New AOP Pointcuts and Primitives 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.

Key words: Software Security, Security Hardening, Aspect-Oriented Programming, Control Flow Graph, Pointcut, Hierarchical Graph Labeling, Dominators

Email addresses: mourad@ciise.concordia.ca (Azzam Mourad),
a_soeanu@ciise.concordia.ca (Andrei Soeanu),
ma_laver@ciise.concordia.ca (Marc-André Laverdière),
debbabi@ciise.concordia.ca (Mourad Debbabi).

[1] This research is the result of a fruitful collaboration between CSL (Computer Security Laboratory) of Concordia University, DRDC (Defence Research and De-

1 Motivations & Background

Security is taking an increasingly predominant role in today's computing world. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. The challenge is even greater when legacy systems must be adapted to networked/web environments, while they are not originally designed to fit into such high-risk environments. Tools and guidelines have been available for developers for few years already, but their practical adoption is limited so far. Software maintainers must face the challenge to improve program security and are often 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.

Few concepts emerged in the literature in order to help and guide developers to harden security into software. 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 [1–5].

However, AOP was not initially designed to address security issues, which resulted in many shortcomings in the current technologies [6–10]. We were not able to apply some security hardening activities due to missing features. Such limitations forced us, when applying security hardening practices, to perform programming gymnastics (when is possible), resulting in additional modules that must be integrated within the application, at a definitive runtime, memory and development cost. As a result, the specification of new security-related primitives and pointcuts is becoming a very challenging and interesting domain of research.

In this context, we propose in this paper new AOP pointcuts and primitives

velopment Canada) Valcartier and Bell Canada under the NSERC DND Research Partnership Program.

that are needed for security hardening. The proposed pointcuts, *GAFlow* and *GDFlow*, which are introduced first in [11], allow to identify join points in a program's control flow graph (CFG). The proposed primitives, *ExportParameter* and *ImportParameter*, extend the current AOP weaving capabilities and allow to pass parameters from one pointcut to another through the programs' context-insensitive call graph. These features are necessary to develop many security hardening solutions, and, to the best of our knowledge, none of the existing pointcuts and primtives can provide their functionalities. Moreover, we combined all the deployed and proposed pointcuts in the literature and were not able to find a method that would isolate a single node in our CFG that satisfies the criteria we define for *GAFlow* and *GDFlow*.

This paper provides our new contributions toward developing a security hardening framework. This framework is based on AOP, thus addressing the shortcomings 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 remainder of this paper is organized as follows. Section 2 explores the appropriateness of AOP for securing software as well as the shortcomings associated with the current AOP technologies. 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. In Section 5, the algorithms necessary for implementing the proposed pointcuts and primitives are presented. This section also shows the implementation results into case studies. We move on to the related work in Section 6, and then we present some summarizing conclusions in Section 7.

2 Securing Software Using AOP

Many contributions that discuss and explore the useability and relevance of AOP for integrating security code into software have been published recently [1–4,12]. We presented an overview on them in [13]. In this section, we first introduce AOP and its advice-pointcut model, then summarize briefly our elaborated AOP-based approach that explores the relevance of AOP for software securing hardening and also distinguishes the shortcomings of the current AOP technologies for some security issues.

2.1 AOP / Advice-Poincut Model

Aspect-Oriented Programming is a family of approaches that allow to integrate different concerns into useable software in a manner that is not possible using the classical object-oriented decomposition. The foundation of AOP is the

principle of "Separation of Concerns", where issues that affect and crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed such as AspectJ [14], AspectC [15], AspectC++ [16], AspectC# [17] and an AOP version targeting the Smalltalk programming language [18]. The approach adopted by most of these initiatives is called the Pointcut-Advice model. The join points, pointcuts and advices constitute its main elements.

To develop under this paradigm, one must first determine what code needs to be injected into the basic model. This code describes the behavior of the issues that affect and crosscut the application. Each atomic unit of code to be injected is called an advice. Then, it is necessary to 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's join points for which the advice will be injected. A join point is an identifiable execution point in the application's code and the pointcut constitutes the constructor that designates a set of join points. The pointcut expressions typically allow 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. Examples of implemented aspects are presented in Section 4. Finally, the aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

2.2 AOP-Based Security Hardening Approach

In our prior work [19], we defined software security hardening as *any process, methodology, product or combination thereof that is used to add security functionalities and/or remove vulnerabilities or prevent their exploitation in existing software*. Security hardening practices are usually applied manually by injecting security code into the software [20–23]. This task entails that the developers have high security expertise and deep knowledge of the inner working of the software code, which is not available all the time. In this context, we elaborated in [24,13] an approach based on aspect orientation to perform security hardening in a systematic way. The approach architecture are illustrated in Figure 1.

Each component participates by playing a role and/or providing functionalities in order to have a complete security hardening process. The developer is the person responsible of writing plans by deriving them from the security requirements. These plans contain the abstract actions required for security hardening and uses the security hardening patterns that are developed by

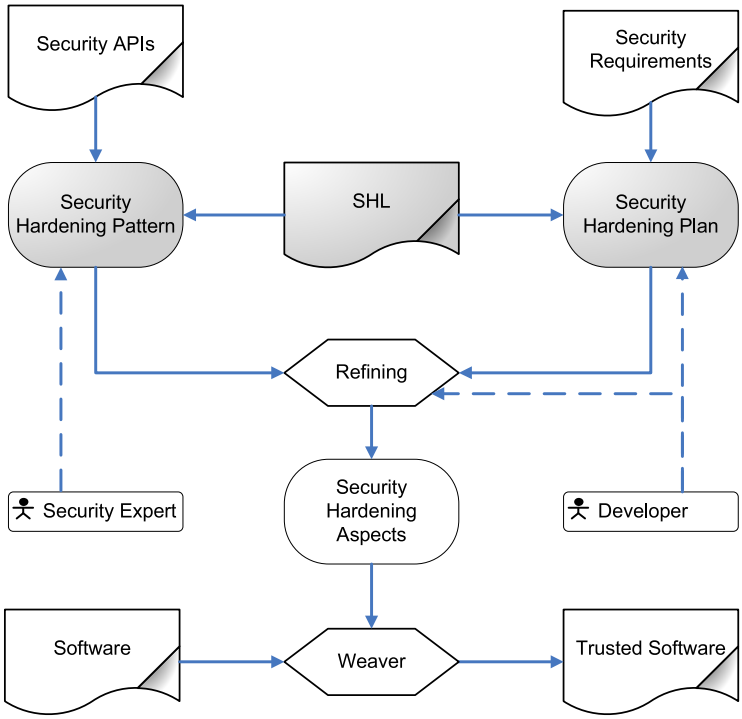


Fig. 1. Schema of Our Approach

security experts and provided in catalogs. The security APIs constitute the building blocks used by the patterns to achieve the desired solutions. The *SHL* language, proposed in [24], is used to define and specify the security hardening plans and patterns.

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

The abstraction of the hardening plans is bridged by concrete steps defined in the hardening patterns using also *SHL*. This dedicated language, together with a well-defined template that instantiates the patterns with the plan's given parameters, allow to specify the precise steps to be performed for the hardening, taking into consideration technological issues such as platforms, libraries and languages. We built *SHL* on top of the current AOP languages

because we believe, after a deep investigation on the nature of security hardening practices and the experimental results we got, that aspect orientation is the most natural and appealing approach to reach our goal.

Once the security hardening solutions are built, the refinement of the solutions into aspects or low level code can be performed using a tool or by programmers that do not need to have any security expertise. Afterwards, an AOP weaver (e.g. AspectJ, AspectC++) can be executed to harden the aspects into the original source code, which can now be inspected for correctness. As a result, the approach constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them, and allows the software engineers to use these solutions to harden software by specifying and developing high level security hardening plans. We illustrated the feasibility of the whole approach by applying it during the elaboration and development of several security hardening solutions to secure connections in client-server applications, add access control features to a program, encrypt memory contents for protection and remedy some low-level security issues in C programs.

2.3 Shortcomings of AOP for Security

Our experimental results presented in [24,13] showed the usefulness of AOP in reaching the objective of having systematic security hardening. On the other hand, we have also distinguished, together with many previous work [6–10] (see Section 6), the shortcomings of the available AOP technologies for many security issues. Some hardening activities cannot be implemented at all, or can only be implemented by increasing the software complexity. We identify in the following some of the missing features in the current AOP languages:

- *Parameter Passing:* It should be possible to pass parameter between advices in a manner that is simple, safe, and does not increase code complexity
- *Just in time library initialization:* It should be possible to have the libraries used by the patterns initialized only when they are needed, and not in the main function or class
- *Parameter name pointcut complement:* It should be possible to complement pointcut over function calls or execution with a selector based on the names of the variables passed.

The first two shortcomings are well illustrated in the aspects presented in Listing 7, where we see the code injected around the `main` function and the use of a hash table to pass parameters. The functions of interest are underlined. Such limitations also prevent us from applying many other security hardening solutions. In the scope of this paper, we address the identified problems and

improve the current AOP technologies, and hence our approach, by elaborating new pointcuts and primitives needed for security hardening concerns.

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 our proposition.

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

parameter ::= <type> <identifier>
paramList ::= parameter [,paramList]
e ::= ExportParameter(<paramList>)
i ::= ImportParameter(<paramList>)
advice <p> : (before|after|around) (<arguments>)
[: e | i | e,i] {<advice-body>}
```

Table 1
Syntax of the Pointcuts and Primitives

A function signature is denoted by *s*. The *GAFlow* and the *GDFlow* are the new control flow based pointcuts. Their parameters are also pointcuts. The new *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

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

3.2 GDFlow

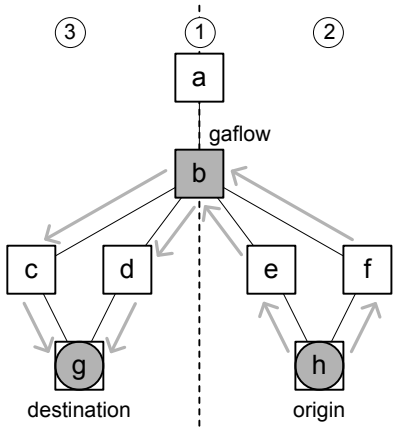
The *GDFlow* pointcut operates on the CFG of a program. Its input is a set of join points defined as a pointcut and its output is a join point. In other words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. This output (1) is the common descendant of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

3.3 ExportParameter and ImportParameter

The two primitives *ExportParameter* and *ImportParameter* should always be combined and used together in order to provide the information needed for parameter passing from one joint point to another. The proposed approach for parameter passing operates on the context-insensitive call graph of a program [25], with each node representing a function and each arrow representing call site. It exports the parameter over all the possible paths going from the origin to the destination nodes. This is achieved by performing the following three steps: (1) Calculating the closest guaranteed ancestor (*GAFlow*) of the origin and destination join points, (2) passing the parameter from the origin to the aforementioned *GAFLow* and then (3) from this *GAFlow* to the destination. The AOP primitives that are responsible of passing the parameters 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.

By passing the parameter from the origin to *GAFlow* and then to the destination, we ensure that the parameter will be effectively passed through all the possible execution paths between the two join points. Otherwise, the parameter could not be passed or passed without initialization, which would create software errors and affect the correctness of the solution. Figure 2 illustrates our approach. For instance, 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.

The proposed parameter passing capability changes the function signatures and the relevant call sites in order to propagate useful security hardening variables via `inout` function parameters. It changes the signatures of all the functions involved in the call graph between the exporting and importing join points. All calls to these functions are modified to pass the parameter as is,



1: Identify GAFlow 2: Origin → GAFlow 3: GAFlow → destination

Fig. 2. Parameter Passing in a Call Graph

in the case of the functions involved in this transmission path (e.g. nodes b, c,d,e,f).

4 Discussion

This section discusses the usefulness, advantages and limitations of our proposition.

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 [20–23]. Examples of such cases would be the injection of security library initialization/deinitialization, privilege level changes, atomicity guarantee, logging, etc. The current AOP models allow us only to identify a set join points in the program, and therefore inject code before, after and/or around each one of them. However, to the best of our knowledge, none of the current pointcuts enable the identification of a join point common to a set of other join points where we can efficiently inject the code 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

In the case of security library initialization (e.g. access control, authorization, cryptography, etc.), our pointcuts allow us to initialize the needed library 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 and deinitialization, as depicted in Listing 1. Another possible solution could be the loading and unloading of the library before and after its use, which may cause runtime problems since API-specific data structures could be needed for other functions. However, in the case of large applications, especially for embedded ones, the two solutions may not work and create an accumulation of code injection statements that would create a significant, and possibly damaging, waste of system resources. In listing 2, we see an improved aspect that solves such problem and can yield a more efficient and wider applicable result using the proposed pointcuts.

Listing 1. Excerpt of Hardening Aspect for Securing Connections Using GnuTLS

```
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 GnuTLS

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

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

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 [21]) shows an

aspect implementing a lowering of privilege around certain operations. It uses restrict tokens and the SAFER API available in Windows XP. This solution injects code before and after each of the corresponding operations, incurring overhead, particularly in the case where the operations a, b and c would be executed consecutively. This could be avoided by using *GAFlow* and *GDFlow*, as we show in Listing 4.

Listing 3. Hypothetical Aspect Implementing Least Privilege

```
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

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

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

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. Aspect Adding Atomicity

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

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

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

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

Listing 5, 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 for the execution of another unwanted critical section, possibly damaging b's internal state. Improving this aspect in order to handle this case requires foreknowledge of the program's 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.

Listing 6. Improved Aspect Adding Atomicity

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

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

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

4.1.4 Logging

It is possible that a set of operations are of interest for logging purposes, but that their individual log entry 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's 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.
- ***Optimization***: 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. 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*.
- ***Raising the Abstraction Level***: Programmers can develop more abstract and reusable aspect libraries.

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 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 a widely used protocols 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.

Listing 7. Excerpt of an AspectC++ Aspect Hardening Connections Using GnuTLS

```
aspect SecureConnection {

 advice execution ("% main(...)") : around () {
   hardening_socketInfoStorageInit();
   hardening_initGnuTLSSubsystem(NONE);

   tjp->proceed();

   hardening_deinitGnuTLSSubsystem();
   hardening_socketInfoStorageDeinit();
 }

 advice call("% connect(...)") : around () {

   //variables declared
   hardening_sockinfo_t socketInfo;
   const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP,
       0};

   //initialize TLS session info
   gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
   gnutls_set_default_priority (socketInfo.session);
   gnutls_certificate_type_set_priority (socketInfo.session,
       cert_type_priority);
   gnutls_certificate_allocate_credentials (&socketInfo.xcred);
   gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
       socketInfo.xcred);

   //Connect
   tjp->proceed();
   if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

   //Save the needed parameters and the information that distinguishes
       between secure and non-secure channels
   socketInfo.isSecure = true;
   socketInfo.socketDescriptor=*(int *)tjp->arg(0);
   hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
   //TLS handshake
   gnutls_transport_set_ptr(socketInfo.session, (gnutls_transport_ptr)(*(int
        *)tjp->arg(0)));
   *tjp->result() = gnutls_handshake (socketInfo.session);
 }

 //replacing send() by gnutls_record_send() on a secured socket
```

```
  advice call("% send(...)") : around () {

    //Retrieve the needed parameters and the information that distinguishes
        between secure and non-secure channels
    hardening_sockinfo_t socketInfo;
    socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

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

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's 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 using shared 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()` or `recv()` are modified for a runtime check that uses the proper sending/receiving function. This effort of sharing the parameter has both development and runtime 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 Pass Parameters

Our study of the literature and our previous work [24,13] showed that it is often necessary to pass state information from one part to another of the program in order to perform security hardening. For instance, in the example provided in Listing 7, we need to pass the `gnutls_session_t` data structure from the advice around `connect` to the advice around `send`, `receive` and/or `close` in order to properly harden the connection. The current AOP models do not allow to perform such operations.

4.3.3 Securing Connection using ExportParameter and ImportParameter

We modified the example of Listing 7 by using the proposed approach for parameter passing. Listing 8 presents excerpt of the new code. 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*. Moreover, an *ImportParameter* for the parameter *session* is added on the declaration of the advice of the pointcut that identifies the function *send*.

Listing 8. Hardening of Connections using GnuTLS and Parameter Passing

```
aspect SecureConnection {

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

  advice call("% connect(...)") : around () : ExportParameter(gnutls_session
      session, gnutls_certificate_credentials xcred){
    //variables declared
    static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
        GNUTLS_CRT_OPENPGP, 0};

    //initialize TLS session info
    gnutls_init (&session, GNUTLS_CLIENT);
    gnutls_set_default_priority (session);
    gnutls_certificate_type_set_priority (session, cert_type_priority);
    gnutls_certificate_allocate_credentials (&xcred);
    gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

    //Connect
    tjp->proceed();
    if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

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

  //replacing send() by gnutls_record_send() on a secured socket
  advice call("% send(...)") : around () : ImportParameter(gnutls_session
      session){

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

5 Algorithms and Implementation

This section presents the elaborated algorithms for dominator set, graph labeling, *GAFlow*, *GDFlow*, *ExportParameter* and *ImportParameter*. Algorithms that operate on CFG have been developed for decades now, and many graph operations are considered to be common knowledge in computer science. Despite this theoretical richness, we are not aware of existing methods allowing to determine the *GAFlow* or *GDFlow* node for a particular set of nodes (i.e. join points) in a CFG by considering all the possible paths. On the other hand, the algorithms used to calculate the Dominator and Post-Dominator sets of a CFG node 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 one is based on the Dominator and Post-Dominator algorithms of classical CFG, while the other one operates on labeled graph. 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 [26]. 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 PostDominator

The problem of finding the dominators in a control-flow graph has a long history in the literature and many algorithms have been proposed, improved and implemented [27,28]. To compute dominance information, as presented in [27], the compiler can annotate each node in the CFG with a *DOM* and *PDOM* sets.

DOM(b): A node n in the CFG dominates b if n lies on every path from the entry node of the CFG to b. *DOM(b)* contains every node n that dominates b, including b.

The dominators of a node n are given by the maximal solution to the following data-flow equations:

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

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

Where *entry* is the start node. The dominator of the start node is the start node itself. The set of dominators for any other node n is the intersection of the set of dominators for all predecessors p of n. The node n is also in the set of dominators for n. To solve these equations, the iterative algorithm presented in [27] can be used.

We elaborated and implemented Algorithm 1 to calculate the Dominator set. It is based on the mechanisms for identifying the possible paths of reaching one destination from one source node [29]. However, any other available algorithm that gives the same result can be useful. This choice is completely left for the developer who is expert in such domain.

The proposed algorithm for finding the non-trivial dominator nodes of a node n, starting from an entry point node is based on finding all the connecting 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.

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

Algorithm 1 Algorithm to Determine the Dominator Set

```
1: Set DOM(Node entryPointNode, 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 pth in pathList do
9:       meetSet = meetSet ⋂ (Set)pth
10:   end for
11: end if
12: return meetSet;
13:
14: markNode(Node targetNode, Node currentNode, Node branchingNode, int
    markIndex)
15: if ∄ currentNode.pathMarkingMap.get(branchingNode) then
16:   currentNode.pathMarkingMap.put(branchingNode, markIndex);
17:   if currentNode != targetNode then
18:      markIndex = markIndex + 1;
19:      for each parent p in currentNode.parentList do
20:         markNode(targetNode, p, currentNode, markIndex);
21:      end for
22:   end if
23: end if
24:
25: tracePath(Node targetNode, Node currentNode, Stack ascendList, Stack
    pathList)
26: ascendList.push(currentNode);
27: if currentNode == targetNode then
28:   List path = new List();
29:   path.addAll(ascendList);
30:   pathList.add(path);
31: else
32:   for each parent p in currentNode.parentList do
33:      if ∃ p.pathMarkingMap.get(currentNode) then
34:         pathMarkValue=p.pathMarkingMap.get(currentNode);
35:         for each markingKey in currentNode.pathMaringMap.keySet() do
36:            int markingValue = currentNode.pathMarkingMap.get(markingKey);
37:            if markingValue + 1 == pathMarkValue then
38:               currentNode.pathMarkingMap.remove(markingKey);
39:               tracePath(targetNode, p, ascendList, pathList);
40:               currentNode.pathMarkingMap.put(markingKey,
                  markingValue);
41:               break;
42:            end if
43:         end for
44:      end if
45:   end for
46: end if
47: 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 [28]. 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).

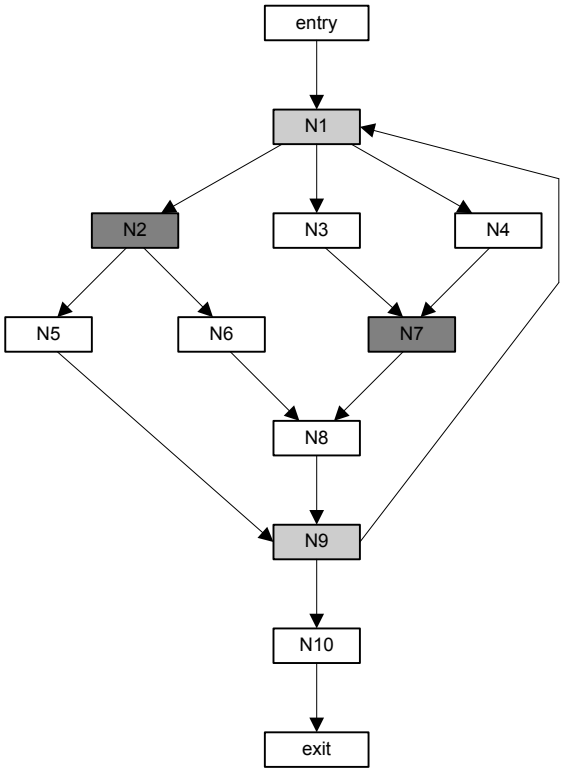


Fig. 3. Graph Illustrating the *GAFlow* and *GDFlow* of N2 and N7

5.1.1 GAFLow

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

We implemented Algorithm 2 using Algorithm 1 to calculate the dominator set of a particular node. Then, we applied this implementation on the case study illustrated in Figure 3. The result of calculating the *GAFlow* of some selected nodes are illustrated in Table 2.

Algorithm 2 Algorithm to determine *GAFlow* using dominator

Require: *SelectedNodes* is initialized with the contents of the pointcut match

1: `GAFlow`(NodeSet SelectedNodes):
2: $CommonDomSet \leftarrow \emptyset$
3: **for all** $node \in SelectedNodes$ **do**
4: $\quad CommonDomSet \leftarrow CommonDomSet \cup (DOM(node) - node)$
5: **end for**
6: **return** $GetLastNode(CommonDomSet)$

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

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

Table 2
Results of the Execution of Algorithm 2 on Figure 3

5.1.2 GDFLow

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

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

Require: *SelectedNodes* is initialized with the contents of the pointcut match

1: `GDFlow`(NodeSet SelectedNodes):
2: $CommonPostDomSet \leftarrow \emptyset$
3: **for all** $node \in SelectedNodes$ **do**
4: $\quad CommonPostDomSet \leftarrow CommonPostDomSet \cup (DOM(node) - node)$
5: **end for**
6: **return** $GetFirstNode(CommonPostDomSet)$

Similarly to Algorithm 2, we implemented Algorithm 3 by reversing the edge

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

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

Table 3
Results of the Execution of Algorithm 3 on Figure 3

direction of the CFG, starting from the exit node and applying Algorithm 1 to calculate the post-dominator set of a particular node [28]. Then, we applied this implementation on the case study illustrated in Figure 3. The result of calculating the *GDFlow* of some selected nodes are illustrated in Table 3.

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. 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 Figure 4. This example will be used throughout the rest of this paper.

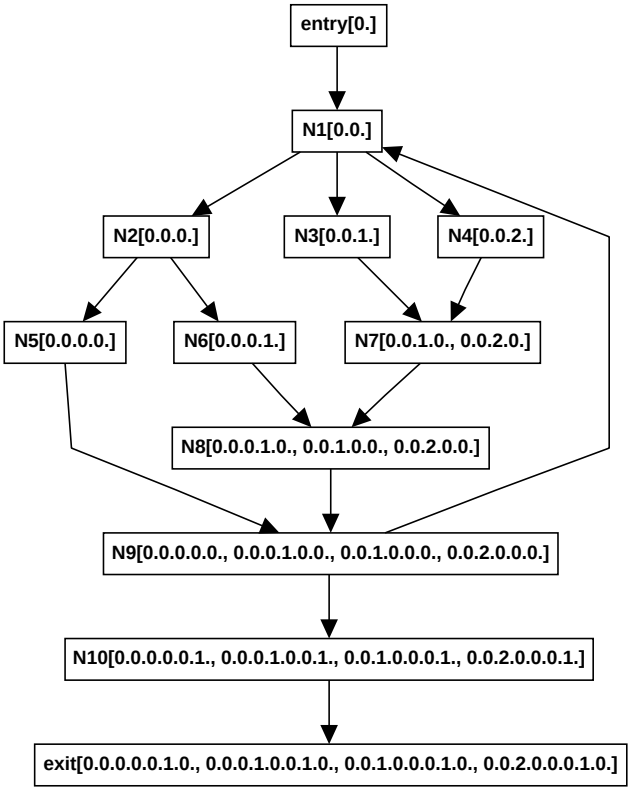


Fig. 4. Sample Labeled Graph

5.2.1 GAFlow

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

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

5.2.2 GDFlow

The same mechanism for reversing the edge direction of the CFG [28], 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, the

Algorithm 4 Hierarchical Graph Labeling Algorithm

```
1: labelNode(Node s, Label l):
2: s.labels ← s.labels ∪ {l}
3: NodeSequence children = s.children()
4: for k = 0 to |children| − 1 do
5:    child ← children[k]
6:    if ¬hasProperPrefix(child, s.labels) then
7:       labelNode(child, l +_c k +_c ".");
8:    end if
9: end for
10:
11: hasProperPrefix(Node s, LabelSet parentLabels):
12: if s.label = ϵ then
13:    return false
14: end if
15: if ∃s ∈ Prefixes(s.label) : s ∈ parentLabels then
16:    return true
17: else
18:    return false
19: end if
20:
21: Prefixes(Label l):
22: LabelSet labels ← ∅
23: Label current ←""
24: for i ← 0 to l.length() do
25:    current.append(l.charAt(i))
26:    if Label1.charAt(i) = '.' then
27:       labels.add(current.clone())
28:    end if
29: end for
```

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

Table 4
Results of the Execution of Algorithm 5 on Figure 4

labeling of the CFG can be performed and then the same algorithm used for calculating the *GAFLow* (Algorithm 5) can be applied to determine the *GDFLow*.

We used the same implementation of Algorithm 4 and case study illustrated in Figure 4. Then, we applied the aforementioned mechanism and implemented

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 ← ∅
3: for all node ∈ SelectedNodes do
4:    Labels ← Labels ∪ node.labels()
5: end for
6: return GetNodeByLabel(FindCommonPrefix(Labels))
7:
8: FindCommonPrefix (LabelSequence Labels):
9: if |Labels| = 0 then
10:    return error
11: else if |Labels| = 1 then
12:    return Labels.removeHead()
13: else
14:    Label Label1 ← Labels.removeHead()
15:    Label Label2 ← Labels.removeHead()
16:    if |Labels| = 2 then
17:       for i ← 0 to min(Label.length(), Label2.length() do
18:          if Label1.charAt(i) ≠ Label2.charAt(i) then
19:             return Label1.substring(0, i − 1)
20:          end if
21:       end for
22:       return Label1.substring(0, min(Label.length(), Label2.length())
23:    else
24:       Label PartialSolution ← FindCommonPrefix(Label1, Label2)
25:       Labels.append(PartialSolution)
26:       return FindCommonPrefix(Labels)
27:    end if
28: end if
```

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

Table 5
Results of the Execution of Reverse Edge Direction and Algorithm 5 on Figure 4

Algorithm 5 to calculate the *GDFlow* for the seclected nodes. Table 5 contains the results along with Figures 6.

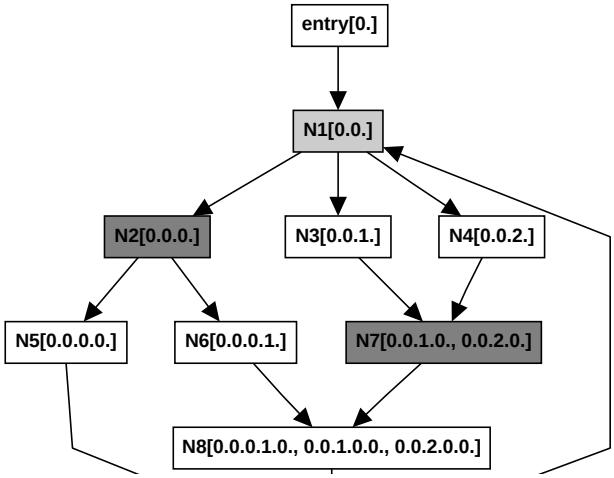


Fig. 5. Excerpt of Labeled Graph Illustrating the *GAFlow* of N2 and N7

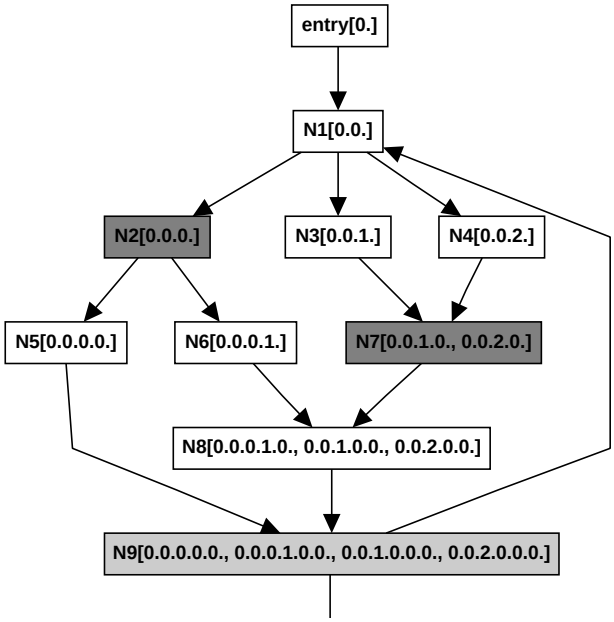


Fig. 6. Excerpt of Labeled Graph Illustrating the *GDFlow* of N2 and N7

5.3 ExportParameter and ImportParameter

This section presents the implementation methodology of the proposed primitives together with some experimental results. The elaborated algorithms of parameter passing operate on a program's call graph. The origin node is the poincut where *ExportParameter* is called, while the destination node is the pointcut where *ImportParameter* is called.

After calculating the closest guaranteed ancestor (*GAFlow*) of the two pointcuts specified by the two primitives, the elaborated Algorithm 7 is performed first in order to pass the parameter from the origin to the closest guaran-

teed ancestor, then executed another time to pass the parameter from the closest guaranteed ancestor to the destination. This procedure is described in Algorithm 6 and operates on one parameter at a time.

Algorithm 6 Algorithm to Pass the Parameter between two pointcuts

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

Algorithm 7 is a building block that allows to modify the function signatures and calls in a way that would keep the program's syntactical correctness and intent (i.e. would still compile and behave the same). It finds all the paths between an origin node and a destination node in a call graph. For each path, it propagates the parameter from the called function to the callee, starting from the end of the path. In order to be optimal, it modifies all the callers only one time and keeps track of the modified nodes.

Listing 9. Excerpt of a Program to be Hardened

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

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

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

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

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

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

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

    /* Shutdown */
    close(sd);

    /* ...   */
}
```

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
4: end if
5: paths ← findPathsBetween(origin, destination)
6: for all path ∈ paths do
7:    path.remove(origin)
8:    while ¬path.isEmpty() do
9:       currnode ← path.tail()
10:      path.remove(currnode)
11:      if ¬node.signature.isModified() ∧
         ¬∃parameter ∈ currnode.signature() : parameter = param then
12:         node.signature.addParameter(param)
13:         node.signature.markModified()
14:         modifyFunctionsCallsTo
            (currNode, param)
15:      end if
16:   end while
17: end for
18: return success
19:
20: function modifyFunctionsCallsTo(Node currnode, Parameter param):
21: for all caller ∈ currnode.getCallers() do
22:    for all call ∈ caller.getCallsTo(node) : ¬call.modified do
23:       call.parameters.add(param)
24:       call.modified = true
25:    end for
26: end for
```

We implemented a program similar to the scenario of the call graph illustrated in Figure 2. This program presented in Listing 9 is essentially a client application that establishes a connection, sends a request and receive a response from the server. Then, we simulated the mechanisms for passing parameters and applied the aspects presented in Listings 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.

Listing 10. Resulting Hardened Program

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

```
int dosend(int sd, char * buffer, unsigned int bufSize, gnutls_session_t *
    session){
  if (session != NULL) return gnutls_record_send(*session, buffer, bufSize);
  else return send(sd, buffer, bufSize, 0);
}

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

int doConnect(int sd, struct sockaddr_in servAddr, gnutls_session_t * session
    , gnutls_certificate_credentials_t * xcred){

  static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
      GNUTLS_CRT_OPENPGP, 0};
  int rc;
  gnutls_init (session, GNUTLS_CLIENT);
  gnutls_set_default_priority (*session);
  gnutls_certificate_type_set_priority (*session, cert_type_priority);
  gnutls_certificate_allocate_credentials (xcred);
  gnutls_credentials_set (*session, GNUTLS_CRD_CERTIFICATE, *xcred);

  rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
  if (rc >= 0){
    gnutls_transport_set_ptr (*session, (gnutls_transport_ptr) sd);
    rc = gnutls_handshake (*session);
  }
  return rc;
}


int main (int argc, char *argv[]) {
    gnutls_global_init ();

    /* ... */

      /* create socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd<0) {
      perror("cannot␣open␣socket");
      exit(1);
    }

    doConnect(sd, servAddr,&session,&xcred);
    dosend(sd,HTTPrequest,strlen(HTTPrequest) + 1,&session);
    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;

}
```

6 Related Work

Many shortcomings of AOP for security concerns have been documented and some improvements have been suggested so far. In the sequel, we present the most noteworthy. A dataflow pointcut that is used to identify join points based on the origin of values is defined and formulated in [9] for security purposes. The authors expressed the usefulness of their pointcut by presenting an example on sanitizing web-applications. For instance, such a pointcut can detect if the data sent over the network depends on information read from a confidential file. This poincut is not fully implemented yet.

In [7], Harbulot and Gurd proposed a model of a loop pointcut that explores the need for a loop join point that predicts infinite loops, which are used by attackers to perform denial of service attacks. Their approach for recognizing loops is based on a control-flow analysis at the bytecode level in order to avoid ambiguities due to alternative forms of source-code that would produce identical loops. This model contains also a context exposure mechanism for writing pointcuts that select only specific loops.

In [6], Bonér discussed a poincut that is needed to detect the beginning of a synchronized block and add some security code that limits the CPU usage or the number of instructions executed. 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 [8] without a precise definition. Such pointcut may allow to select points within the control flow of a join point starting from the root of the execution to the parameter join point. In the same presentation, an operator is introduced in order to obtain the minimum of two pcflow pointcuts, but it is never clearly defined what this minimum can be or how can it be obtained. These proposals could be used for software security, in the enforcement of policies that prohibit the execution of a given function in the context of the execution of another one.

Local variables set and get poincuts were introduced in [30] for increasing the efficiency of AOP in security. They allow to track the values of local variables inside a method. It seems that these poincuts can be used to protect the privacy and integrity of sensitive data. Their idea is based on the approach presented in [10], which describe an extension to Java called JFlow. This language allows to statically checks information flow annotations within programs and provides several new features such as decentralized label model, label polymorphism, run-time label checking and automatic label inference.

It also support objects, subclassing, dynamic type tests, access control, and exceptions.

Regarding the appropriateness of AOP for security, few contributions have been published on applying it to secure software. Most of them are presented as case studies that show the relevance of AOP for security or explore the usability of a proposed AOP language. The following is a brief overview on the available contributions. Cigital labs proposed an AOP language called CSAW [4], which is a small superset of C programming language dedicated to improve the security of C programs. De Win, in his Ph.D. thesis [2], discussed an aspect-oriented approach that allowed the integration of security aspects within applications. It is based on AOSD concepts to specify the behavior code to be merged in the application and the location where this code should be injected. In [1], Ron Bodkin surveyed the security requirements for enterprise applications and described examples of security crosscutting concerns, with a focus on authentication and authorization. Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang et al. [3] introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. These research initiatives focus mainly on exploring the usefulness of AOP for securing software.

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 [31] 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. [32]. The construction of call graphs has been documented by Grove et al. [33] in the case of object-oriented languages, and an elaborated study of different algorithms was provided by Grove and Chambers in [25].

In [27], the authors proposed a simple and fast algorithm to calculate the dominance information (e.g. dominator set, post-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 algorithm (Class DominaceInfo) has been provided in [28] as part of the Machine-SUIF control flow analysis (CFA) library. It is built on top of the control flow graph (CFG) [34] library and provides dominator analysis and natural-loop analysis. Other approaches that use lattice theory allow to efficiently compute a Lower Upper Bound (LUB) and Greater Lower Bound (GLB) over lattices [35]. 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 *GAFlow* and *GDFlow*. Moreover, the lattices do not support the full range of expression provided by the CFG, as the latter can be a directed cyclic graph.

7 Conclusion

AOP appears to be 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 first showed the appropriateness and limitations of the current AOP languages for securing software. Then, we illustrated with examples the usefulness of our proposed pointcuts for performing security hardening activities and presented a motivating example that explores the need for parameter passing. Afterwards, we defined the new pointcuts and primitives and we presented the corresponding elaborated algorithms. Finally, we presented the experimental results of our case studies' implementation.

Concerning our future work, we are currently working on implementing and deploying the proposed pointcuts primitives into AspectJ and AspectC++ weavers. Moreover, we are going to address other shortcomings of AOP for security and provide solutions for them.

References

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

[2] B. DeWin, Engineering application level security through aspect oriented software development, Ph.D. thesis, Katholieke Universiteit Leuven (2004).

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

[4] V. Shah, An aspect-oriented security assurance solution, Tech. Rep. AFRL-IF-RS-TR-2003-254, Cigital Labs (2003).

[5] P. Slowikowski, K. Zielinski, Comparison study of aspect-oriented and container

managed security, in: Proceedings of the ECCOP workshop on Analysis of Aspect-Oriented Software, 2003.

[6] J. Bonér, Semantics for a synchronized block join point, http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-join-point/ (accessed 2007/09/26) (2005).

[7] B. Harbulot, J. R. Gurd, A join point for loops in AspectJ, in: Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March, 2005.

[8] G. Kiczales, The fun has just begun, keynote talk at AOSD 2003, http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt (accessed 2007/04/19) (2003).

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

[10] A. C. Myers, Jflow: Practical mostly-static information flow control, in: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99), ACM Press, New York, NY, USA, 1999, pp. 228–241.

[11] M.-A. Laverdière, A. Mourad, A. Soeanu, M. Debbabi, Control flow based pointcuts for security hardening concerns, in: Proceedings of the 2007 International Conference on Privacy, Security and Trust (IFIP-PST 2007), Springer, 2007.

[12] D. Xu, V. Goel, K. Nygard, An aspect-oriented approach to security requirements analysis, in: 30th Annual International Computer Software and Applications Conference (COMPSAC'06), IEEE Computer Society, Los Alamitos, CA, USA, 2006, pp. 79–82.

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

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

[15] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn, Using aspectc to improve the modularity of path-specific customization in operating system code, in: Proceedings of Foundations of software Engineering, Vienne, Austria, 2001.

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

[17] H. Kim, An aosd implementation for c#, Tech. Rep. TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin (2002).

[18] K. Bollert, On weaving aspects, in: International Workshop on Aspect-Oriented Programming at ECOOP99, 1999.

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

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

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

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

[23] D. A. Wheeler, Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010, 2003, `http://www.dwheeler.com/secure-programs/` (accessed 2007/04/19).

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

[25] D. Grove, C. Chambers, A framework for call graph construction algorithms, ACM Trans. Program. Lang. Syst. 23 (6) (2001) 685–746.

[26] E. Gomez, Cs624- notes on control flow graph, `http://www.csci.csusb.edu/egomez/cs624/cfg.pdf` (2003).

[27] K. Cooper, T. Harvey, K. Kennedy, A simple, fast dominance algorithm, Software Practice and Experience 4 (1-10).

[28] G. Holloway, M. D. Smith, The machine-suif control flow analysis library. Harvard University, `http://www.eecs.harvard.edu/machsuif/software/nci/cfa.html` (accessed 2007/09/24) (1998).

[29] E. Dijkstra, Dijkstra's algorithm, `http://en.wikipedia.org/wiki/Dijkstra_algorithm` (accessed 2007/10/12).

[30] D. Hadidi, N. Belblidia, M. Debbabi, Security crosscutting concerns and AspectJ, in: Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006), McGraw-Hill/ACM Press, 2006.

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

[32] D. Callahan, A. Carle, M. W. Hall, K. Kennedy, Constructing the procedure call multigraph, IEEE Trans. Softw. Eng. 16 (4) (1990) 483–487.

[33] D. Grove, G. DeFouw, J. Dean, C. Chambers, Call graph construction in object-oriented languages, SIGPLAN Not. 32 (10) (1997) 108–124.

[34] G. Holloway, M. D. Smith, The machine-suif control flow graph library. Harvard University, `http://www.eecs.harvard.edu/hube/software/nci/cfg.html` (accessed 2007/09/24) (1998).

[35] H. Aït-Kaci, R. Boyer, P. Lincoln, R. Nasr, Efficient implementation of lattice operations, ACM Trans. Program. Lang. Syst. 11 (1) (1989) 115–146.