New aspect-oriented constructs for security hardening concerns

Azzam Mourad*, Andrei Soeanu, Marc-André Laverdière, Mourad Debbabi

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal (QC), Canada

Abstract

In this paper, we present new pointcuts and primitives to Aspect-Oriented Programming (AOP) languages that are needed for systematic hardening of security concerns. The two proposed pointcuts allow to identify particular join points in a program’s control-flow graph (CFG). The first one is the GAFlow, Closest Guaranteed Ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second one is the GDFlow, Closest Guaranteed Descendant, which returns the closest child join point that can be reached by all paths starting from the pointcut of interest. The two proposed primitives are called ExportParameter and ImportParameter and are used to pass parameters between two pointcuts. They allow to analyze a program’s call graph in order to determine how to change function signatures for passing the parameters associated with a given security hardening. We find these pointcuts and primitives to be necessary because they are needed to perform many security hardening practices and, to the best of our knowledge, none of the existing ones can provide their functionalities. Moreover, we show the viability and correctness of the proposed pointcuts and primitives by elaborating and implementing their algorithms and presenting the result of explanatory case studies.

1. Introduction

Security is taking an increasingly predominant role in today computing world. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. The challenge is even greater when legacy systems must be adapted to network/web environments, while they are not originally designed to fit into such high-risk environments. Security tools and guidelines have been available for developers for few years already, but their practical adoption is limited so far. Software maintainers must face the challenge to improve program security and are often under-equipped to do so. In some cases, little can be done to improve the situation, especially for Commercial-Off-The-Shelf (COTS) software products that are no longer supported, or for in-house programs for which their source code is lost. However, whenever the source code is available, as it is the case for Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided.

As a result, integrating security into open source software becomes a very challenging and interesting domain of research. In this context, we have defined (Mourad et al., 2006) software security hardening as any process, methodology,
product or combination thereof that is used to add security functional
ties and/or remove vulnerabilities or prevent their exploitation in exis
ting software. Few concepts and approaches emerged in the litera
ture to help and guide developers to secure software. We can dis
tinguish from them the hardening methods at the operating system and network levels, secure programming solutions published in many books and reviews, security patches, security design patterns, etc. (Schumacher et al., 2006; Seacord, 2005; Howard and LeBlanc, 2002; Howard and Lipner, 2006; Wheeler, 2003; Bishop, 2005). Their security hardening practices are typically carried out by integrating and injecting manually and in ad-hoc manner the security components and code into software. This requires from the developers to have a good knowledge of the internal functioning of the programme, in addition to high expertise in the field of security applied.

However, security hardening is a difficult and critical procedure. If it is applied manually, specially in large scale software (e.g., thousands, millions line of code), it requires high security expertise and lot of time to be tackled. It may also create other vulnerabilities. Moreover, there is always a difficulty in finding the software engineers or developers who are specialized in both the security solution domain and the software functionality domain. In fact, this is an open problem raised by several companies’ managers. As such, any attempt to address security concerns must take into consideration the aforementioned problems, which is not the case in the existent approaches. In this context, the main intent of our research is to create methods and solutions to integrate systematically and consistently security models and components into software (Mourad et al., 2007a,b, 2008).

One way of achieving these objectives is by separating out the security concerns from the rest of the application, such that they can be addressed independently and applied globally. More recently, several proposals have been advanced for code injection, via an aspect-oriented computational style, into source code for the purpose of improving its security (Bodkin, 2004; DeWin, 2004; Huang et al., 2004; Shah, 2003; Xu et al., 2006; Slowikowski and Zielinski, 2003; Shah and Hill, 2003; Navarro et al., 2006; Clarke et al., 2007). In this context, AOP appears to be a promising paradigm for software security hardening (see definition in Section 2.2), which is an issue that has not been adequately addressed by previous programming models such as object-oriented programming (OOP). It is based on the idea that computer systems are better programmed by separately specifying the various concerns, and then relying on underlying infrastructure to compose them together. The techniques in this paradigm were precisely introduced to address the development problems that are inherent to crosscutting concerns. Aspects allow us to precisely and selectively define and integrate security objects, methods and events within application, which make them interesting solutions for many security issues.

However, AOP was not initially designed to address security issues, which resulted in some limitations in the current technologies (Bonér, 2005; Harbulot and Gurd, 2005; Kiczales, 2003; Masuhara and Kawachi, 2003; Myers, 1999). Indeed, some security hardening activities cannot be applied due to missing features. For instance, while implementing an AOP-based solution for securing the connections of client applications, we opted to initialize/de-initialize and build/de-
build the data structures and objects needed for GNU/TLS security library in the main function (please see Listing 7 in Section 4.3.1 for more detail). Although this solution works for small size applications with single feature, it is not applicable and relevant for large scale applications with multiple functionalities. Many APIs initialization and unneeded operations may be performed, even if their corresponding features (i.e., the features using them) are not called during an execution context of a program. Such solution could also be ruinable for embedded applications, where the energy and memory resources are limited.

Moreover, during our security hardening experiments, we faced the problem of passing needed variables and parameters related to GNU/TLS library (e.g., TLS Session) between the application components. Such limitations forced us, when applying security hardening practices, to perform programming gymnastics (when possible), resulting in integrating additional modules and changing several functions in the application to pass the needed variables (please see Listing 7 in Section 4.3.1 for more detail). Such solution is not realistic in the case of large scale applications with multiple features, where there are complex dependencies and relations between their components. Any changes in one component lead to apply several modifications in all its dependent ones, which requires many complex re-engineering actions to be performed.

In this context, we present new pointcuts and primitives to AOP languages in general, and SHL in particular, which are needed for systematic hardening of security concerns. The two proposed pointcuts, which are first introduced in Lav-erdière et al. (2007), allow the identification of particular join points in a program control-flow graph (CFG). The first one is the GAFlow, Closest Guaranteed Ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second one is the GDFlow, Closest Guaranteed Descendant, which returns the closest child join point that can be reached by all paths starting from the pointcut of interest. The two proposed primitives are called ExportParameter and ImportParameter and are used to pass parameters between two pointcuts. They allow to analyze a program call graph in order to determine how to change function signatures for passing the parameters associated with a given security hardening task.

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

In this paper, we provide our new contributions toward the development of a security hardening framework. The latter is based on AOP, thus addressing the limitations of this technology and enriching it with new pointcuts and primitives for security hardening concerns constitutes an essential task to reach our objectives. We include a brief summary of the whole
proposed approach for software security hardening in Section 2.2. The remaining of this paper is organized as follows. The background and related work are presented in Section 2. Then, the proposed pointcuts and primitives are defined and specified in Section 3. Afterwards, the usefulness of our propositions and their advantages are discussed in Section 4. Then, the algorithms necessary for implementing the proposed pointcuts and primitives are presented in Section 5. This section also shows the implementation results into case studies. Finally, some summarizing conclusions are presented in Section 6.

2. Background and related work

Our experiments explored the usefulness of AOP in reaching the objective of having systematic security hardening. On the other hand, we have also distinguished, together with other documented related work (Boneár, 2005; Harbulot and Gurd, 2005; Kiczales, 2003; Masuhara and Kawauchi, 2003; Myers, 1999), the limitations of the available AOP technologies and languages for some security issues. Addressing such limitations can be achieved by elaborating pointcuts and primitives constructs that improve the conditions on which we can inject appropriately the security code. Many authors have made contributions in this field, which we summarize in this section. Moreover, we present a brief introduction on AOP and its advice-pointcut model.

2.1. AOP/advice-pointcut model

AOP represents a family of approaches that allow and promote an effective integration of different concerns into usable software in a way that cannot be achieved using the classical object-oriented decomposition. The foundation of AOP is the principle of “Separation of Concerns”, where issues that affect and crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed such as AspectJ (Kiczales et al., 2001), AspectC (Coady et al., 2001), AspectC++ (Spinczyk et al., 2002), AspectC# (Kim, 2002) and an AOP version targeting the Smalltalk programming language (Böllert, 1999). The approach adopted by most of these initiatives is called the Pointcut-Advice model. The join points, pointcuts and advices constitute its main elements.

In order to develop under this paradigm, one must first determine the code that needs to be injected into the basic (original) model. This code describes the behavior of the issues that affect and crosscut the application. Each atomic unit of code to be injected is called an advice. Then, it is necessary to identify where to inject the advice into the program. This is done by using pointcut expressions whose matching criteria restricts the set of a program join points for which the advice will be injected. A join point is an identifiable execution point in the application code and the pointcut constitutes the constructor that designates a set of join points. The pointcut expressions typically allow for matching on function calls and executions, on the control flow subsequent to a given join point, on the membership in a class, etc. At the heart of this model, is the concept of an aspect, which embodies all these elements.

An aspect is composed of zero, one, or more pointcuts and advices. In addition to this, we can use within an aspect, more specifically within the advice body, all the types and functionalities provided by a programming language libraries by including the required header files. We can also add our own libraries by implementing them inside the aspect or including them as header files. Fig. 1 describes the structure of an aspect and illustrates the matching and weaving mechanisms. The aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers. The example presented in Fig. 1 matches the call to the function $f()$, insert `code1` and `code2` before $f()$ and insert `code3` and `code4` after $f()$.

Examples of implemented aspects are presented in Section 4.

2.2. AOP-based security hardening

Security hardening practices are usually applied manually by injecting security related code into the software (Bishop, 2005; Howard and LeBlanc, 2002; Seacord, 2005; Wheeler, 2003). This
task entails experienced developers with high security expertise and knowledge of the inner workings of the software to be hardened, a requirement which is seldom fulfilled. Recently, few research initiatives discussing the appropriateness of AOP for software security have been published (Shah, 2003; DeWin, 2004; Bodkin, 2004; Huang et al., 2004). Most of them are presented as case studies that show the relevance of AOP for injecting security code. In this context, we elaborated in Mourad et al. (2007a,b, 2008) an approach based on aspect orientation to perform security hardening in a systematic way. Excerpt of the approach architecture is illustrated in Fig. 2.

The resulting security hardening solutions are described into security hardening aspect(s) to be weaved with the software code. An AOP weaver (e.g., AspectJ, AspectC++) can be employed in order to harden the aspects into the original source code. Our proposed approach constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them, and allows the developers to perform security hardening of software by applying well-defined solutions without the need to have expertise in the security solution domain. At the same time, the security hardening is applied in an organized and systematic way in order not to alter the original functionalities of the software. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build and develop our solutions. For more information, please refer Mourad et al. (2007a,b, 2008).

We illustrated the feasibility of the whole approach by applying it during the elaboration and development of several security hardening solutions including securing the connections of client-server applications, adding access control features to a program, encrypting memory contents for protection and remedying some low-level security issues in C programs. On the other hand, these experiments led us to distinguish few limitations of AOP for security hardening, which we propose solutions for some of them in this paper. Adopting AOP in our approach makes addressing such limitation an essential task towards reaching our objectives.

2.3. Related approaches

Many shortcomings of AOP for security concerns have been documented and some improvements have been suggested so far. However, to the best of our knowledge, none of the existing approaches offers the features provided by our propositions. In the sequel, we present the most noteworthy. We also present some references related to the algorithms operating on the control flow and call graphs.

A dataflow pointcut that is used to identify join points based on the origin of values is defined and formulated in Masuhara and Kawauchi (2003) for security purposes. The authors expressed the usefulness of their pointcut by presenting an example on sanitizing web-applications. For instance, such a pointcut can detect if the data sent over the network depends on information read from a confidential file. This pointcut is not fully implemented yet.

Harbulot and Gurd (2005) proposed a model of a loop pointcut that explores the need for a loop join point that predicts infinite loops, which are used by attackers to perform denial of service attacks. Their approach for recognizing loops is based on a control-flow analysis at the bytecode level in order to avoid ambiguities due to alternative forms of source code that would produce identical loops. This model contains also a context exposure mechanism for writing pointcuts that select only specific loops.

Bonér (2005) discussed a pointcut that is needed to detect the beginning of a synchronized block and add some security code that limits the CPU usage or the number of executed instructions. The author also explores the usefulness of capturing synchronized blocks in calculating the time acquired by a lock and thread management. This result can also be applied in the security context and can help in preventing many denial of service attacks.

A predicted control flow (pcflow) pointcut was introduced by Kiczales in a keynote address (Kiczales, 2003) without a precise definition. Such pointcut may allow to select points within the control flow of a join point starting from the root of the execution to the parameter join point. In the same presentation, an operator is introduced in order to obtain the minimum of two pcflow pointcuts, but it is never clearly defined what this minimum can be or how can it be obtained. These proposals could be used for software security, in the enforcement of policies that prohibit the execution of a given function in the context of the execution of another one.

Local variables set and get pointcuts were introduced in Hadidi et al. (2006) for increasing the efficiency of AOP for security concerns. They allow to track the values of local variables inside a method. It seems that these pointcuts can be used to protect the privacy and integrity of sensitive data. Their idea is based on the approach presented in Myers (1999), which describes an extension of Java called JFlow. This language allows to statically check information flow annotations within programs and provides several new features such as decentralized label model, label polymorphism, runtime label checking and automatic label inference. It also supports objects, subclassing, dynamic type tests, access control, and exceptions.

Aberg et al. (2003) presented an aspect system that addresses the crosscutting of event notifications scattered over kernel code to support Bossa, an event-based framework for process-scheduler development. This aspect system uses temporal logic to precisely describe code insertion points and sequences of instructions that require events to be inserted. In each case, the choice of event depends on properties of one or a sequence of instructions. They propose to guide the event insertion by using a set of rules, amounting to an aspect, that describes the control-flow contexts in which each event should be generated.

In a position paper Cottenier et al. (2007), argued that Aspect-Oriented Modeling (AOM) technologies have the
potential to simplify the deployment and the ability to reason about a category of crosscutting concerns that have been categorized in the literature as stateful aspects. Stateful aspects trigger on a sequence of join points instead of on a single join point. They identified three properties of AOM languages that enable them to provide more natural solutions to the stateful aspect problem. They also presented a JAsCo aspect example that captures a sequence of events (e.g., methodA – methodB – methodC) and attaches an advice to the last event (i.e., methodC).

Algorithms that operate on the call graphs and control-flow graphs have long history in the literature. In this context, we provide here a brief overview on some references related to this issue. Ryder (1979) provided one of the earliest contributions for efficient context-insensitive call graph construction in procedural languages, and her contribution was quickly followed by the notion of context sensitivity by Callahan et al. (1990). The construction of call graphs in the case of object-oriented languages and an elaborated study of different algorithms have been provided by Grove and Chambers (2001).

In Cooper et al., the authors proposed a simple and fast algorithm to calculate the dominance information (e.g., dominator set) of CFG nodes. They also surveyed most of the related algorithms and approaches and compared them to their proposition. An implementation of one of these algorithms (ClassDominanceInfo) has been provided in Holloway and Smith (1998) as part of the Machine-SUIF control-flow analysis (CFA) library. It is built on top of the control-flow graph (CFG) (Holloway and Smith, 1998) library and provides dominance analysis and natural-loop analysis. Other approaches that use lattice theory allow to efficiently compute a Lower Upper Bound (LUB) ancestor and Greater Lower Bound (GLB) descendant over lattices (Aıt-Kaci et al., 1989). However, their results do not guarantee that all paths will be traversed by the results of LUB and GLB, which is a central requirement for our related propositions. Moreover, the lattices do not support the full range of expression provided by the CFG, as the latter can be a directed cyclic graph.

3. Pointcut and primitive definitions

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

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

3.1. GAFlow and GDFlow pointcuts

The GAFlow pointcut operates on the CFG of a program. Its input is a set of join points defined in a pointcut and its output is a single join point. In other words, from the CFG notation perspective, the input is a set of nodes while the output is one node. The output node represents the closest common ancestor that constitutes (1) the closest common parent node of all the nodes belonging to the input set and (2) through which all the possible paths that reach them pass. In the worst case, the closest common ancestor will be the starting point in the program.

The GDFlow pointcut operates on the CFG of a program. Its input is a set of join points defined as a pointcut and its output is a join point. In other words, from the CFG notation perspective, the input represents a set of nodes while the output is one node. The output node (1) is the common descendant of the input nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes. In the worst case, the first common descendant will be the end point in the program.

3.2. ExportParameter and ImportParameter primitives

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

4. Discussion

This section discusses the usefulness, advantages and limitations of the proposed pointcuts and primitives for software security hardening.

4.1. Usefulness of GAFlow and GDFlow for security hardening

Many security hardening practices require the injection of code around a set of join points or possible execution paths (Bishop, 2005; Howard and LeBlanc, 2002; Seacord, 2005; Wheeler, 2003). Examples of such cases would be the injection of security library initialization/deinitialization and data structure construction, privilege level changes, atomicity guarantee, logging, etc. The current AOP models allow us only to identify a set of join points in the program and therein inject code before, after and/or around each of the identified join points. However, to the best of our knowledge, none of the

<table>
<thead>
<tr>
<th>Table 1 – Syntax of the pointcuts and primitives.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ p := \text{call}(s) \mid \text{definition}(s) \mid \text{GAFlow}(p) \mid \text{GDFlow}(p) \mid p \mid p \mid p \mid p \mid p \mid p \mid p ]</td>
</tr>
<tr>
<td>advice ( \langle \text{before} \mid \text{after} \mid \text{around} \rangle \mid e \mid i \mid e \mid i \rangle \langle \text{advice-body} \rangle )</td>
</tr>
<tr>
<td>e := \text{ExportParameter}(\langle \text{paramList} \rangle)</td>
</tr>
<tr>
<td>i := \text{ImportParameter}(\langle \text{paramList} \rangle)</td>
</tr>
<tr>
<td>paramList := parameter [ \langle \text{paramList} \rangle ]</td>
</tr>
<tr>
<td>parameter := \langle \text{type} \rangle \langle \text{identifier} \rangle</td>
</tr>
</tbody>
</table>
current pointcuts enables the identification of a join point, common to a set of other join points and satisfying the criteria of GAFlow and GDFlow, where we can inject the code only when it is needed and once for all of them. In the sequel, we present briefly the necessity and usefulness of our proposed pointcuts for some security hardening activities.

4.1.1. Security library initialization/deinitialization and data structure construction

During the development of an AOP-based solution for securing the connections of client applications, we initialized/de-initialized and built/de-built the data structures and objects needed for GNU/TLS security library in the main function. Such solution works for small size applications with single feature. However, it is not relevant for large scale applications with multiple functionalities. Many APIs initialization and unneeded operations may be performed, even if their corresponding features are not called during an execution context of a program. In the case of embedded applications where the energy and memory resources are limited, such solution could also be ruinable. The proposed pointcuts allow to solve this problem by executing these operations only for the branches of code where they are needed by identifying their GAFlow and/or GDFlow. Having both pointcuts would also avoid the need to keep global state variables about the current state of library initialization. We use as an example a part of an aspect that we elaborated for securing the connections of a client application. With the current AOP pointcuts, the aspect targets the main function as the location for the TLS library initialization, deinitialization and data structure construction, as depicted in Listing 1. In Listing 2, we see an improved aspect targeting the pointcuts GAFlow and GDFlow to perform these operations and offering more efficient results.

### Listing 1 – Excerpt of hardening aspect for securing connections using GNU/TLS.

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

### Listing 2 – Excerpt of improved hardening aspect for securing connections using GNU/TLS.

```java
advice GAFlow(call("% connect(…)") || call("% send(…)") || call("% recv(…)") || call("% recv(…)") || call("% send(…)") || call("% close(…)"); before()) {
    hardening_socketInfoStorageInit();
    hardening_initGnuTLSSubsystem(NONE);
} 
advice GDFlow(call("% connect(…)") || call("% send(…)") || call("% recv(…)"); after()) {
    hardening_deinitGnuTLSSubsystem();
    hardening_socketInfoStorageDeinit();
}
```

4.1.2. Principle of least privilege

For processes implementing the principle of least privilege, it is necessary to increase the active rights before the execution of a sensitive operation, and to relinquish such rights directly after its completion. Our pointcuts can be used to deal with a group of operations requiring the same privilege by injecting the privilege adjustment code at the GAFlow and GDFlow join points. This is applicable only in the case where no unprivileged operations are in the execution path between the initialization and the deinitialization points. The example in Listing 3 (made using combined code examples from Howard and LeBlanc (2002)) shows an aspect implementing a lowering

### Listing 3 – Hypothetical aspect implementing least privilege.

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

### Listing 4 – Improved aspect implementing least privilege.

```java
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");
advice GAFlow(abc): before() {
    SAFER_LEVEL_HANDLE hAuthzLevel;
    // Create a normal user level.
    if (SaferCreateLevel(SAFER_SCOPEID_USER, SAFER_LEVELID_CONSTRAINED, 0, &hAuthzLevel, NULL)){
        // Generate the restricted token that we will use.
        HANDLE hToken = NULL;
        if (SaferComputeTokenFromLevel(hAuthzLevel, NULL, &hToken, 0, NULL)){
            // Sets the restrict token for the current thread
            HANDLE hThread = GetCurrentThread();
            SetThreadToken(&hThread,NULL);
        }
    }
}
```
of privilege around certain operations. It uses restrict tokens and the SAFER API available in Windows XP. This solution injects code before and after each of the corresponding operations, incurring overhead, particularly in the case where the operations a, b and c would be executed consecutively. This could be avoided by using GAFlow and GDFlow, as we show in Listing 4.

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

Listing 5, although correct-looking, can create unwanted side effects if two calls (say, a and b) were intended to be part of the same critical section (i.e., in the same execution path), as the lock would be released after a, and acquired again before b, allowing the execution of another unwanted critical section, possibly damaging b internal state. Improving this aspect in order to handle this case requires foreknowledge of the program event flow, contradicting the core principle of separation of concerns and thus complicating further maintenance activities and preventing aspect reuse. In contrast, by using our proposal, the lock is acquired and released independently of the individual join points while guaranteeing that they will be, altogether, considered as one critical section. Listing 6 shows this improvement.

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

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

- **Ease of use**: Programmers can target places in the application control-flow graph where to inject code before or after a set of join points without needing to manually determine the precise point where to do so.
- **Ease of maintenance**: Programmers can change the program structure without needing to rewrite the associated aspects that were relying on explicit knowledge of the structure in order to pinpoint where the advice code would be injected. For example, if we need to change the execution path to a particular function (e.g., when performing refactoring), we also need to find manually the new common ancestor and/or descendant, whereas this would be done automatically using the proposed pointcuts.
- **Execution time and memory consumption**: Programmers can inject certain pre-operations and post-operations only where needed in the program, without having to resort to injection in the catch-all main. This can improve the apparent responsiveness of the application since certain lengthy operations (such as library initialization) can be avoided if the branches of code requiring them are not executed, thus saving CPU cycles and memory usage. Also, this avoids the execution of the pre-operations and post-operations needed around each targeted join point, which is the default solution using actual AOP techniques. This is replaced by executing them only once around the GAFlow and GDFlow.

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

4.3.1. Securing connection using the current AOP technologies
Securing channels between two communicating parties is the main security solution applied to avoid eavesdropping, tampering with the transmission and/or session hijacking. The Transport Layer Security (TLS) protocol is widely used

<table>
<thead>
<tr>
<th>Listing 5 – Aspect adding atomicity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Semaphore sem = new Semaphore(1);</td>
</tr>
<tr>
<td>pointcut abc: call(&quot;% a(...)&quot;)</td>
</tr>
<tr>
<td>advice abc: before({</td>
</tr>
<tr>
<td>try{</td>
</tr>
<tr>
<td>sem.acquire();</td>
</tr>
<tr>
<td>}catch(InterruptedException e) {/...}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>advice abc: after({</td>
</tr>
<tr>
<td>sem.release();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 6 – Improved aspect adding atomicity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointcut abc: call(&quot;% a(...)&quot;)</td>
</tr>
<tr>
<td>advice GAFlow(abc): before({</td>
</tr>
<tr>
<td>static Semaphore sem = new Semaphore(1);</td>
</tr>
<tr>
<td>try{</td>
</tr>
<tr>
<td>sem.acquire();</td>
</tr>
<tr>
<td>}catch(InterruptedException e) {/...}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>advice GDFlow(abc): after({</td>
</tr>
<tr>
<td>sem.release();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
for this task. We thus present in this section a part of a case study, in which we implemented an AspectC++ aspect that secures a connection using TLS and weaved it with Client/Server applications to secure their connections. To generalize our solution and make it applicable on wide range of applications, we assume that not all the connections are secured, since many programs have different local interprocess communications via sockets. In this case, all the functions responsible of sending and receiving data on the secure channels are replaced by the ones provided by TLS. On the other hand, the other functions that operate on the non-secure channels are kept untouched. Moreover, we addressed also the cases where the connection processes and the functions that send and receive the data are implemented in different components (i.e., different classes, functions, etc.). In Listing 7, we see an excerpt of AspectC++ code allowing to harden a connection.

In Listing 7, the reader will notice the appearance of hardening_sockinfo_t as well as some other related functions, which are underlined for the sake of convenience. These are the data structure and functions that we developed to distinguish between secure and insecure channels and export the parameter between the application components at runtime. We found that one major problem was the passing of parameters between functions that initialize the connection and those that use it for sending and receiving data. In order to avoid sharing memory directly, we opted for a hash table that uses the Berkeley socket number as a key to store and retrieve all the needed information (in our own defined data structure). One additional information that we store is whether the socket is secured or not. In this manner, all calls to a send() are replaced at runtime by the secure sending functions if the the socket is protected. This effort of sharing the parameter has both development and runtime

<table>
<thead>
<tr>
<th>Listing 7 – Excerpt of an AspectC++ aspect hardening connections using GnuTLS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>aspect SecureConnection {</td>
</tr>
<tr>
<td>advice execution(&quot;% main(...)&quot;): around () {</td>
</tr>
<tr>
<td>hardening_sockinfoStorageInit();</td>
</tr>
<tr>
<td>hardening_initGnuTLSSubsystem(NONE);</td>
</tr>
<tr>
<td>tjp-&gt;proceed();</td>
</tr>
<tr>
<td>hardening_deinitGnuTLSSubsystem();</td>
</tr>
<tr>
<td>hardening_sockinfoStorageDeinit();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>advice call(&quot;% connect(...)&quot;): around () {</td>
</tr>
<tr>
<td>//variables declared</td>
</tr>
<tr>
<td>hardening_sockinfo_t socketinfo;</td>
</tr>
<tr>
<td>const int cert_type_priority[Seacord, 2005] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP, 0};</td>
</tr>
<tr>
<td>//Initialize TLS session info</td>
</tr>
<tr>
<td>gnutls_init(&amp;socketinfo.session, GNUTLS_CLIENT);</td>
</tr>
<tr>
<td>gnutls_set_default_priority(socketinfo.session);</td>
</tr>
<tr>
<td>gnutls_certificate_type_set_priority(socketinfo.session, cert_type_priority);</td>
</tr>
<tr>
<td>gnutls_certificate_allocate_credentials(&amp;socketinfo.xcred);</td>
</tr>
<tr>
<td>gnutls_credentials_set(socketinfo.session, GNUTLS_CRD_CERTIFICATE, socketinfo.xcred);</td>
</tr>
<tr>
<td>//Connect</td>
</tr>
<tr>
<td>tjp-&gt;proceed();</td>
</tr>
<tr>
<td>if(*tjp-&gt;result()&lt;0) {perror(&quot;cannot connect &quot;); exit(1);}</td>
</tr>
<tr>
<td>//Save the needed parameters and the information that distinguishes between secure and non-secure channels</td>
</tr>
<tr>
<td>socketinfo.isSecure = true;</td>
</tr>
<tr>
<td>socketinfo.socketDescriptor = *(int *)tjp-&gt;arg0;</td>
</tr>
<tr>
<td>hardening_storeSocketInfo(*(int *)tjp-&gt;arg0, socketinfo);</td>
</tr>
<tr>
<td>//TLS handshake</td>
</tr>
<tr>
<td>gnutls_transport_set_ptr(socketinfo.session, (gnutls_transport_ptr)(*(int *)tjp-&gt;arg0));</td>
</tr>
<tr>
<td>*tjp-&gt;result() = gnutls_handshake(socketinfo.session);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>//replacing send() by gnutls_record_send() on a secured socket</td>
</tr>
<tr>
<td>advice call(&quot;% send(...)&quot;): around () {</td>
</tr>
<tr>
<td>//Retrieve the needed parameters and the information that distinguishes between secure and non-secure channels</td>
</tr>
<tr>
<td>hardening_sockinfo_t socketinfo;</td>
</tr>
<tr>
<td>socketinfo = hardening_getSocketInfo(*(int *)tjp-&gt;arg0);</td>
</tr>
<tr>
<td>//Check if the channel, on which the send function operates, is secured or not</td>
</tr>
<tr>
<td>if(socketinfo.isSecure)</td>
</tr>
<tr>
<td>if(the channel is secured, replace the send by gnutls_send <em>(tjp-&gt;result()) = gnutls_record_send(socketinfo.session,</em>(char*)</td>
</tr>
<tr>
<td>tjp-&gt;arg1), *(int *)tjp-&gt;arg2);</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>tjp-&gt;proceed();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
overhead that could be avoided by the use of a primitive automating the transfer of concern-specific data within advices without increasing software complexity. Further, other experiments with another security feature (encrypting sensitive memory) showed that the use of hash table could not be generalized.

4.3.2. Need to features for passing parameters

Our study of the literature and our previous experiments (Mourad et al., 2007a,b) showed that in order to perform security hardening, it is often necessary to pass state information from one part of the program to another. For instance, in the example provided in Listing 7, we need to pass the gnutls_session_t data structure from the advice around connect to the advice around send in order to properly harden the connection. The current AOP models do not allow to perform such operations. To address this limitation, we integrated additional modules and data structures and changed some functions within the application in order to pass the parameters. In the case of large scale applications with multiple features and complex dependencies and relations between their components, such solution is not realistic. It requires many complex re-engineering actions to be performed since any changes in one component lead to apply several modifications in all its dependent ones.

4.3.3. Securing connection using ExportParameter and ImportParameter

We modified the example of Listing 7 by using the proposed approach for parameter passing. An excerpt of the new code is presented in Listing 8. All the data structure and algorithms (underlined in Listing 7) are removed and replaced by the primitives for exporting and importing. An ExportParameter for the parameters session and xcred is added on the declaration of the advice of the pointcut that identifies the function connect. On the other side, an ImportParameter for the parameter session is added on the declaration of the advice of the pointcut that identifies the function send.

5. Methodologies, algorithms and implementation

This section presents the elaborated methodologies and algorithms for dominator set, graph labeling, GAFlow, GDFlow, ExportParameter and ImportParameter. Algorithms that operate on CFG have been developed for decades now, and many graph operations are considered to be common knowledge in computer science. Despite this theoretical richness, we are not aware of existing methods allowing to determine the GAFlow or GDFlow node for a particular set of nodes (i.e., join

**Listing 8 – Hardening of connections using GnuTLS and parameter passing.**

```java
aspect SecureConnection {
    advice execution ("% main(...) "): around () {
        gnutls_global_init ();
        tjp->proceed();
        gnutls_global_deinit();
    }
    advice call("% connect(...) "): around (): ExportParameter(gnutls_session session, gnutls_certificate_credentials xcred){
        //variables declared
        static const int cert_type_priority(Seacord, 2005) = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP, 0};
        //initialize TLS session info
        gnutls_init (&session, GNUTLS_CLIENT);
        gnutls_set_default_priority (session);
        gnutls_certificate_type_set_priority (session, cert_type_priority);
        gnutls_certificate_allocate_credentials (&xcred);
        gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);
        //Connect
        tjp->proceed();
        if(*tjp->result() < 0) {perror("cannot connect "); exit(1);} //TLS handshake
        gnutls_transport_set_ptr (session, (gnutls_transport_ptr) (*(int *)tjp->arg(0)));
        *tjp->result() = gnutls_handshake (session);
    }
    //replacing send() by gnutls_record_send() on a secured socket
    advice call("% send(...) "): around (): ImportParameter(gnutls_session session){
        //Check if the channel, on which the send function operates, is secured or not
        if (session! = NULL) {
            //if the channel is secured, replace the send by gnutls_send
            *tjp->result() = gnutls_record_send("session", (char*) tjp->arg(1), (int *)tjp->arg(2));
            else
            tjp->proceed();
        }
    }
}
```
points) in a CFG by considering all the possible paths. On the other hand, the algorithms used to calculate the Dominator and Post-Dominator sets of a CFG node consider such criteria, so they can be extended and used to build the algorithms of GAFlow and GDFlow.

In this context, we propose two different sets of algorithms for GAFlow and GDFlow. The first set is based on the Dominator and Post-Dominator algorithms of classical CFG, while the second one operates on labeled graph (i.e., a label is associated to each node). Choosing between these algorithms is considered only during the implementation phase and left for the developers. We assume that the CFG is shaped in the traditional form, with a single start node and a single end node. In the case of program with multiple starting points, we consider each starting point as a different program in our analysis. Most of these assumptions have been used so far (Gomez, 2003). With these statements in place, we ensure that our algorithms will return a result (in the worst case, the start node or the end node) and that this result will be a single and unique node for all inputs.

5.1. GAFlow and GDFlow using dominator and post-dominator

The problem of finding the dominators in a control-flow graph has a long history in the literature and many algorithms have been proposed, improved and implemented (Cooper et al.; Holloway and Smith, 1998). To compute dominance information, as presented in Cooper et al., the compiler can annotate each node in the CFG with a set \( \text{DOM}(n) \) containing every node \( n \) that dominates \( b \), including \( b \). The dominators of a node \( n \) are given by the maximal solution to the following data-flow equations:

\[
\text{DOM}(\text{entry}) = \{ \text{entry} \} \tag{1}
\]

\[
\text{DOM}(n) = \left( \cap_{p \in \text{preds}(n)} \text{DOM}(p) \right) \cup \{ n \} \tag{2}
\]

where \( \text{entry} \) is the start node and \( \text{preds}(n) \) is the set of all the predecessors of \( n \). The dominator of the start node is the start node itself. The dominators for any other node \( n \) is the intersection of the set of dominators for all predecessors \( p \) of \( n \). The node \( n \) is also in the set of dominators for \( n \). To solve these equations, the iterative algorithm presented in Cooper et al. can be used.

We elaborated and implemented Algorithm 1 to calculate the Dominator set. It is based on the mechanisms identifying the possible paths reaching one destination from one source node (Dijkstra). However, other available algorithms, compilers and/or tools that give similar results can be useful (Cooper et al.; Holloway and Smith, 1998). This choice is completely left for the developer who is expert in such domain.

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

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

**Algorithm 1. (Algorithm to determine the dominator set.)**

1. Set \( \text{DOM}(\text{Node entryPointNode}, \text{Node n}) \)
2. mark(entryPointNode, n);
3. Stack pathList = new Stack();
4. tracePath(entryPointNode, n, new Stack(), pathList);
5. Set meetSet = {};
6. if pathList.isEmpty() then
7. meetSet.addAll(pathList.pop());
8. for each path \( p \) in pathList do
9. meetSet = meetSet \( \cap \) (Set)\( p \)
10. end for
11. end if
12. return meetSet;
13. markNode(Node targetNode, Node currentNode, Node branchingNode, int markIndex)
14. if \( \exists \) currentNode.pathMarkingMap.get(branchingNode) then
15. currentNode.pathMarkingMap.put(branchingNode, markIndex);
16. if currentNode = targetNode then
17. markIndex = markIndex + 1;
18. for each parent \( p \) in currentNode.parentList do
19. markNode(targetNode, p, currentNode, markIndex);
20. end for
21. end if
22. end if
23. tracePath(Node targetNode, Node currentNode, Stack ascendTimeList, Stack pathList)
24. ascendTimeList.push(currentNode);
25. if currentNode = = targetNode then
26. List path = new List();
27. path.addAll(ascendTimeList);
28. pathList.addAll(path);
29. else
for each parent p in currentNode.parentList do
if p.pathMarkingMap.get(currentNode) then
    pathMarkValue = p.pathMarkingMap.get(currentNode);
    for each markingKey in currentNode.pathMarkingMap.keySet() do
        markingValue = currentNode.pathMarkingMap.get(markingKey);
        if markingValue + 1 = pathMarkValue then
            currentNode.pathMarkingMap.remove(markingKey);
            tracePath(targetNode, p, ascendList, pathList);
            currentNode.pathMarkingMap.put(markingKey, markingValue);
            break;
        end if
    end for
end if
end for
end if
ascendList.pop();

A simple method to calculate the post-dominator sets is to reverse the edge direction of the CFG, start from the exit node and apply the dominator algorithm (Holloway and Smith, 1998). The post-dominator of the exit node is the exit node itself. In the case of multiple end points, we consider each ending point as different program in our analysis (in fact, each ending point will be a starting point after applying the CFG reverse edge direction mechanism).

5.1.1. Pointcut GAFlow
In order to compute the GAFlow, we developed a mechanism built on top of the dominator algorithm. We calculate the common dominator set of all the selected nodes specified in the parameter of GAFlow. The calculated set is ordered by the dominance relation. The last node in this ordered set will be returned by Algorithm 2 as the closest guaranteed ancestor.

Algorithm 2. (Algorithm to determine GAFlow using dominator.)
Require: SelectedNodes is initialized with the contents of the pointcut match
1: GAFlow(NodeSet SelectedNodes):
2: CommonDomSet = AllNodes
3: for all node e SelectedNodes do
4:     CommonDomSet = CommonDomSet \ (DOM(node))
5: end for
6: return GetLastNode(OrderedCommonDomSet)

5.1.2. Pointcut GDFlow
The closest guaranteed descendant is determined by elaborating a mechanism built on top of the post-dominator algorithms. We calculate the common post-dominator set of all the selected nodes specified in the parameter of GDFlow. The calculated set is ordered by the dominance relation. The first node in this ordered set will be returned by Algorithm 3 as the closest guaranteed descendant.

Algorithm 3. (Algorithm to determine GDFlow using post-dominator.)
Require: SelectedNodes is initialized with the contents of the pointcut match
1: GDFlow(NodeSet SelectedNodes):
2: CommonPostDomSet = AllNodes
3: for all node e SelectedNodes do
4:     CommonPostDomSet = CommonPostDomSet \ (DOM(node))
5: end for
6: return GetLastNode(OrderedCommonPostDomSet)

Table 2 – Part1 – results of the execution of Algorithm 2 on the graph of Fig. 3.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>Common dominator set</th>
<th>GAFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>N2, N7</td>
<td>Entry, N1</td>
<td>N1</td>
</tr>
<tr>
<td>N5, N6</td>
<td>Entry, N1, N2</td>
<td>N2</td>
</tr>
<tr>
<td>N4, N6, N10</td>
<td>Entry, N1</td>
<td>N1</td>
</tr>
<tr>
<td>N8, N9</td>
<td>Entry, N1</td>
<td>N1</td>
</tr>
</tbody>
</table>

Table 3 – Part2 – results of the execution of Algorithm 2 on the graph of Fig. 3.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>N4, N6, N10</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM(N4)</td>
<td>Entry, N1</td>
</tr>
<tr>
<td>DOM(N6)</td>
<td>Entry, N1, N2</td>
</tr>
<tr>
<td>DOM(N10)</td>
<td>Entry, N1, N9</td>
</tr>
<tr>
<td>Common dominator set (N4, N6, N10)</td>
<td>Entry, N1</td>
</tr>
<tr>
<td>GAFlow</td>
<td>N1</td>
</tr>
</tbody>
</table>
5.1.2.1. Case study. Similarly to Algorithm 2, we implemented Algorithm 3 by reversing the edge direction of the CFG, starting from the exit node and applying Algorithm 1 to calculate the post-dominator set of a particular node (Holloway and Smith, 1998). Then, we applied this implementation on the case study illustrated in Fig. 3. The result of calculating the GDFlow for a set of selected nodes is illustrated in Tables 4 and 5.

5.2. GAFlow and GDFlow using labeled graph

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

Each node down the hierarchy is labeled in the same manner as the table of contents of a book (e.g., 1., 1.1., 1.2., 1.2.1., …), as depicted by Algorithm 4, where the operator +c denotes string concatenation (with implicit operand type conversion). To that effect, the labeling is done by executing Algorithm 4 on the start node with label “0.”, thus recursively labeling all nodes.

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

5.2.1. Pointcut GAFlow

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

5.2.1.1. Case study. We implemented Algorithm 5 and we applied it on the labeled graph of Fig. 4. We selected, as case study, some nodes in the graph with various combinations. Our results, are summarized in Table 6 and Fig. 5.

Algorithm 4. (Hierarchical graph labeling algorithm.)

1: labelNode(Node s, Label l):
2: s.labels ← s.labels ∪ {l}

### Table 4 – Part1 – results of the execution of Algorithm 3 on the graph of Fig. 3.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>Common post-dominator set</th>
<th>GDFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>N2, N7</td>
<td>N9, N10, exit</td>
<td>N9</td>
</tr>
<tr>
<td>N4, N5, N6</td>
<td>N9, N10, exit</td>
<td>N9</td>
</tr>
<tr>
<td>N6, N7</td>
<td>N8, N9, N10, exit</td>
<td>N8</td>
</tr>
<tr>
<td>N8, N9</td>
<td>N10, exit</td>
<td>N10</td>
</tr>
</tbody>
</table>

### Table 5 – Part2 – results of the execution of Algorithm 3 on the graph of Fig. 5.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>Common post-dominator set</th>
<th>GDFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>N4, N5, N6</td>
<td>N7, N8, N9, N10, exit</td>
<td>N9</td>
</tr>
<tr>
<td>N6, N7</td>
<td>N8, N9, N10, exit</td>
<td>N8</td>
</tr>
<tr>
<td>N8, N9</td>
<td>N9, N10, exit</td>
<td>N9</td>
</tr>
</tbody>
</table>

**Fig. 4 – Sample labeled graph.**
Table 6 – Results of the execution of Algorithm 5 on the labeled graph of Fig. 4.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>GAFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>N2, N7</td>
<td>N1</td>
</tr>
<tr>
<td>N5, N6</td>
<td>N2</td>
</tr>
<tr>
<td>N4, N6, N10</td>
<td>N1</td>
</tr>
<tr>
<td>N8, N9</td>
<td>N1</td>
</tr>
</tbody>
</table>

25: current.append(l.charAt(i))
26: if Label1.charAt(i) = '.' then
27: labels.add(current.clone())
28: end if
29: end for

5.2.2. Pointcut GDFlow
The same mechanism for reversing the edge direction of the CFG (Holloway and Smith, 1998), that calculates the post-dominator set by using the dominator algorithm, can also be applied to determine the closest guaranteed descendant on a labeled graph (see Section 5.1 for more detail). Once the edges’ directions are reversed, labeling the CFG can be performed and then the same algorithm used for calculating the GAFlow (Algorithm 5) can be applied to determine the GDFlow.

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

5.2.2.1. Case study. We used the same implementation of Algorithm 4 and case study illustrated in Fig. 4. Then, we applied the aforementioned mechanism and implemented Algorithm 5 to calculate the GDFlow for the selected nodes. Table 7 contains the results along with Fig. 6.

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

Algorithm 7 allows parameter passing between two nodes of the context-insensitive call graph of a program (Grove and Chambers, 2001), with each node representing a function and each arrow representing a call site. To ensure the declaration and initialization of the passed parameter all the time, whatever the selected execution path, we elaborated on top of

Table 7 – Results of the execution of reverse edge direction and Algorithm 5 on the labeled graph of Fig. 4.

<table>
<thead>
<tr>
<th>Selected nodes</th>
<th>GDFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>N2, N7</td>
<td>N9</td>
</tr>
<tr>
<td>N4, N5, N6</td>
<td>N9</td>
</tr>
<tr>
<td>N6, N7</td>
<td>N8</td>
</tr>
<tr>
<td>N8, N9</td>
<td>N10</td>
</tr>
</tbody>
</table>
this algorithm a mechanism based on the GAFlow. This mechanism exports the parameter over all the possible paths going from the origin to the destination nodes.

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

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

*Fig. 7* shows an illustration of Algorithm 6 on a call graph example. To pass the parameter from h to g, their GAFlow, which is b in this case, is first identified. Afterwards, the parameter is passed over all the paths from h to b, then from b to g again over all the paths.

Algorithm 6. (Algorithm to pass the parameter between two pointcuts)
1: function passParameter(Node origin, Node end, Parameter param):
2: if origin = destination then
3: return success
4: end if
5:start ← GuaranteedAncestor(origin, end )
6:passP aramOnBranch(start, origin, param)
7:node.addLocalV ariable(param)
8:passP aramOnBranch(start, end, param)

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

![Fig. 6 – Excerpt of labeled graph illustrating the GDFlow of N2 and N7.](image-url)
optimal in the presence of loops, it modifies all the callers only one time and keeps track of the modified nodes.

### 5.3.1 Case study

We implemented a program similar to the scenario of the call graph illustrated in Fig. 7. This program presented in Listing 9 is essentially a client application that establishes a connection, sends a request and receives a response from the server. Then, we simulated the execution of the proposed primitives’ algorithms and applied manually the aspects presented in Listing 8 on this application in order to secure its communication channels, producing the programme in Listing 10. We successfully tested the correctness of the hardened applications with SSL enabled web server by capturing the exchange of data packets, demonstrating that the communication was effectively encrypted. The practical implementation of the primitives’ algorithms is still in progress.

**Algorithm 7.** (Algorithm to pass a parameter between two nodes of a call graph)

1. function passParamOnBranch(Node origin, Node destination, Parameter param):
2. if (origin = destination) then
3. return success

---

**Listing 10 – Resulting hardened program.**

```c
const char * HTTPrequest = "GET/HTTP/1.1 \nHost: localhost\n\n";
int dosend(int sd, char * buffer, unsigned int bufSize, gnutls_session_t * session) {
    if (session != NULL) return gnutls_record_send(session, buffer, bufSize);
    else return send(sd, buffer, bufSize, 0);
}

int doreceive(int sd, char * buffer, unsigned int bufSize, gnutls_session_t * session) {
    if (session != NULL) return gnutls_record_recv(session, buffer, bufSize);
    else return recv(sd, buffer, bufSize, 0);
}

int doConnect(int sd, struct sockaddr_in servAddr, gnutls_session_t * session, gnutls_certificate_credentials_t * xcred) {
    static const int cert_type_priority(Seacord, 2005) = { GNUTLS_CRT_X509,
                                                        GNUTLS_CRT_OPENPGP, 0};
    int rc;
    gnutls_init (session, GNUTLS_CLIENT);
    gnutls_set_default_priority (*session);
    gnutls_certificate_type_set_priority (*session, cert_type_priority);
    gnutls_certificate_allocate_credentials (xcred);
    gnutls_credentials_set (*session, GNUTLS_CRD_CERTIFICATE, *xcred);
    rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
    if (rc > 0) {
        gnutls_transport_set_ptr (*session, (gnutls_transport_ptr) sd);
        rc = gnutls_handshake (*session);
    }
    return rc;
}

int main (int argc, char *argv[]) {
    gnutls_global_init ();
    /*...*/
    /* create socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd < 0) {
        perror("cannot open socket");
        exit(1);
    }
    doConnect(sd, servAddr, &session, &xcred);
    dosend(sd, HTTPrequest, strlen(HTTPrequest) + 1, &session);
    fprintf(stderr,"Sent %u characters:\n%s\n", rc, HTTPrequest);
    memset((void *)buf, 0, MAX_MSG);
    doreceive(sd, buf, MAX_MSG, &session);
    fprintf(stderr,"Received %u characters:\n%s", rc, buf);
    /* Shutdown */
    close(sd);
    gnutls_bye(session, GNUTLS_SHUT_RDWR);
    gnutls_deinit(session);
    gnutls_certificate_free_credentials(xcred);
    gnutls_global_deinit();
    return 0;
}
```
AOP is a very promising paradigm for software security hardening. However, this technology was not initially designed to address security issues and many research initiatives showed its limitations in such domain. Similarly, we explored in this paper the shortcomings of the AOP in applying many security hardening practices and the need to extend this technology with new pointcuts and primitives. In this context, we proposed new pointcuts and primitives to AOP for security hardening concerns: GAFlow, GDFlow, ExportParameter and ImportParameter. The GAFlow returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The GDFlow returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. The two primitives pass parameters from one advice to the other through the programs’ call graph. We explored the viability of the proposed pointcuts and primitives by (1) exploring their advantages for security hardening, (2) developing their corresponding algorithms and (3) presenting the result of explanatory case studies.

Concerning our future work, we are currently working on implementing and deploying the proposed construct into AspectJ and AspectC++ weavers. Moreover, we are going to address other limitations of AOP for security.

Fig. 7 – Parameter passing in a call graph.

6. Conclusion

AOP is a very promising paradigm for software security hardening. However, this technology was not initially...
Kim H. An AOSD implementation for C#, tech. rep. TCD-CS2002-
Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. 
Mourad, Laverdière MA, Debbabi M. Security hardening of open 
Kiczales G. The fun has just begun, keynote talk at AOSD 2003. 
Huang M, Wang C, Zhang L. Toward a reusable and generic 
Howard M, Lipner S. The security development lifecycle. 
Holloway G, Smith MD. The Machine-SUIF control-flow analysis 
Harbulot JR, Gurd A. Join point for loops in AspectJ. In: 
Grove D, Chambers C. A framework for call graph construction 
Gomez E. CS624-notes on control flow graph. Available at:http://
Navarro LDB, Sudholt M, Vanderperren W, Fraine BD, Suvée D. 
Myers A. Jflow: practical mostly-static information flow control. 
Mourad A, Laverdière MA, Debbabi M. Security crossing concerns and 
Mourad A, Laverdière MA, Debbabi M. Security hardening of open 
Mourad A, Laverdière MA, Debbabi M. A high-level aspect- 
Mourad A, Laverdière MA, Debbabi M. An aspect oriented 
Mourad A, Laverdière MA, Debbabi M. An aspect oriented 
Mourad Debbabi is a Full Professor and the Director of the Concordia Institute for Information Systems Engineering at Concordia University. He is also a Concordia Research Chair Tier I in Information Systems Security. He holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay University, France. He published more than 140 research papers in international journals and conferences on IT security, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory, specification and verification of safety-critical systems and formal semantics. In the
past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Corporate Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. Nowadays, he is leading two major projects on cyber forensics and open source security.