among any two jointly woven candidates a and b :

- Triggering: The candidate a triggers or enables b if it adds, removes, or modifies elements such that b ’s predicate additionally succeeds for at least one previously unselected join point.
- Inhibition: The candidate a inhibits or disables b if it adds, removes, or modifies elements such that b ’s predicate becomes false for at least one previously selected join point.

The graph of triggering and inhibition relations provides a precise characterization of the interactions of different weaving candidates. Kniesel shows that acyclic graphs correspond to programs whose interactions can be resolved automatically, preventing interferences. Graphs with cycles define different problem categories depending on the structure of the cycle. The problems range from the need to use weavers that support interference repair (repairable interference), to the need for additional user input (incomplete program), and finally to the impossibility of weaving this set of aspects correctly and completely (conflict).

To ensure correct and complete weaving, it is essential to concentrate on the case of acyclic graphs where interferences can be solved using prevention. The main idea is to statically analyse the candidate set for potential triggering and inhibition relations and to order it such that each candidate is executed after all the candidates whose effects could trigger or inhibit it. The required weaving order is a partial order on the candidates defined as: a before $b \equiv a$ triggers $b \vee a$ inhibits b . Any total order that is consistent with the required weaving order

is guaranteed to ensure correctness and completeness of weaving, as proved by Kniesel.

To adapt this algorithm to GIMPLE weaving, we need to generate the base facts of GIMPLE representations, and we need to translate the aspectual constructs of SHL to CTs. The former issue can be achieved by adapting the algorithm described by Marpons *et al.* [52]. The authors extracted, from GIMPLE representation, program information as Prolog facts that describe structural properties of software. They distinguished between global and local entities. Global entities can appear outside a function or a method definition. For each global entity either declared or defined, a fact is generated stating that this entity exists. Following the identification, a Prolog predicate exists for every relevant property of global entities, and terms are generated for every occurrence of the property. Additionally, relations among global entities exist. Local entities can appear inside a function or a method and are not global entities. A Prolog term is generated for local entities such as local variables and function arguments.

On the other hand, the translation of aspectual constructs of SHL to CTs can be achieved by describing how to convert SHL constructs into CTs. Basic SHL constructs can be converted to LogicAJ [60] constructs, which in turn compile them to CT sets. LogicAJ [60] is a uniformly generic aspect language. It replaces wildcards by explicit meta-variables that can be uniformly used in pointcuts, introductions, and advice as placeholders for (almost) arbitrary base program elements. Thus, it goes beyond type genericity and can express heterogeneous, context-dependent actions. LogicAJ has call, execution, set, get, and withincode pointcuts. SHL constructs can be converted to these pointcuts by specifying the targeted name of a method or a variable. For example, the SHL base location `call f` is converted to the LogicAJ pointcut `call * *.f(..)`.

6 Implementation of GIMPLE Weaving Capabilities into GCC

We implement into the GCC compiler the weaving features inspired by the defined semantics. This implementation allows weaving patterns into the GIMPLE representation of programs before generating the corresponding executables. We handle before, after, and replace behaviors. In addition, we target call, set, get, and withincode locations.

Three input files are needed by the extended compiler to perform the weaving: a source code, a configuration file, and a library containing the subroutines to be woven. To enable our GCC extension to do code weaving, we use command-line options. The implementation methodology that is adopted consists of the following steps: First, we generate a configuration file from the SHL file. The configuration file contains all the information needed for the weaving using our extended GCC. A line in the configuration file is a string composed of five fields delimited by spaces. The structure of this file is shown in Table 1. To generate the configuration file, the specification of SHL is designed and implemented using *ANTLR V3 Beta 6* and its associated *ANTLRWorks 1.1.7* GUI development

Order	Purpose	Value and Meaning
1	Weaving position	0: before 1: after 2: replace
2	Location kind	0: call 1: set 2: get 3: withincode
3	Name of a concerned function or variable	A string of function name or a string of a variable name
4	Name of the function to be woven	A string of a function name
5	Returned type of the weaved function	0: void 1: integer 2: real 3: boolean

Table 1. Configuration File Structure

environment [3]. Using this tool, Java code is generated to parse hardening patterns, to check the correctness of their syntax, and consequently to populate the configuration file. The name of this configuration file is included as a command-line option to be used with an extended version of the GCC compiler version 4.2.0. In order to invoke this extension, the following option should be included at the command line:

`-ftree-security-weaving=weaving_configuration_file`

where `-ftree-security-weaving` signals the extended GCC to enable the weaving functionality and `weaving_configuration_file` is the configuration file. In addition to the above option, the library that contains the code to be woven must be specified. This is done through GCC's options `-l` and `-L`. An example of a complete command that performs the weaving using our extended GCC is:

```
gcc -L \absolute\dir\to\SecLib -l SecLib -ftree-security-weaving=
weaving_configuration_file source_file.c
```

Then, a GIMPLE tree is built for the code of each behavior in a pattern. Afterward, each generated tree is injected in the program tree depending on the insertion point and the location specified in the configuration file. Once this weaving procedure is done, the GCC compiler takes over and continues the classical compilation of the modified tree to generate the executable of the hardened program.

7 Case Studies

In this section, we detail the experiments conducted to explore the correctness of the security hardening process and the feasibility of our propositions. We conduct various experiments over small programs as well as large-sized software, in

particular *gzip* version 1.2.4 (96 files totaling 24,247 lines of code). Case studies of small programs are developed to solve problems: the unsafe creation of temporary files and the use of deprecated functions. These problems are inspired from CERT coding rules [4, 5, 12] and U.S. Department of Homeland Security coding rules [15], where they are representations of knowledge gained from real-world experiences about potential vulnerabilities that exist in programming languages. For scalability, we target large-sized software (*gzip* version 1.2.4) for monitoring purposes. In the following, we detail the experiments.

7.1 Unsafe Temporary File Creation

Temporary files are created for different purposes such as information sharing, temporary data storage, and computation speed. Since temporary files are usually created in shared folders, it is necessary to set appropriate permissions to these files to ensure protection against attackers. The function `umask()` sets the permission modes for newly created files. We should set the `umask` to the most restrictive value possible so the group or world is not allowed to read, write, or execute before creating any new file. The CERT rule that appears in the C Secure Coding Standard and targets this issue is:

FIO43-C: Do not create temporary files in shared directories

This vulnerability exists in well-known packages such as:

- The package *openssh-5.0p1*, which encrypts all traffic to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, *openssh-5.0p1* provides secure tunneling capabilities and several authentication methods.
- The package *shadow-4.1.1*, which contains programs for handling passwords in a secure way.
- The package *patchutils-0.1.5*, which is a small collection of programs that operate on patch files.
- The package *kstart-3.14*, which is a daemon version of *kinit*.
- The package *inn-2.4.6*, which is a full-featured, flexible, and configurable news server.
- The package *binutils-2.19.1*, which is a collection of programming tools for the manipulation of object code in various object file formats.

Fig. 12 presents the pattern elaborated in SHL to solve this problem. The pattern sets the `umask` to the most restrictive value possible using the function `umask (S_IRWXG — S_IRWXO)` before any call for the function `fopen`. The function `umask (S_IRWXG — S_IRWXO)` represents the *Code* in SHL syntax. We inline the code part inside the pattern to facilitate reading. The function `fopen` opens a file for reading, writing, or updating. In Fig. 13, `fopen` creates a file for writing. Fig. 14 shows the run of the code presented in Fig. 13 without hardening, whereas Fig. 15 shows the run of the same code hardened by the GIMPLE weaving capabilities using the pattern presented in Fig. 12. The execution of the vulnerable code shows that the created temporary file has (rw - r

-r - -) permissions, whereas the execution of the hardened code shows that the temporary file created has (rw - - - - -) permissions, the most restrictive.

```

1 Pattern Umask_Pattern
2 BeginPattern
3
4   BeginBehavior
5     Before Call <fopen>
6       //Set the umask to the most restrictive value possible.
7       umask(S_IRWXG | S_IRWXO);
8     EndBehavior
9
10 EndPattern

```

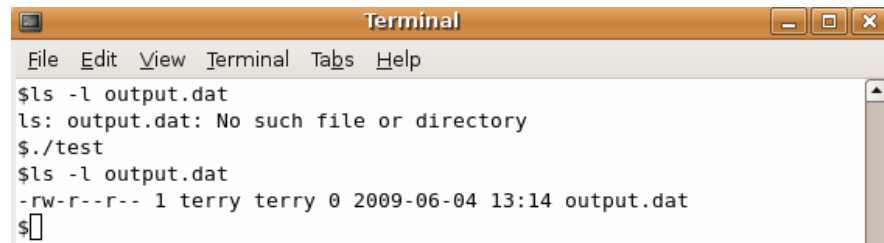
Fig. 12. SHL Hardening Pattern for Umask Function

```

1 int fp = fopen("output.dat", "w+");

```

Fig. 13. Vulnerable Code - Umask Function



```

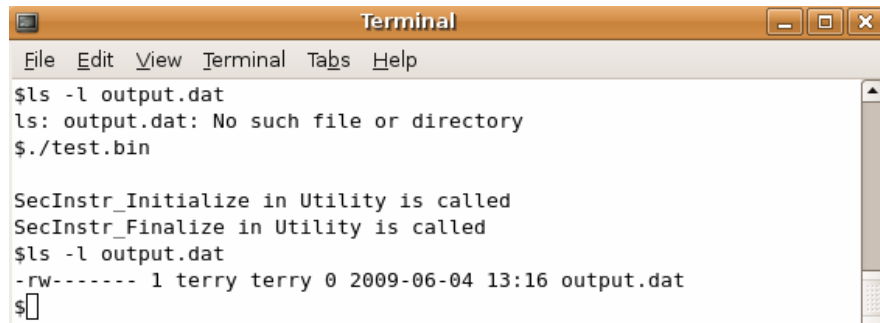
Terminal
File Edit View Terminal Tabs Help
$ls -l output.dat
ls: output.dat: No such file or directory
$./test
$ls -l output.dat
-rw-r--r-- 1 terry terry 0 2009-06-04 13:14 output.dat
$

```

Fig. 14. Vulnerable Code Execution - Umask

7.2 Use of Deprecated Function

The C library `rand()` function is a deprecated one, and it does not have good random number properties. It produces numbers that can be easily guessed by attackers and should never be used especially for cryptographic purposes. The CERT coding rules forbid the usage of this function and recommend using the function `random()` instead. The CERT rules that target this issue are:



```

Terminal
File Edit View Terminal Tabs Help
$ls -l output.dat
ls: output.dat: No such file or directory
$./test.bin

SecInstr_Initialize in Utility is called
SecInstr_Finalize in Utility is called
$ls -l output.dat
-rw----- 1 terry terry 0 2009-06-04 13:16 output.dat
$

```

Fig. 15. Hardened Code Execution - Umask

```

1 Pattern Rand_Pattern
2 BeginPattern
3
4   BeginBehavior
5     Replace Call <rand>
6     //Create seed based on current time
7     time_t now = time(NULL);
8     if (now == (time_t) -1)
9     {
10      /* handle error */
11    }
12    srand(now);
13    //Use random() instead of rand()
14    random();
15  EndBehavior
16
17 EndPattern

```

Fig. 16. SHL Hardening Pattern for Rand Function

```

1 printf("%d, ", rand()); /* Always generates the same sequence */

```

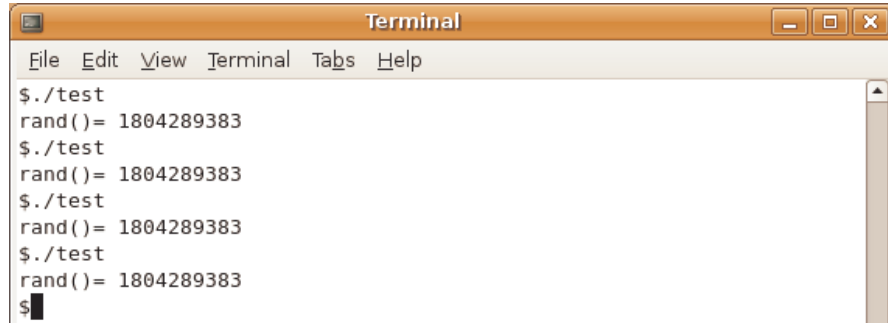
Fig. 17. Vulnerable Code - Rand Function

- This rule appears in the C Secure Coding Standard as MSC30-C. Do not use the `rand()` function for generating pseudorandom numbers.
- This rule appears in the C++ Secure Coding Standard as MSC30-CPP. Do not use the `rand()` function for generating pseudorandom numbers.
- This rule appears in the C Secure Coding Standard as MSC32-C. Ensure your random number generator is properly seeded.
- This rule appears in the C++ Secure Coding Standard as MSC32-CPP. Ensure your random number generator is properly seeded.

This vulnerability exists in well-known packages such as:

- The package *apache-1.3.41*, which is used for password generation.

- The package *pzebra-0.95a* and the package *emacs-22.3*, which are used for time synchronization purposes.



```

Terminal
File Edit View Terminal Tabs Help
$./test
rand()= 1804289383
$./test
rand()= 1804289383
$./test
rand()= 1804289383
$./test
rand()= 1804289383
$

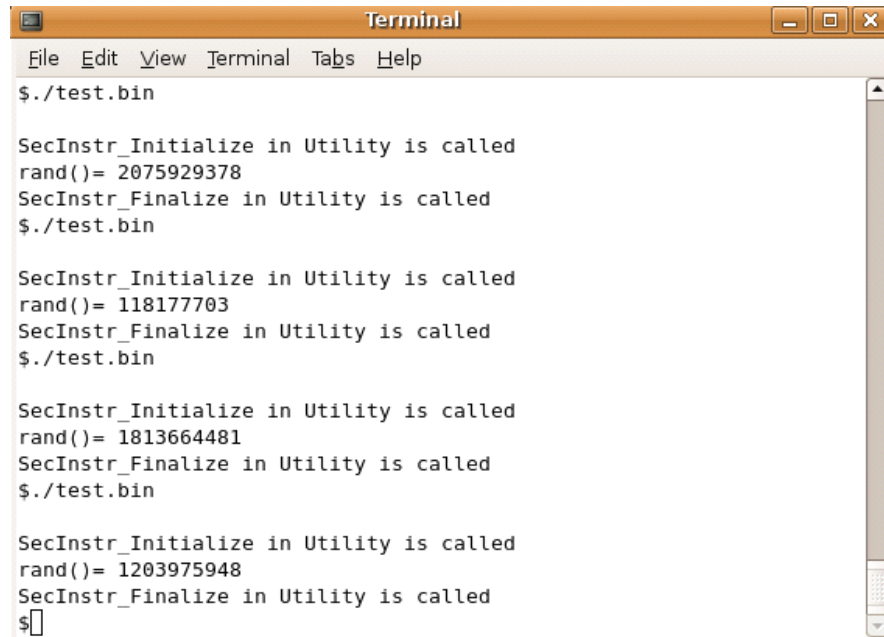
```

Fig. 18. Vulnerable Code Execution - Rand

Fig. 16 presents the pattern elaborated in SHL to solve this problem. The pattern replaces any call to the function `rand()` with two consecutive instructions. The first instruction creates a new seed based on the current time whereas the second one executes the `random()` function using the generated seed by the first instruction. The lines 7-14 in Fig. 16 represent the *Code* in SHL syntax. The code in Fig. 17 prints a random number using the the function `rand()`. Fig. 18 shows multiple executions of the code presented in Fig. 17 without hardening, whereas Fig. 19 shows executions of the same code hardened by the GIMPLE weaving capabilities using the pattern in Fig. 16. The execution of the vulnerable code several times generates the same pseudorandom number using the `rand()` function. When `rand()` is not seeded, it uses 1 as a default seed. No matter how many times this code is executed, it always produces the same pseudorandom number. Even if the `rand()` function is properly seeded, this solution does not work because the numbers generated by `rand()` have a comparatively short cycle and may be predictable. On the other hand, the execution of the hardened code produces different random number sequences at different calls exploring the correctness of the security hardening process and the feasibility of our propositions.

7.3 Monitoring

Many dynamic program analysis tools need to observe program execution and gather information in order to analyze the testee programs' runtime behavior. This task can be carried out by monitoring tools. Program monitoring can be performed at different levels. At a low level, hardware monitors extend the target system with specialized hardware architecture. For example, researchers have



```

Terminal
File Edit View Terminal Tabs Help
$./test.bin

SecInstr_Initialize in Utility is called
rand()= 2075929378
SecInstr_Finalize in Utility is called
$./test.bin

SecInstr_Initialize in Utility is called
rand()= 118177703
SecInstr_Finalize in Utility is called
$./test.bin

SecInstr_Initialize in Utility is called
rand()= 1813664481
SecInstr_Finalize in Utility is called
$./test.bin

SecInstr_Initialize in Utility is called
rand()= 1203975948
SecInstr_Finalize in Utility is called
$

```

Fig. 19. Hardened Code Execution - Rand

included counters in their microprocessor's hardware to profile the execution of some predefined events. The advantage of hardware monitoring is its low overhead. The major drawback of this approach is that monitors can only observe low-level data at restricted types of observation points. Software monitoring is a more widely-used approach to observe a program's runtime behavior. In contrast to hardware monitors, software monitors can easily access high-level data, e.g., objects and structures in modern programming languages. Additionally, they can be designed to observe various program points such as function calls and to access particular variables. Implementing software monitoring techniques necessitates program instrumentation. In this case study, we target *gzip* version 1.2.4 (96 files totaling to 24,247 lines of code) for monitoring purposes. The experiment picks out calls to the standard C library functions for memory management and instruments the needed code to help detect memory management vulnerabilities. There are various instrumentation techniques to achieve this goal. In the following we discuss these techniques with their advantages and disadvantages.

Library Replacement Instrumentation The simplest application to program instrumentation is realized without transforming the program. The technique involves only replacing the library used by the program. For example, the

dmalloc memory debugging library replaces the standard C library to monitor memory management behavior of the monitored program. The monitoring functionality is included in the *dmalloc* library. This approach is clean and efficient. However, it prevents us from monitoring program behaviors that are not associated with external libraries. Therefore, it can only be used in very dedicated applications.

Preprocessor Macros Instrumentation C and C++ compilers call the preprocessor during the first phase of compilation to include external files into the compiled source code and perform textual substitutions that are defined by macros. Simple code instrumentation can be implemented by utilizing the code transformation functionality of the preprocessor. Using macro-assisted source code transformation to perform code instrumentation is easy to implement. However, only a few program points can be effectively instrumented with this approach because preprocessors cannot access the lexical structure and semantics of the instrumented program.

Parser-assisted Source Code Instrumentation Parser builds the parse tree and abstract syntax tree of the program after source code is preprocessed. Therefore, it can access more information than can the preprocessor. The information includes the full syntactical structure. Parser-assisted source code transformation can be very powerful. For example, the Puma library from AspectC++ project [67] can perform sophisticated source code transformation at various program points. Two main disadvantages of source code weaving are:

- Language dependency: Source code weaving is written explicitly for the syntax of the input language.
- Limited expressiveness: Aspects are limited to the expressive power of the source language. For example, it is not possible to add multiple-inheritance to a single-inheritance language. Additionally, AspectC++ intentionally does not implement the field access pointcuts such as field set and get because of the existence of free pointers. Field access through pointers is quite common in C/C++ and implies a danger of "surprising" side effects for advice.

Bytecode Instrumentation Instead of targeting the source code for instrumentation, bytecode has been chosen to inject the required monitoring code. Based on bytecode instrumentation, aspect weaving tools, such as AspectJ, allow insertion of code at well-defined points in Java programs without resorting to source code manipulation. The main disadvantages of bytecode weaving are:

- Bytecode dependency: Bytecode weaving is written explicitly for Java bytecode instruction sets and accordingly the source code should be written in Java.
- Bytecode is not a structured language. Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines and block structures.

- Compared to GIMPLE representation, bytecode representation needs more instructions per one line in the source code. This means more time is needed to pick out the required points in a program [43].
- In order to know whether a class/method/field should be considered for weaving, the weaver needs to do matching on metadata for this class or member. Most AOP frameworks and AOP-applied products have some sort of high-level expression language (pointcut expressions) to define where (at which join points) a code block (advice) should be weaved into. These expression languages can, for example, let you pick out all methods that have a return type that implements an interface of type T . This information is not available in the bytecode instructions representing the call to a specific method M . The only way of knowing if this specific method M should be weaved or not is to look it up in some sort of class database, query its return type, and check if its return type implements the given interface T [43].
- Lack of a program's semantic information, such as control flow and dataflow.

GIMPLE Instrumentation With GCC's internal APIs, it is easy to manipulate the GIMPLE representation. The advantages of GIMPLE weaving over the aforementioned approaches are:

- GIMPLE is an intermediate representation produced by the GCC compiler to represent programs written in C, C++, Objective C, Fortran, Java, and Ada. GIMPLE instrumentation allows defining a common monitoring implementation for all programming languages supported by the GCC compiler. For example, instead of having a specific compiler for every aspect-oriented programming language that tries to match join points for monitoring purposes depending on source code or bytecode, the matching and the weaving are done on GIMPLE trees without focusing on specific source code or bytecode.
- GCC is a well-developed and -tested multi-platform compiler. It can cross compile code for various hardware chips and operating systems.
- GIMPLE has a structured grammar [13]. With GCC's internal APIs, it is easy to manipulate the GIMPLE representation [14].
- GIMPLE three-address code provides a common infrastructure for control flow and dataflow analysis and optimization. Together with its associated APIs, GIMPLE helps provide a platform not only for the development of high-level code-optimization techniques, but also for new static-analysis tools, applicable to all of GCC's input languages.
- The existing optimization techniques can apply to the instrumented code, resulting in efficient executable file. Optimization of the instrumented program is crucial to dynamic analysis. For sounder analysis results, the instrumented program is usually executed as many times as needed to explore different execution paths of the program. Therefore, a better optimized program can reduce analysis time.
- GIMPLE weaving allows efficient aspect interactions detection, as explained in Section 5.2. If we do not depend on GIMPLE, we need to define a sepa-

rate base facts generator for each programming language and a separate CT generator for each aspect-oriented language.

- GIMPLE weaving can deal with many security hardening transformations that are language-dependent. These transformations can be written using SHL using the same language of the source code. Both the source code and the transformations are transformed to GIMPLE to be weaved together to produce the secure code.

We evaluate the scalability and the usability of our approach with *gzip* version 1.2.4. We enable code instrumentation targeting the standard C library functions for memory management. The code instrumentation is performed successfully and without difficulties. Regarding overhead, compile-time performance is generally worse in aspect weavers than in their traditional compiler counterparts due to the additional work necessary for picking out joint points that match the specified pointcuts. Since bytecode weaving is the most comparable approach to GIMPLE weaving, it is reasonable to conduct a comparison between bytecode weaving and GIMPLE weaving from a performance perspective. We can say that GIMPLE weaving outperforms the bytecode weaving for the following reasons: Bytecode representation needs more instructions per one line in the source code [43]. This means more time is needed to pick out the required points in a program. Most AOP frameworks and AOP-applied products have some sort of high-level expression language (pointcut expressions) to define where (at which join points) a code block (advice) should be weaved in. These expression languages can, for example, let you pick out all methods that have a return type which implements an interface of type "T". This information is not available in the bytecode instructions representing the call to a specific method "M". The only way of knowing if this specific method "M" should be weaved or not is to look it up in some sort of class database, query its return type, and check if its return type implements the given interface "T" [43]. In contrast, this information is specified in GIMPLE grammar and can be extracted easily by the internal APIs. GIMPLE helps provide a platform not only for the development of high-level code-optimization techniques, but also for new static-analysis tools, applicable to all of GCC's input languages. The existing optimization techniques can apply to the instrumented code, resulting in an efficient executable file. Optimization of the instrumented program is crucial to dynamic analysis. For sounder analysis results, the instrumented program is usually executed as many times as needed to explore different execution paths of the program. Therefore, a better optimized program can reduce analysis time.

8 Related Work

In this section, we discuss some relevant contributions related to the defined framework in this paper.

8.1 General Approaches for Security Injection

Application wrappers are software containers that control the interactions between untrusted programs and their execution environments. Accordingly, they have been used to enhance the security of applications [37, 38, 68]. Current software wrapper technology revolves around interception of exported functions. Fraser *et al.* [38] presented techniques for developing wrappers to augment security in Commercial Off-The-Self (COTS) applications. This approach works well with interception at kernel level Application Programming Interfaces (API), which are interfaces between applications and operating systems. HEALERS [37] is a fault-containment wrapper that has been designed to improve the security and robustness of applications. For any shared C library, it can find all functions defined in that library and automatically derive properties for those functions. Through automated fault injection experiments, it can detect arguments that cause the library to crash and derive safe argument types for each function. The toolkit can prevent heap and stack buffer overflows that are a common cause of security breaches. HEALERS can protect existing applications without access to the source code. Susskraut and Fetzer [68] have proposed a novel approach to harden software libraries to improve their robustness and security. Their approach, which is automated, general, and extensible, consists of the following stages: First, they have used a static analysis to prepare and guide the following fault injection. In the dynamic analysis stage, fault injection experiments execute the library functions with both usual and extreme input values. The experiments are used to derive and verify one protection hypothesis per function. In the hardening stage, a protection wrapper is generated from these hypothesis to reject unrobust input values of library functions. In some cases, aspects take the form of method wrappers [22], which allow aspect code to be inserted around method bodies like advice in AspectJ.

Welch and Stroud [74, 75] have proposed an approach using Kava, which is a portable reflective Java implementation for enforcing security policies on compiled code. Kava binds Java classes to metaobject classes at loadtime. Kava does not require the source code of the target class and this makes it suitable for use with compiled code such as mobile code, compiled programs, or software components. The metaobject approach traps invocations sent from an object and specify before and after behavior that invokes required security mechanisms.

Type-qualifier inference is a static analysis that can be used to statically find many classes of bugs. Type qualifiers [39, 64, 77] are used to perform taint analysis, which finds security vulnerabilities such as SQL injection and format string vulnerabilities. Fraser *et al.* [39] used type qualifiers to check the authorization properties in C programs. This involves introducing type qualifiers for arguments to sensitive operations, and the system checks these properties using type qualifier inference. They have demonstrated their work over large code bases such as the MINIX kernels. However, their approach assumes that there are variables that are common to sensitive operations and security checks; in several instances in Java code base this assumption does not hold.

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. The current research in the domain of security patterns is characterized by various publications [16, 36]. However, there is no core reference in this area because most of these contributions target specific domains.

Several other tools, such as PoET/PSLang [35] and Polymer [23] follow an aspect-oriented approach to enforce authorization policies on legacy code. In all these tools, a security analyst provides a description of locations to be protected (join points) as well as the policy check at each location (advice). These tools then weave calls to a reference monitor at each of these locations. However, when legacy servers manage their own resources, identifying locations where policy checks must be weaved becomes a challenge. By adopting an aspect-oriented approach at the design level, Montrieux *et al.* [21] specified security hardening solutions using Aspect-oriented Modeling (AOM). The secure code is generated afterward by producing either only Java code, or Java code for the functional model as well as AspectJ code for enforcing the security solutions.

8.2 AOP Approaches for Security Injection

Most of the contributions [28, 33, 42, 63] 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. We present in the following an overview of these contributions.

Cigital labs proposed an aspect-oriented extension to the C programming language called CSAW [63, 69]. Their work is primarily dedicated to improving the security of C programs. They presented typical aspects that defend against specific types of attacks and addressed local problems such as buffer overflow and data logging. By means of an example of access control, De Win *et al.* [33] investigated how well AOP could deal with the separation of security from an application. They suggested abstraction of the relevant pointcuts out of the aspect implementation in order to construct a more generic solution. Additionally, the feasibility of building a security aspect framework has been touched upon.

Ron Bodkin [28] surveyed the security requirements for enterprise applications and described examples of security crosscutting concerns. His main focus has been on authentication and authorization. He discussed cases and scenarios for these security issues and he explored how their security rules could be implemented using AspectJ. He also outlined several of the problems and opportunities in applying aspects to secure web applications that are written in Java. Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang *et al.* [42] introduced and implemented in AspectJ a reusable and generic aspect library that provides security functions. It is based on the Java Cryptography Extension (JCE) and Java Authentication and Authorization Service (JAAS) packages. To make their aspects reusable, they left the responsibility to specify and implement the pointcut to the programmer. This approach targets defining security libraries that can be used with Java code in

particular. Its goal is to prove the feasibility of reusing and integrating pre-built aspects.

Shlowikowski and Ziekinski [66] discussed security solutions based on J2EE architecture, JBoss application server, 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 woven in the original application. Masuhara and Kawauchi [53] defined the dataflow pointcut but have not provided a formal framework for this pointcut. The pointcut identifies join points based on the origins of values using dataflow tags and picks out many famous web application vulnerabilities such as cross-site scripting and SQL injection.

JBoss [6] is one of the most popular Java application servers in the industry. JBoss includes full support for J2EE-based APIs, but, beyond that, it provides many new and novel features for enterprise development, including a powerful framework for aspect-oriented programming. In JBoss, simple Java objects can leverage features such as transactions and security that are usually reserved for J2EE (e.g., EJB) objects. These features are provided as a collection of predefined aspects, and can be applied to application objects via dynamic weaving without requiring the application itself to be recompiled. In other words, the JBoss AOP framework allows developers to write plain Java objects and apply enterprise-type services later in the development cycle without changing a line of Java code. JBoss AOP framework allows software developers to build, apply, and deploy aspects into their applications. The set of aspects that come with JBoss are aspects for remoteness, acidity, transactions, security, asynchronous invocations, and replicated/transactional caching. Spring AOP [11] introduces a simple and powerful way of writing custom aspects using either a schema-based approach or the AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving. Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. The AspectBench Compiler for Aspect (abc) [20] is an alternative compiler for the AspectJ language. It is set up as a compiler framework that allows for extensions, thus aiming to facilitate the implementation of experimental language features. The project focuses on extending the AspectJ language, as well as providing optimized implementations of the language. In addition, it works by weaving on JIMPLE, which is a 3-address intermediate representation of Java programs. Moreover, the JIMPLE representation is typed and stackless code. As compared to our approach, JBoss AOP, Spring AOP, and abc do not try to address multiple target languages or platforms.

Aspicere2 [1] is an aspect language for C where advice code is written in normal C. Its pointcut language is based on Prolog and can pick out call, execution, and local continuation join points. Aspicere2 weaver is based on Low-Level Virtual Machine (LLVM), which is very similar to GIMPLE, but GCC can generate

code for far more targets than can LLVM. Padayachee and Eloff [59] demonstrated that aspect-orientation may be used to monitor the information flows between objects in a system for the purposes of misuse detection. Misuse detection involves identifying behavior that is close to some previously defined pattern signature of a known intrusion. Jones and Hamlen [65] demonstrated that restricting aspectual policy specifications to purely declarative (but stateful) advice, and strengthening the pointcut language to include declarative predicates over runtime values, improves the feasibility of aspectual policy analysis yet remains expressive enough to encode large classes of important security properties. Such policies can be effectively enforced as in-lined reference monitors that target only Java bytecode binaries.

8.3 Semantics for AOP Weaving

The related work that addresses AOP weaving semantics is presented in this subsection. None of these works defined a semantics that demonstrates how to weave in GIMPLE trees. There has been a contribution by Adam [17] attempting AOP for GIMPLE representation. The contribution in this paper over Adam's work is a new general aspect-oriented language together with the formal semantics for advice weaving in GIMPLE trees. The most prominent research proposals in this area are the contribution of Walker *et al.* [70], wherein the authors defined the semantics of MinAML, an aspect-oriented language, and the contribution of Dantas *et al.* [32] wherein the authors defined PolyAML, a typed functional, aspect-oriented language. They used labels to mark points where advice pieces were to be injected. Advice pieces are applied to the argument or to the result of a function.

Tatsuzawa *et al.* [55] implemented an aspect-oriented version of core O'Caml called Aspectual Caml. Aspectual Caml carries out type inference on advice pieces without consulting the types of the functions designated by the pointcuts. In addition, there are no formal definitions for Aspectual Caml.

Wand *et al.* [71] presented a denotational semantics for pointcuts and advice pieces of an AOP language defined in the Aspect Sand Box (ASB) project [34]. 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.* [72] provided seamless integration of the AOP paradigm and strongly-typed functional languages through a static weaving process that deals with advice pieces 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 [30] has been introduced in which aspects are the primitive computational entity. The authors 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.

Belblidia and Debbabi [24] scrutinized both the source code of programs and the corresponding compiled units in AspectJ in order to determine how the basic

primitives were interpreted by the compiler. As a second step, they transformed this knowledge into formal semantics.

Jürjens [44] aimed to develop security-critical systems in a formally-based approach using the notation of the Unified Modeling Language (UML). He presented the extension UMLsec of UML that allows one to express security-relevant information within the diagrams in a system specification. UMLsec is defined in form of a UML profile using the standard UML extension mechanisms. In particular, the associated constraints give criteria to evaluate the security aspects of a system design by referring to formal semantics of a simplified fragment of UML.

9 Conclusion

We have presented in this paper an aspect-oriented approach based on GIMPLE for the systemization of application security hardening. The security solutions are woven into GIMPLE representations in a systematic way, eliminating the need for manual hardening that may generate a considerable number of errors. In this respect, we have presented a formal specification for GIMPLE weaving and the implementation strategies of the proposed weaving semantics. To achieve this goal, syntax for a common aspect-oriented language that is abstract and multi-language support together with syntax for a core set for GIMPLE constructs have been presented to express the weaving semantics. Afterward, we have handled the correctness and completeness of GIMPLE weaving in two different ways. In the first approach, we have proved them according to the rules and algorithms provided in this paper. On the other hand, we have accommodated in the second approach Kniesel's discipline to explore the correctness and completeness of GIMPLE weaving. At the end, we have explored the viability and the relevance of our propositions by applying the defined approach for systematic security hardening to develop case studies. The main advantage of the defined solution is that it supports multiple languages. GIMPLE weaving accompanied by a common aspect-oriented language allows security experts providing security solutions using this common language, lets developers focus on the main functionality of programs by relieving them from the burden of security issues, unifies the matching and the weaving processes for mainstream languages, and facilitates introducing new security features in AOP languages. Regarding future work, we are currently working on extending the GIMPLE weaving semantics to include other defined constructs, such as the cflow pointcut, and their implementation into GCC. Moreover, we are focused on applying this approach using other programming languages such as Java to prove the multi-language support claim. GIMPLE is not object-oriented, thus object-oriented features are transformed to imperative equivalents during the translation to GIMPLE. There is a need to understand all these transformations to identify all the new possible pointcuts that can pick out points related to these object-oriented features.

References

1. Aspicere2, more AOP for C, available at <http://users.ugent.be/~badams/aspicere2/> (accessed in Feb. 2009)
2. Bell Canada, available at <http://bell.ca/home/> (accessed in Jun. 2009)
3. ANTLR Parser Generator, available at <http://www.antlr.org/> (accessed in Dec. 2012)
4. CERT C Secure Coding Standard, available at <https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard> (accessed in Dec. 2012)
5. CERT C++ Secure Coding Standard, available at <https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637> (accessed in Dec. 2012)
6. Community driven open source middleware, available at <http://www.jboss.org/> (accessed in Dec. 2012)
7. ConTraCT, available at <http://roots.iai.uni-bonn.de/research/contract/> (accessed in Dec. 2012)
8. GCC-the GNU Compiler Collection, available at <http://gcc.gnu.org/> (accessed in Jun. 2009)
9. JTransformer Framework, available at <http://roots.iai.uni-bonn.de/research/jtransformer/> (accessed in Feb. 2009)
10. OCaml for Scientists, available at http://www.ffconsultancy.com/products/ocaml_for_scientists/chapter1.html (accessed in Dec. 2012)
11. SpringSource.org, available at <http://www.springsource.org/> (accessed in Feb. 2009)
12. The CERT Sun Microsystems Secure Coding Standard for Java, available at <https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java> (accessed in Dec. 2009)
13. Rough GIMPLE Grammar - GNU Compiler Collection, available at <http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gccint/Rough-GIMPLE-Grammar.html> (accessed in Dec. 2012)
14. GNU Compiler Collection (GCC) Internals, available at http://idlebox.net/2011/apidocs/gcc-4.6.0.zip/gccint-4.6.0/gccint_12.html (accessed in Dec. 2012)
15. US Department of Homeland Security Coding Rules, available at <https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html> (accessed in Sept. 2009)
16. A. M. Braga, C.M.F.R., Dahab, R.: Tropic: A pattern language for cryptographic software. Tech. Rep. IC-99-03, Institute of Computing, UNICAMP (1999)
17. Adam, B.: Language-independent aspect weaving. In: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005 (2005)
18. Alhadidi, D., Belblidia, N., Debbabi, M.: Security crosscutting concerns and AspectJ. In: Proceedings of the International Conference on Privacy, Security and Trust. McGraw-Hill, Markham, Ontario, Canada (2006)
19. Alhadidi, D., Debbabi, M., Bhattacharya, P.: New aspectj pointcuts for integer overflow and underflow detection. Information Security Journal: A Global Perspective 17(5&6), 278–287 (2008)
20. Avgustinov, P., Christensen, A., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ

- compiler. In: Proceedings of the 4th international conference on Aspect-oriented software development, AOSD '05. pp. 87–98. ACM (2005)
21. Montrieux, L. and Jürjens, J. and Haley, Ch. B. and Yu, Y. and Schobbens, P. and Toussaint, H.: Tool support for code generation from a UMLsec property. In: Proceedings of the international conference on Automated software engineering, ASE '10. pp. 357–358. ACM (2010)
 22. Baker, J., Hsieh, W.: Runtime aspect weaving through metaprogramming. In: Proceedings of the 1st international conference on Aspect-oriented software development, AOSD'02. pp. 86–95. ACM (2002)
 23. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. SIGPLAN Notices 40(6), 305–314 (2005)
 24. Belblidia, N., Debbabi, M.: Formalizing AspectJ weaving for static pointcuts. In: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods. pp. 50–59. IEEE (2006)
 25. Bishop, M.: Computer Security: Art and Science. Addison-Wesley Professional (2002)
 26. Bishop, M.: 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 in Nov. 2011)
 27. Machata, P.: Construction of GNU compiler collection front end (2007), available at http://www.feec.vutbr.cz/EEICT/2007/sbornik/02-magisterske_projekty/07-informacni_systemy/09-pmachata.pdf (accessed in Nov. 2012)
 28. Bodkin, R.: Enterprise security aspects. http://www.cs.kuleuven.ac.be/~distrinet/events/aosdsec/AOSDSEC04-Ron_Bodkin.pdf,
 29. Böllert, K.: On weaving aspects. In: Proceedings of the Workshop on Object-Oriented Technology. pp. 301–302. Springer-Verlag (1999)
 30. Bruns, G., Jagadeesan, R., Jeffrey, A., Riely, J.: muABC: A minimal aspect calculus. In: Proceedings of the Fifteenth International Conference on Concurrency Theory (CONCUR 2004). Lecture Notes in Computer Science, vol. 3170. Springer-Verlag (2004), <http://cm.bell-labs.com/who/ajeffrey/papers/concur04.pdf>
 31. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9. pp. 88–98. ACM (2001)
 32. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: Polyaml: a polymorphic aspect-oriented functional programming language. In: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ICFP '05. pp. 306–319. ACM (2005)
 33. DeWin, B.: Engineering Application Level Security through Aspect Oriented Software Development. Ph.D. thesis, Katholieke Universiteit Leuven, Belgium (2004)
 34. Dutchyn, C., Kiczales, G., Masuhara, H.: Aspect Sand Box Project (2002), available at <http://www.cs.ubc.ca/labs/spl/projects/asb.html> (accessed in Nov. 2008)
 35. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University, Ithaca, NY, USA (2004)
 36. F. L. Brown, J. DeVietri, G.D.E.B.F.: The authenticator pattern. In: Proceedings of Pattern Language of Programs (PloP'99). Chicago, Illinois, USA (1999)

37. Fetzer, C., Xiao, Z.: HEALERS: A toolkit for enhancing the robustness and security of existing applications. *International Conference on Dependable Systems and Networks* 0, 317 (2003)
38. Fraser, T., Badger, L., Feldman, M.: Hardening COTS software with generic software wrappers. *DARPA Information Survivability Conference and Exposition* 2, 1323 (2000)
39. Fraser, T., N. Petroni, J.W.A.: Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In: *Proceedings of the 2006 workshop on Programming languages and analysis for security*. pp. 3–6. ACM (2006)
40. Harbulot, B., Gurd, J.R.: A join point for loops in AspectJ. In: *AOSD '06: Proceedings of the 5th International Conference on Aspect-oriented Software Development*. pp. 63–74. ACM, New York, NY, USA (2006)
41. Howard, M., LeBlanc, D.E.: *Writing Secure Code*. Microsoft, Redmond, WA, USA (2002)
42. Huang, M., Wang, C., Zhang, L.: Toward a reusable and generic security aspect library. http://www.cs.kuleuven.ac.be/~distrinet/events/aosdsec/AOSDSEC04_Minwell
43. Vasseur, A., Dahlstedt, J., BEA Systems: Java Virtual Machine support for Aspect-Oriented Programming. http://jonasboner.com/publications/JVM_AOP_AOSD2006.pdf (accessed in Nov. 2012)
44. Jürjens, J.: Model-based security engineering with UML: Introducing security aspects. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) *Formal Methods for Components and Objects*, 4th International Symposium, FMCO. *Lecture Notes in Computer Science*, vol. 4111, pp. 64–87. Springer-Verlag (2005)
45. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) *Proceedings European Conference on Object-Oriented Programming*, vol. 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York (1997), citeseer.ist.psu.edu/kiczales97aspectoriented.html
46. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01*. pp. 327–353. Springer-Verlag (2001)
47. Kim, H.: An AOSD implementation for C#. Tech. Rep. TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin (2002)
48. Kniesel, G.: A logic foundation for conditional program transformations. Tech. Rep. IAI-TR-2006-01, CS Dept. III, University of Bonn, Germany (2006)
49. Kniesel, G., Koch, H.: Static composition of refactorings. *Science of Computer Programming* 52(1-3), 9–51 (2004)
50. Kniesel, G.: Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development* 5, 135–186 (2009)
51. Löding, H., Peleska, J.: Symbolic and abstract interpretation for C/C++ programs. *Electronic Notes Theoretical Computer Science* 217, 113–131 (2008)
52. Marpons, G., Marino, J., Polo, A.: Adding coding rule checking capabilities to the GCC toolchain. In: Hutton, A., Ross, C., Lockhart, J. (eds.) *Proceedings of the GCC Developers' Summit 2008*. pp. 43–54. Ottawa, Canada (June 17–19 2008)
53. Masuhara, H., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In: Ohori, A. (ed.) *Programming Languages and Systems, First Asian Symposium, APLAS 2003*, Beijing, China. *Lecture Notes in Computer Science*, vol. 2895, pp. 105–121. Springer-Verlag (2003)

54. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Hedin, G. (ed.) *Compiler Construction, CC 2003*, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland. *Lecture Notes in Computer Science*, vol. 2622, pp. 46–60. Springer-Verlag (2003)
55. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: *ICFP05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. ACM (2005)
56. Mourad, A., Laverdière, M.A., Debbabi, M.: Security hardening of open source software. In: *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM (2006)
57. Mourad, A., Laverdière, M.A., Debbabi, M.: A high-level aspect-oriented based language for software security hardening. In: *Proceedings of the International Conference on Security and Cryptography (Secrypt)*. Barcelona, Spain (2007)
58. Mourad, A., Laverdière, M.A., Debbabi, M.: Towards an aspect oriented approach for the security hardening of code. In: *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops, AINAW '07*. pp. 595–600. IEEE (2007)
59. Padayachee, K., Eloff, J.H.P.: An Aspect-Oriented Model to Monitor Misuse. In: *Proceedings of Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, AINAW '07*. pp. 273–278. Springer Netherlands (2007)
60. Rho, T., Kniesel, G.: Uniform genericity for aspect languages. Tech. Rep. IAI-TR-2004-4, CS Dept. III, University of Bonn, Germany (2004)
61. Schumacher, M.: *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Springer (2003)
62. Seacord, R.C.: *Secure Coding in C and C++*. SEI Series, Addison-Wesley (2005)
63. Shah, V.: An aspect-oriented security assurance solution. Tech. Rep. AFRL-IF-RS-TR-2003-254, Cigital Labs (2003)
64. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: *Proceedings of the 10th conference on USENIX Security Symposium, SSYM'01*. pp. 16–16. USENIX Association, Berkeley, CA, USA (2001)
65. Jones, M., Hamlen, K. W.: Disambiguating aspect-oriented security policies. In: *Proceedings of the International Conference on Aspect-Oriented Software Development, AOSD '10*. pp. 193–204. ACM (2010)
66. Slowikowski, P., Zielinski, K.: Comparison study of aspect-oriented and container managed security. In: *Proceedings of the ECCOP Workshop on Analysis of Aspect-Oriented Software* (2003)
67. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An aspect-oriented extension to the C++ programming language. In: *Proceedings of the Fortieth International Conference on Tools Pacific, CRPIT '02*. pp. 53–60. ACM (2002)
68. Susskraut, M., Fetzter, C.: Robustness and security hardening of COTS software libraries. In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. pp. 61–71. IEEE (2007)
69. Viega, J., Bloch, J., Chandra, P.: Applying aspect-oriented programming to security. *Cutter IT Journal* 14(2), 31–39 (2001)
70. Walker, D., Zdancewic, S., Ligatti, J.: A theory of aspects. In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, ICFP '03*. pp. 127–139. ACM (2003)

71. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems* 26(5), 890–910 (2004)
72. Wang, M., Chen, K., Khoo, S.C.: Type-directed weaving of aspects for higher-order functional languages. In: *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '06*. pp. 78–87. ACM (2006)
73. Callanan, S., Grosu, R., Huang, X., Smolka, S. A., Zadok, E.: Compiler-assisted software verification using plug-ins. In: *Proceedings of the conference on Parallel and distributed processing, IPDPS'06*. pp. 285–285. IEEE (2006)
74. Welch, I., , Stroud, R.J.: Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security* 10(4), 399–432 (2002)
75. Welch, I., Lu, F.: Policy-driven reflective enforcement of security policies. In: *Proceedings of the 2006 ACM symposium on Applied computing*. pp. 1580–1584. ACM (2006)
76. Yang, Z.: On Building A Dynamic Vulnerability Detection System. Master's thesis, Concordia University (2007)
77. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for static analysis of authorization hook placement. In: *Proceedings of the 11th USENIX Security Symposium*. pp. 33–48. USENIX Association, Berkeley, CA, USA (2002)

Second Response to Reviewers

We would like to sincerely thank the reviewers for their insightful comments. The paper has been revised to address all the comments. Below, we give detailed feedback to the comments from each reviewer.

Reviewers' comments:

Reviewer #1:

The authors have taken my comments into account, so I am happy to suggest acceptance.

*** Authors: We thank the reviewer for the valuable suggestions.

Reviewer #2:

The authors addressed the reviewers comments and amended the paper accordingly. An overall English review is required for the final version.

*** Authors: An English editor revised the whole manuscript to improve the language. Additionally, the final version has been revised by the authors.

Information about the English Editor:

Roberta Fountain
Green Office
3960 Hollyhock Way
San Luis Obispo CA 93401
805-235-4635
E-MAIL: RFOUNTAIN@GREENOFFICE.INFO
WWW.GREENOFFICE.INFO