Control Flow Based Pointcuts for Security Hardening Concerns

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

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{ma_laver,mourad,a_soeanu,debbabi}@ciise.concordia.ca

Abstract. In this paper, we present two new control flow based pointcuts to Aspect-Oriented Programming (AOP) languages that are needed for systematic hardening of security concerns. They allow to identify particular join points in a program's control flow graph (CFG). The first proposed primitive is the *GAFlow*, the closest guaranteed ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second proposed primitive is the *GDFlow*, the closest guaranteed descendant, which returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. We find these pointcuts to be necessary because they are needed to perform many security hardening practices and, to the best of our knowledge, none of the existing pointcuts can provide their functionalities. Moreover, we show the viability and correctness of our proposed pointcuts by elaborating and implementing their algorithms and presenting the results of an explanatory case study.

1 Motivations & Background

In today's computing world, security takes an increasingly predominant role. 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 a few years already, but their practical adoption is limited so far. Software maintainers must face the challenge to improve program security and are often underequipped to do so. In some cases, little can be done to improve the situation,

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

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.

Very few concepts and approaches 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, 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 [3, 5, 9, 16, 17].

However, AOP was not initially designed to address security issues, which resulted in many shortcomings in the current technologies [11, 7]. 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 various programming gymnastics, resulting in additional modules that must be integrated within the application, at an elevated runtime, memory and development cost. Moreover, the result of applying this coding strategy is of a higher level of complexity from the standpoint of auditing and evaluation.

The specification of new security-related pointcuts is becoming a very challenging and interesting domain of research [14, 4, 10]. Pointcuts are used in order to specify where code should be injected, and can informally be defined as a subset of the execution points in a given program execution flow. In this context, we propose in this paper AOP pointcuts that are needed for security hardening concerns and allow one to identify join points in a program's control flow graph (CFG). The proposed primitives are *GAFlow*, and *GDFlow*. *GAFlow* returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. *GDFlow* returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. These poincuts are needed to develop many security hardening solutions. Moreover, we combined all the deployed and proposed pointcuts in the literature, and, as far as we know, 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*.

The remainder of the paper is organized as follows: we first glimpse at security hardening and the related problem that we address in Section 2. In Sections 3 and 4 we show the usefulness of our proposal and its advantages. Afterwards, in Section 5, we describe and specify the *GAFlow* and *GDFlow* pointcuts. In Section 6, we present the algorithms necessary for implementing the proposed pointcuts, together with the required hierarchical graph labeling method. This

section also shows the results of our implementation in a case study. We move on to the related work in Section 7, and then we present some summarizing conclusions in Section 8.

2 Security Hardening

In our prior work [12], we proposed that software security hardening be defined 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.* This definition focuses on solving vulnerabilities, not on their detection. In this context, the following key concepts constitute a classification of security hardening methods:

Code-Level Hardening Changes in the source code in a way that prevents vulnerabilities without altering the design. For example, we can add bound-checking on array operations, and use bounded string operations.

Software Process Hardening Addition of security features in the software build process without changes in the original source code. For instance, the use of compiler-generated canary words and compiler options against double-freeing of memory would be considered as Software Process Hardening.

Design-Level Hardening Re-engineering of the application in order to integrate security features that were absent or insufficient. Design-level changes would be, for example, adding an access control feature, changing communication protocol, or replacing temporary files with interprocess communication mechanisms.

Operating Environment Hardening Improvements to the security of the execution context (network, operating systems, libraries, utilities, etc.) that is relied upon by the software. Examples would be deploying `libsafe`, using hardened memory managers and enabling security features of middleware.

Security hardening practices are usually applied manually by injecting security code into the software [2, 8, 15, 18]. This task entails that the security architects have a deep knowledge of the inner working of the software code, which is not available all the time. In this context, we elaborated in [13] an approach based on aspect orientation to perform security hardening in a systematic and automatic way. The primary objective of this approach is to allow the security architects to perform security hardening of software by applying proven solutions so far and without the need to have expertise in the low-level security solution domain. At the same time, security hardening is applied in an organized and systematic way in order to preclude the alteration of the original software functionalities. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build our solutions. Our experimental results show the usefulness of AOP in reaching the objective of having systematic security hardening. During our

work, we have developed security hardening solutions to secure connections in a client-server application, added access control features to a program, encrypted memory contents for protection and corrected some low-level security issues in C programs. On the other hand, we have also distinguished shortcomings in the available AOP technologies in security and the need to elaborate new pointcuts for security hardening concerns.

3 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 [2, 8, 15, 18]. Examples of such cases would be the injection of security library initialization/deinitialization, privilege level changes, atomicity guarantee, logging, etc. The current AOP models only allow us 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 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.

3.1 Security Library Initialization/Deinitialization

In the case of security library initialization (e.g. access control, authorization, cryptography, etc.), our primitives 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 primitives 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 create an accumulation of code injection statements that would create a significant, and possibly useless, waste of system resources. In listing 2, we see an improved aspect that would 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();
}
```

3.2 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 primitives 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 [8]) 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
}
```

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

3.4 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 or after a set of interesting transactions.

4 General Advantages of *GAFlow* and *GDFlow*

It is clear that our proposed primitives support the principle of separation of concerns by allowing to implement program modification on sets of join points based on a specific concern (as previously exemplified). We now present some general advantages of our 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 our 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.

5 Pointcut Definitions

We provide here the syntax that defines a pointcut p after adding our proposed pointcuts:

$$p \texttt{ ::= call(}s\texttt{) | execution(}s\texttt{) | gaflow(}p\texttt{) | gdflow(}p\texttt{) | }p\texttt{||}p\texttt{ | }p\texttt{\&\&}p$$

where s is a function signature. The *GAFlow* and the *GDFlow* are the new control flow based pointcut primitives. Their parameter is also a pointcut p.

The *GAFlow* primitive 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.

The *GDFlow* primitive 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 a common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes. In the worst case, the first common descendant will be the end point in the program.

6 Algorithms and Implementation

In this section, we present the elaborated algorithms for graph labeling, *GAFlow* and *GDFlow*. We assume that our 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. In the case of multiple ending points, we consider them connected to single end point. Most of these assumptions have been used so far [6]. With these assumptions 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.

6.1 Graph Labeling

Algorithms that operate on graphs have been developed for decades now, and many graph operations (such as finding ancestors, finding descendants, finding paths and so on) are considered to be common knowledge in computer science. Despite this theoretical richness, we are not aware of existing methods allowing to efficiently determine the *GAFlow* and *GDFlow* for a particular set of join points in a CFG by considering all the possible paths. Some approaches use lattice theory to efficiently compute a Least Upper Bound (LUB) and Greatest Lower Bound (GLB) over lattices [1]. 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 expressiveness provided by the CFG, as the latter can be a directed cyclic graph. In order to determine the *GAFlow* and *GDFlow*, we chose to use a graph labeling algorithm developed by our colleagues that we slightly modified in order to meet our requirements. Algorithm 1 describes our 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 1, where the operator $+_c$ denotes string concatenation (with implicit operand type conversion). To that effect, the labeling is done by executing algorithm 1 on the *start* node with label "0.", thus recursively labeling all nodes.

We implemented Algorithm 1 and tested it on a hypothetical CFG. The result is displayed in Figure 1. This example will be used throughout the rest of this paper.

6.2 *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 insured 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 2. Please note that the `FindCommonPrefix` function was specified recursively for the sake of simplicity and understanding.

Moreover, we implemented Algorithm 2 and we applied it on the labeled graph in Figure 1. We selected, as case study, some nodes in the graph for various combinations. Our results, are summarized in Table 1 and Figure 2.

6.3 *GDFlow*

The closest guaranteed descendant is determined by elaborating a mechanism that operates on a labeled CFG of a program. By using Algorithm 3, we obtain the sorted list of all the common descendants of the selected nodes in the input

Algorithm 1 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, N8, N13	N1
N6, N11	N2
N14, N13	N1
N14, N15	N14

Table 1. Results of the Execution of Algorithm 2 on Figure 1

list of the pointcut. The principle of this algorithm is to calculate the set of descendants of each of the input nodes and then perform the intersection operation on them. The resulting set contains the common descendants of all the input nodes. Then, we sorted them based on their path length.

Algorithm 4 determines the closest guaranteed descendant. It takes first the result of Algorithm 3, which is considered as its list of possible solutions. Then, it iterates through the ordered list and examines if the current node against the previously retained solution, which we call the candidate. For each selected

Algorithm 2 Algorithm to determine *GAFlow*

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
```

Algorithm 3 Algorithm to Determine the Common Descendants

Require: *SelectedNodes* is initialized with the contents of the pointcut match
Require: *Graph* has all its nodes labeled

```
1: findCommonDescendants(NodeSet SelectedNodes):
2: NodeSet PossibleSolutions ← Graph.allNodes()
3: for all node ∈ SelectedNodes do
4:    PossibleSolutions ← PossibleSolutions ∩ node.AllDescendants()
5: end for
6: Create OrderedSolutions by sorting PossibleSolutions by increasing path length
   between the solution and the nodes in SelectedNodes
7: return OrderedSolutions
```

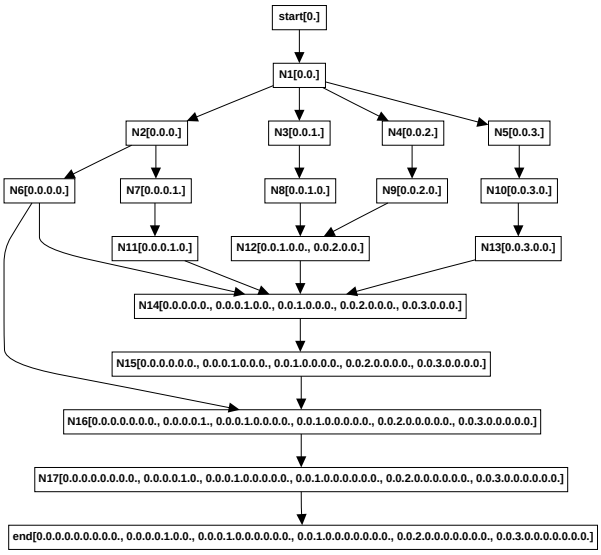

Fig. 1. Labeled Graph

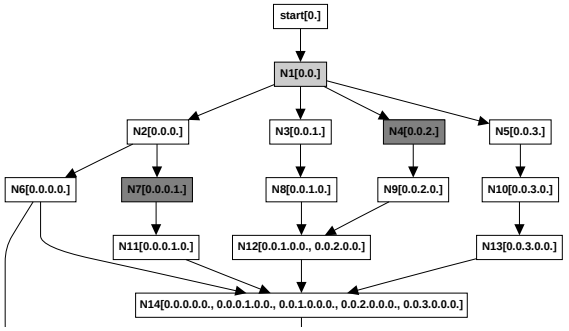

Fig. 2. Excerpt of Graph Illustrating the *GAFlow* of N4 and N7

node, we count the number of labels of the candidate that have proper prefixes identical to the labels of the considered selected node. The resulting candidate of the first iteration is the first encountered node with the largest label count. This candidate is the starting one of the next iteration and so on until all the selected nodes are examined. The final candidate of the last iteration is returned by the algorithm as the closest guaranteed descendant.

We used the same implementation of Algorithm 1 and case study illustrated in Figure 1. With this, we first implemented Algorithm 3 to determine the list

Algorithm 4 Algorithm to Determine the *GDFlow*

Require: *SelectedNodes* is initialized with the contents of the pointcut match
Require: *Graph* has all its nodes labeled

```
1: gdflow(NodeSet SelectedNodes):
2: NodeSequencePossibleSolutions ← findCommonDescendants(SelectedNodes)
3: IntCandidateIndex ← 0
4: for all node ∈ SelectedNodes do
5:    CandidateIndex ← findBestCandidate(PossibleSolutions, CandidateIndex, node)
6: end for
7: return PossibleSolutions[Candidate]
8:
9: findBestCandidate(NodeSequence possibleSolutions, int CandidateIndex, Node
   selectedNode)
10: PreviousFoundPrefixes ← 0
11: for i ← CandidateIndex to |possibleSolutions| − 1 do
12:    Int sol ← possibleSolutions[i]
13:    Int foundPrefixes ← countProperPrefixes(sol, node)
14:    if (PreviousFoundPrefixes < foundPrefixes) ∨ ∃child ∈ sol.children() :
       hasProperPrefix(sol, child.labels()) then
15:       CandidateIndex ← i
16:    end if
17: end for
18: return CandidateIndex
19:
20: countProperPrefixes(Node candidate, Node selectedNode):
21: Int count ← 0
22: for all candidateLabel ∈ candidate.labels() do
23:    for all selectedNodeLabel ∈ selectedNode.labels() do
24:       if ∃p ∈ Prefixes(candidateLabel) : p = selectedNodeLabel then
25:          count + +
26:       end if
27:    end for
28: end for
29: return count
```

of common descendants for different selected nodes, as summarized in Table 2. Then, we implemented Algorithm 4 to calculate the *GDFlow* for the list of common descendants previously computed by applying the aforementioned conditions. Table 2 contains the results for this algorithm along with Figures 3 and 4.

7 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.

Selected Nodes	Common Descendants	*GDFlow*
N2, N8, N13	N14, N15, N16, N17, end	N16
N6, N11	N14, N15, N16, N17, end	N16
N14, N13	N15, N16, N17, end	N15
N14, N15	N16, N17, end	N16

Table 2. Results of the Execution of Algorithm 3 and 4 on Figure 1

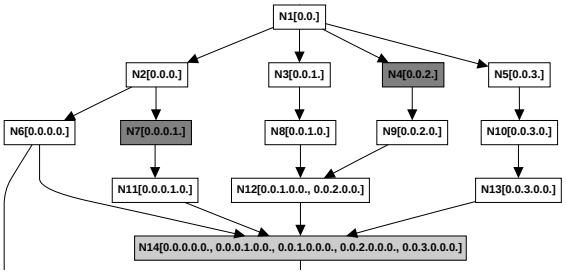

Fig. 3. Illustration of the *GDFlow* of N4 and N7 as N14

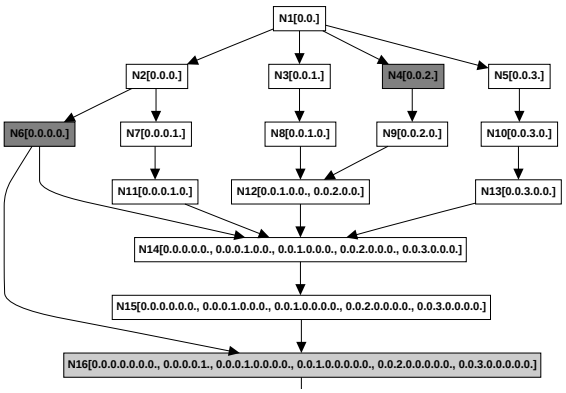

Fig. 4. Illustration of the *GDFlow* of N4 and N6 as N16

A dataflow pointcut that is used to identify join points based on the origin of values is defined and formulated in [11] for security purposes. This poincut is not fully implemented yet. For instance, such a pointcut detects if the data sent over the network depends on information read from a confidential file.

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 of attacks.

Another approach, that discusses local variables set and get poincut, has been proposed by Myers [14]. The author claims that this pointcut is necessary for increasing the efficiency of AOP in security since it allows one to track the values of local variables inside a method. It seems that this poincut can be used to protect the confidentiality of local variables.

In [4], 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 [10] 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.

8 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. In this context, we proposed two new pointcuts to AOP for security hardening concerns: The *GAFlow* and *GDFlow*. 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. We first showed the limitations of the current AOP languages for many security issues. Then, we illustrated the usefulness of our proposed pointcuts for performing security hardening activities. Afterwards, we defined the new pointcuts and we presented the corresponding elaborated algorithms. Finally, we presented our implementation of pointcuts and a case study that explore their correctness.

References

1. Hassan Ait-Kaci, Robert S. Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *Programming Languages and Systems*, 11(1):115–146, 1989.
2. Matt 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).
3. Ron Bodkin. Enterprise security aspects, 2004. `http://citeseer.ist.psu.edu/702193.html` (accessed 2007/04/19).

4. J. Bonér. Semantics for a synchronized block join point, 2005. http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-joint-point/ (accessed 2007/04/19).
5. B. DeWin. *Engineering Application Level Security through Aspect Oriented Software Development*. PhD thesis, Katholieke Universiteit Leuven, 2004.
6. Ernesto Gomez. Cs624- notes on control flow graph. http://www.csci.csusb.edu/egomez/cs624/cfg.pdf.
7. B. harbulot and 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. Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.
9. M. Huang, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *AOSD:AOSDSEC 04: AOSD Technology for Application level Security, March*, 2004.
10. G. Kiczales. The fun has just begun, keynote talk at AOSD 2003, 2003. http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt (accessed 2007/04/19).
11. H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 105–121, 2003.
12. A. Mourad, M-A. Laverdière, and M. Debbabi. Security hardening of open source software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. ACM, 2006.
13. A. Mourad, M-A. Laverdière, and M. Debbabi. Towards an aspect oriented approach for the security hardening of code. In *To appear in the Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems*. IEEE Press, 2007.
14. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
15. R. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.
16. Viren Shah. An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, 2003.
17. Pawel Slowikowski and Krzysztof Zielinski. Comparison study of aspect-oriented and container managed security. In *Proceedings of the ECCOP workshop on Analysis of Aspect-Oriented Software*, 2003.
18. D. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. http://www.dwheeler.com/secure-programs/ (accessed 2007/04/19).