Research Journal of Applied Sciences, Engineering and Technology 9(6): 396-408, 2015
DOI:10.19026/rjaset.9.1419
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: September 24, 2014 Accepted: October 24, 2014 Published: February 25, 2015

Research Article

Trade-off Analysis of Crosscutting Functionalities using Lazy Counting-based Splay Tree in Aspect Oriented Programming

[1]K. Santhi, [1]G. Zayaraz and [2]T. Chellatamilan
[1]Deparnment of CSE, Pondicherry Engineering College, Puducherry, India
[2]Deparnment of CSE, Arunai Engineering College, Tiruvannamalai, India

Abstract: Aspect Oriented Programming (AOP) provides new modularization of software systems through encapsulation of crosscutting functionalities, providing a clear isolation and utilization thereof. The trade-offs are typically a consequence of technical contradictions in requirements. We employ a data structure called a lazy counting based splay tree to analyze the trade-off between the conflicting quality attributes. These contradictions must be conquered in order to achieve breakthrough. The performance of this data structure is verified after considering Cross Site Request Forgery (CSRF) which could be prevented by same-origin policy. The results are promising and show good potential for lazy counting-based splaying, which is capable of analyzing the overall performance of a splay tree compared with a lazy counting-based splay tree and providing interesting results about both.

Keywords: Aspect-oriented programming, crosscutting functionalities, cross-site request forgery, lazy counting-based splay tree, same-origin policy, trade-off analysis

INTRODUCTION

Developing an application software system always demands consideration of both functional and non-functional requirements. Modularizing the expansion of different requirements has important advantages in system evolution. Since such requirements typically originate from different stakeholders, they may cause different iterations of various parts of the software development process. Successful distribution of concerns can guide effortless development, maintenance and possible reuse, amongst others (Boström, 2004). State-of-the-art software techniques already support separation of concerns, for example, by means of method structuring, Object-Oriented Programming (OOP) and design patterns. On the other hand, these techniques are inadequate for separating the crosscutting functionality in broad-based functionality. A major cause of this limitation is the separation of concerns in an intuitive manner by grouping them into objects, though this technique is only efficient in separating concepts that map easily to objects and not for separating concerns.

Aspect-Oriented Programming (AOP) provides techniques for managing crosscutting concerns into a single manageable component, which is referred to as an aspect. The concept of an aspect is at the heart of AOP and is used to solve many problems, such as representations of tangling and scattering. Tangling refers to how concerns intermingle with each other in a module, while scattering refers to how concerns are separated over many modules.

However, the detection and order of crosscutting concerns and their consideration as an aspect, are challenging tasks. Developing secure software systems requires more than protecting objects from illegal manipulation; it also requires the prevention of illegal information flow among objects in a system (Izaki *et al*., 2001).

Another benefit of AOP is that because the core functionality of the system is executed separately, the developer no longer needs to refer to or use security mechanisms in the system. Implementing security could be left to a security expert and security policies can be independently implemented (Win *et al*., 2002).

Incorporating information flow control during software development is tremendously challenging. First, the main issue with building real applications with information flow options is interfacing the new application with existing infrastructure that has not been designed with information flow in mind. Second, it is challenging to manage and assign security policies throughout the software development process. Third, the programmer is required to understand not only the algorithm, but also what the desired security policy is and how it can be formalized (Zdancewic, 2004).

Several authors have discussed the benefits of using AOP to implement security concerns (Viega *et al*., 2001; De Win *et al*., 2001). Aspect-oriented software development is relevant to all the key aspects of

Corresponding Author: K. Santhi, Department of CSE, Pondicherry Engineering College, Puducherry, India

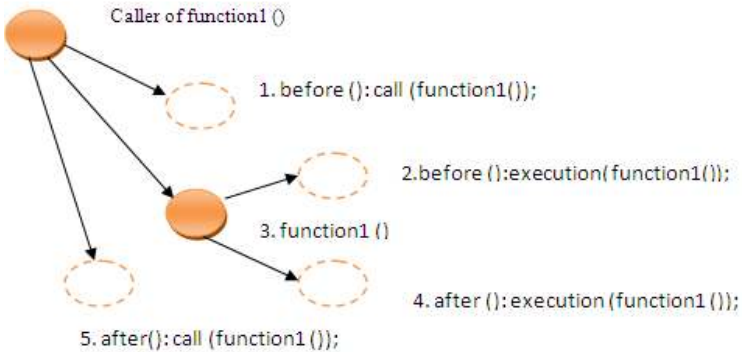


Fig. 1: Pointcut and its advices

security, namely, verification, validation, access control, integrity checks, non-repudiation and synchronization, as well as for supporting the administration and exception handling required for effective security. Aspect-oriented software design is flexible enough to accommodate the implementation of additional security features after the functional system has been developed.

Overview of aspect-oriented programming: AOP is the perfect complement to OOP in software engineering by providing more advanced modularization techniques to handle the scattering and tangling problems than existing models.

In OOP the objects have properties and perform intended actions. However, the process of applying abstracted functionality is performed by the developer, which is more error prone and hence, less secure. On the contrary, in AOP this process is performed methodically, consistently and more precisely by the aspect weaver. In AOP the aspects have properties that can affect the entire performance or that of some of the components, such as the way a method is executed, synchronization, concurrency, resource allocation, exception handling, logging and so on.

The main feature of this technology is its ability to specify both the behavior of one specific functionality as well as binding this functionality to other functionalities or non-functionalities such as its relation to these. An aspect is a modular unit of a crosscutting implementation, which is provided in terms of pointcuts and advices, specifying what type of advice and when a pointcut aspect is going to be executed, as shown in Fig. 1. In the execution of a program, there will be join points where calls to an aspect can be injected. A pointcut is used to find a set of join points where an aspect can be injected. An advice declaration can be used to specify code that should run when the join points specified by the pointcut expression are reached. The advice code will be executed when a particular join point is reached, either before or after execution proceeds. A before/after advice on a method execution defines code that must be run before/after the particular method is executed, while an around advice defines code that is executed when the join point is reached and has control over whether the computation at the join point is allowed to be performed (Kiczales *et al*., 2001).

The final application is created using both the functional code and its specific pointcut aspects. These two entities are combined to generate the byte code at compile time by invoking a special method called a weaver.

LITERATURE REVIEW

A software system that manipulates and stores credentials like passwords, identification documents, security clearances and tax information must prevent such information from being leaked during execution (Sabelfeld and Myers, 2003). Security mechanisms, such as firewalls or anti-virus software and also access control mechanisms are not adequate to protect against this type of information leakage. For example, determining whether communication breaches confidentiality is beyond the scope of any firewall mechanism. Similarly with encryption, there is no guarantee that once data is decrypted that its confidentiality will be maintained (Sabelfeld and Myers, 2003). An access control policy determines the right to access objects containing information (Huang *et al*., 2004). However, this type of control only relates to the release of data and does not control how data 'flows' during the execution of each statement in a program. Numerous instances of information "leakage" arise not from defective access control, but from the lack of policies about information flow (Wand *et al*., 2004).

According to Mourad *et al*. (2008), AOP permits security hardening of applications by allowing the incorporation of supplementary security requirements to previously existing code that was designed to operate in a different security context. AOP also permits the integration of security into applications even when the source code is no longer available. For example, it may be required to apply tighter security requirements to legacy applications for which the source code may have been lost. In addition, with AOP, security can be

selectively applied to important areas of the application either explicitly or declaratively without having to change the code. The capability to declaratively applying security to specific areas of the code also provides the ability to easily apply multiple security features to a specific scheme.

The Bell-La Padula (Bell, 2005) model is a mathematical model that utilizes the principles of mathematical theory to describe access methods in computer systems. This model uses four access modes, namely, read, append, execute and write. A set of rules is defined and proved to possess certain characteristics such as preserving simple security, discretional security and the *-property. The Bell-La Padula model has mostly been applied in military based systems where confidentiality of data is of the utmost priority.

The Biba Stewart *et al*. (2005) model was developed after the Bell-La Padula model and followed similar principles of mathematical state machines to address the issue of data integrity. This model has mainly been used in commercial applications where the integrity of data is of greater concern than its confidentiality.

De Win *et al*. (2001) defined three types of aspects, namely, identification, authentication and authorization, for access control policies in the aspect-oriented paradigm. The identification aspect is used to tag those entities that must be authenticated and is used as a container for identity information of the subject.

The authentication aspect passes authentication information to the access control mechanism. The authorization aspect checks access based on the identity information received from the authentication aspect.

Ramachandran *et al*. (2006) also addressed authentication and authorization within the aspect-oriented paradigm by providing a more general approach. However, they do not address information flows.

Kawauchi and Masuhara (2004) used an aspect to identify cross-site scripting. Their approach is predicated on validating the parameters by replacing special characters by quoted ones within input files submitted by users to web applications. They found that although sanitizing is a crosscutting concern, there is no possible way to define a pointcut that would be able to detect whether a string originated from an unauthorized source or contained unwanted information. Hence, they proposed a new pointcut called dflow that addresses the dataflow between join points as an extension to the AspectJ language. Kawauchi and Masuhara (2004) do not address security classifications or their dataflow definitions and only deal with direct information flow. Furthermore, they do not comment on transmission of information between objects in a method. Since no advancement has been made exclusively in this area and since AOP and OOP are complementary, it is important to investigate information flow control and security policies from this viewpoint first.

Hermosillo *et al*. (2007) proposed a solution for web security against SQL injection and XSS attacks using AOP. Their work is a test of the reward for chaining security policies at runtime, with testing carried out against the first two insecurities, SQL injection and XSS. The solution presented provides a security aspect for a web application server. The authors have used aspects to validate the injected SQL in the web application server and also to validate XSS attacks in the user's requests to the web application and from the web server to a database server. This allows the interception of all database accesses and the validation thereof before potentially dangerous information is stored (Lee *et al*., 2012). A significant amount of work has been carried out in aspect-oriented security to make the process more systematic in terms of software design and development (De Win *et al*., 2002; De Win *et al*., 2001; Hermosillo *et al*., 2007).

Simic and Walden (2013) designed a system to mitigate cross-site scripting and SQL injection vulnerabilities, the most common web application vulnerabilities, with no demand in expensive and potentially hazardous modifications to the source code of Web applications. At runtime, the application executes the protective aspect code to mitigate security issues when a block of vulnerable code is executed.

The subsequent discussion highlights the relevance of aspect-oriented technology in terms of implementing some of the key components of security such as access control, authentication, persistence, transaction and monitoring, exception handling and synchronization in software systems.

METHODOLOGY

An aspect-based approach for representing information flow control: Plugging in non-functional aspects to functional aspects may result in several attacks, which may lead to a change in the composition rules, modifying the precedence of advices, unexpected behavior combinations, unhandled inputs and so on.

These security vulnerabilities can be taken advantage of by using Cross-Site Request Forgery (CSRF or XSRF). A user may not be aware that such an attack has occurred and may only find out about the attack after the damage has been done since no remedy is applied. To avoid such attacks, the approach of using same-origin policies could be applied.

To secure the functionality completely, one needs to perform internal dataflow analysis and ensure that un-trusted input is sanitized before being used and that sensitive data is not released without authorization. Identifying illegal flows between objects requires an aspect based on the principle of a same-origin policy. Here pointcuts can be utilized to identify flows between objects. This aspect observes objects and intercepts

Table 1: Injection of risky aspects

```
Aspect CSRFRestriction {
pointcut aspect injection (Aspect aspect)
execution (*Aspect.* (..)) && this (aspect)
before (Aspect aspect): aspectInjection (aspect) {
//reference to the same origin this aspect is deployed in
Sop sop = …;
//this aspect does not belong to same origin policy
If (! sop. belongs (aspect)) {
throw new AccessControlException ();
}
}}
```

messages flowing between them. The aspect's advice would then determine upon examining the given message and classification of the sender and recipient, whether to allow the information flow. A same-origin policy helps to detect these and many other vulnerabilities by identifying data sources and sinks and how secure they are. Generally, all sources outside the component should be considered untrustworthy (e.g., system calls, third party plug-ins and routines that can copy data from the user space or network).

The same-origin policy approach is utilized to apply the policy model at runtime, thereby allowing aspects originating from the same location, such as a combination of scheme, hostname and port number. The weaver supplements woven software with logic to maintain the permission state of the software. As such, only the weaver is altered and no modification to the virtual machine or language semantics is needed.

Table 1 illustrates how the mechanism of injection can be used to disable a CSRF malicious aspect by intrinsically calling security features.

This policy protects against eventualities in which un-trusted aspects, which may originate from third party libraries (especially from CSRF), are incorrectly woven into the rest of the application code, as shown in Fig. 2. This policy has the means to detect and prevent a number of remote-user exploits, such as cross-site scripting, HTML injection, SQL injection and command injection (Simic and Walden, 2013). Severe security flaws would exist if there were no restrictions isolating the aspect from different origins and all the pointcuts of aspects between them were allowed. While it may be considered safe to perform the pointcut of aspects between different origins through hyperlinks, automated injection of aspects could potentially be dangerous.

Lazy counting-based splay tree: By injecting crosscutting functionalities into core functionalities, cross site request forgery could results in many attacks. CSRF attack can be used to modify the precedence of advices and unexpected combinations, fire unauthorized aspects, prevent the aspects from advising a joining point, or conduct fraudulent financial transactions. To organize injected aspects from the same origin, a counting-based splay tree is used. While the authorized aspect could be compromised after the same origin policy is verified before injection into the core functionality, when a new pointcut aspect enters the system the following actions are needed:

- The CSRF is checked using the same origin policy
- Legitimacy of the code is verified

If the pointcut is already available in the counting-based splay tree, the counters are updated accordingly

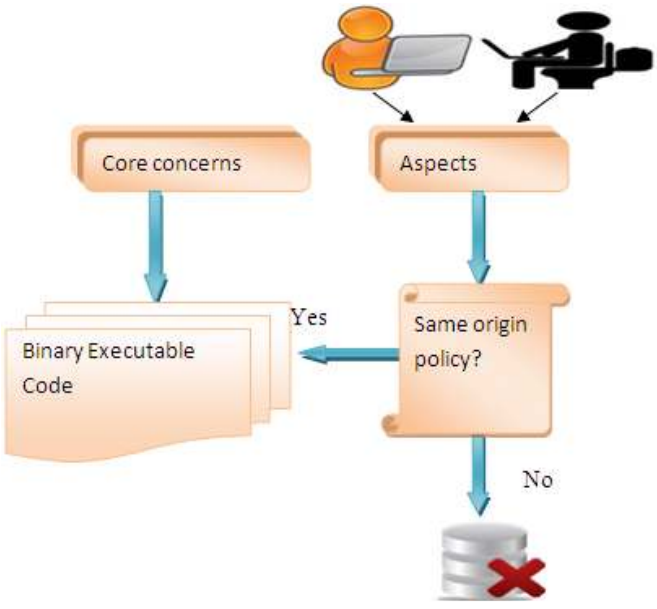


Fig. 2: Approach using a same-origin policy

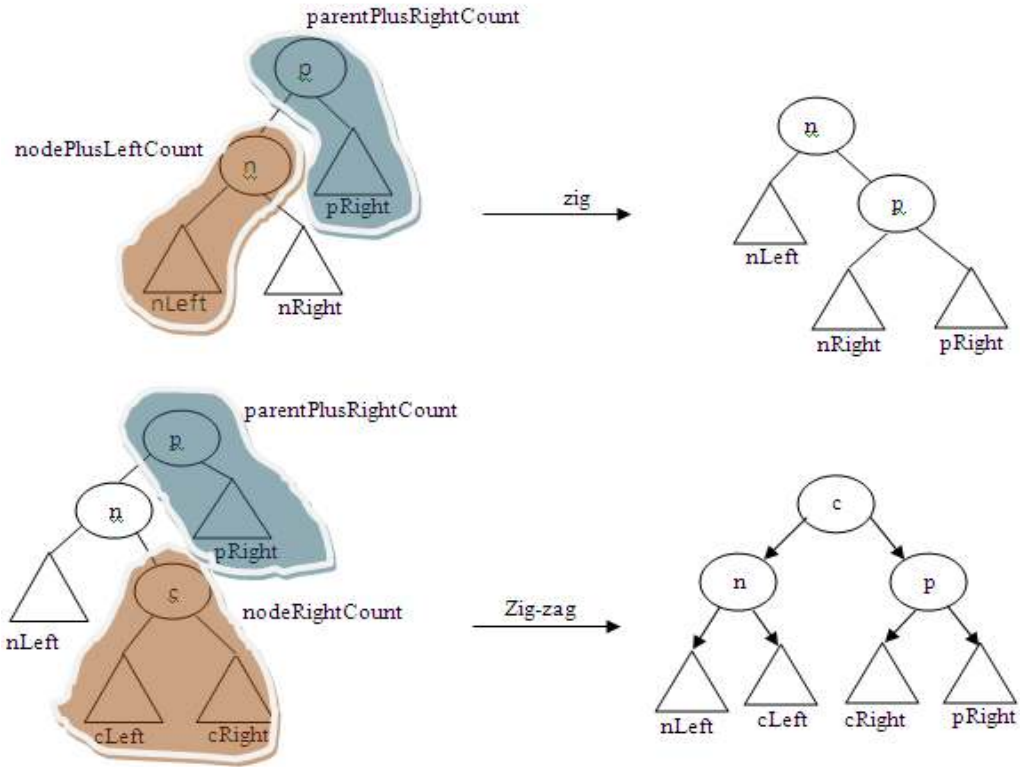


Fig. 3: Splaying steps: zig and zig-zag

and the pointcut aspect is injected into the core functionality. The most frequently accessed aspects are moved closer to the root to minimize search time. If it is a newly arrived pointcut aspect, it is inserted in the required position in the tree.

A counting-based self-adjusting search tree that is similar to splay trees moves more frequently injected nodes closer to the root. After *M* injections on *N* items, *Q* of which access some item *V1*, an operation on *V1* passes through a path length of O (log *M/Q*) while performing fewer if any rotations (Afek *et al*., 2012). In lazy splaying, in addition to the item's value, each node *w* has three counters: selfCnt, which is an estimate of the total number of operations performed on the item in *w* (number of find (w: v) and insert (w: v) operations) and rightCnt and leftfCnt, which are estimates of the total number of operations that have been performed on items in the right and left subtrees, respectively. Each find (*i*) and inject (*i*) operation increments selfCnt of the node containing *i*. When node *i* is found in the tree, all the nodes along the path from the root to *i*'s parent increase their rightCnt/leftCnt counter depending on whether *i* is in their right or left subtree, respectively (Afek *et al*., 2012; Sleator and Robert, 1985; Bronson *et al*., 2010).

Figure 3, zig-zag is carried out if the total number of accesses to the node's right subtree is greater than the total number of accesses to the node-parent and its right subtree. If zig-zag is not performed, then zig is performed if the total number of accesses to the node and its left subtree is greater than the total number of accesses to the node-parent and its right subtree.

After the rotation the rightCnt and leftCnt counters are updated to represent the number of accesses in the

Table 2: Rebalancing algorithm

```
Rebalance (Node parent, Node node)
{
nodePlusLeftCount = node.selfCnt + node.leftCnt;
parentPlusRightCount = parent.selfCnt + parent.rightCnt;
nodeRightCount = node.rightCnt;
//decide whether to perform zig-zag step
if (nodeRightCount  > = parentPlusRightCount)
{
Node grand = parent.parent;
ZigZag (grand, parent, node, rightChild);
parent.leftCnt = rightChild.rightCnt;
node.rightCnt = rightChild.leftCnt;
rightChild.rightCnt + = parentPlusRightCount;
rightChild.leftCnt  + = nodePlusLeftCount;
}
else
//decide whether to perform zig step
if (nodePlusLeftCount>parentPlusRightCount)
{
Node grand = parent. parent;
Zig (grand, parent, node, node.right);
parent.leftCnt = node.rightCnt;
node.rightCnt + = parentPlusRightCount;
    }
}
```

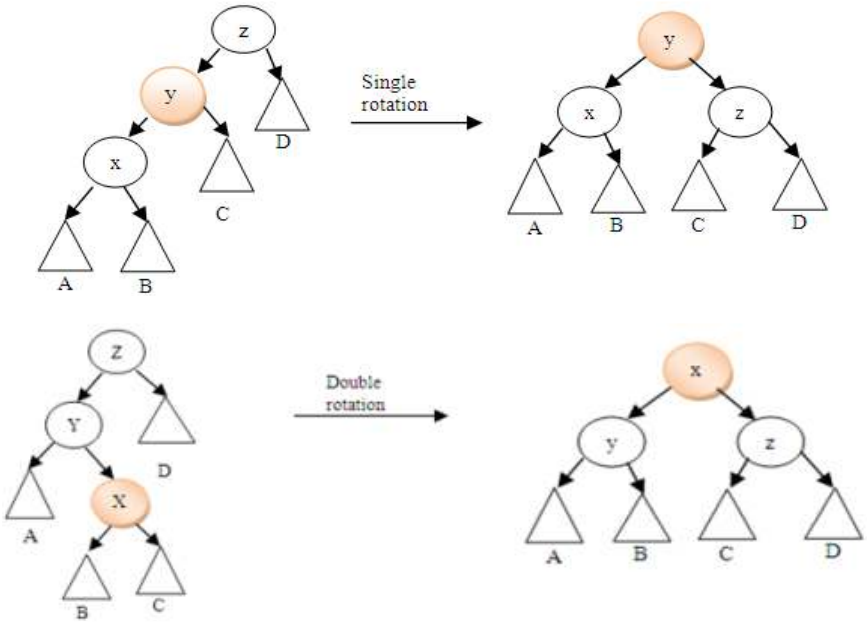


Fig. 4: Semi-splaying: the colored node is the current node for splaying

Table 3: Examples of same-origin/cross-origin URLs

URL1	URL2	Are URL1 and URL2 from the same origin?	Reason
http://pec.edu	https://pec.edu	No	Different scheme
http://pec.edu	http://pec.edu:8080	No	Different ports
http://mail.pec.edu	http://chat.pec.edu	No	Different sub domain
http://pec.edu/usr1/index.php	http://pec.edu/usr2/index.php	Yes	Path is not a part of the origin. Only scheme, host and port number

new right/left subtrees, respectively. Note that zig and zig-zag have symmetric mirror operations when the subtree at node *p* leans to the right.

To avoid a chain of nodes where all nodes are left (or right) parents of their children in the case of a descending/ascending insertion order, when a new node is inserted into the tree a re-balancing operation, as specified in Table 2, which in turn calls Table 3, is performed from this new node up to the root. If the depth is greater than 2logN, splaying is performed up to the root by using either double rotation, in which the total number of accesses to *x*'s right subtree is greater than or equal to the total number of accesses to *x*'s parent and its right subtree, or single rotation, in which the total number of accesses to *x* and its left subtree is greater than the total number of accesses to *x*'s parent and its right subtree, as shown in Fig. 4.

In contrast to the traditional self-adjusting splay tree in which each accessed item is moved closer to the root by means of a sequence of tree rotations, the counting-based splay tree performs rotations infrequently and mostly at the bottom of the tree. Therefore, it scales with the level of concurrency. The algorithms given in Table 2 and 4 are used while injecting aspects.

Table 4: Attempt injection algorithm

```
AttemptInjection (key, parent, node, height)
{    if (node = = null)
        {   node = newnode (key, node);
            node.selfCnt++;  node.rightCnt = 0;
            node.leftCnt = 0; node. height++; break  }
     else
       if (key = = node. key)
         {   if (height >= ((2 * log-size)))
             splay (node); else
           Rebalance (parent, node);
    node.selfCnt++; return node. value;    }
While (true)
{  //child in the direction of key
   Child = node.child (key);
     if (child = = null)          //Not found
         { //generate a new node and link to node
        child = newchild (key, node);
         if (height > = ((2 *log-size)))
         splay(child); return null;  }
    else
result = attemptInjection (key, node, child, height + 1);
       if (direction to child = = left)
         node.leftCnt++; else node.rightCnt++;
if (result = = null)
       {//for new node check if re-balancing needed
       // Find the current child of the parent
     //since it may have changed owing to rotation //in recursive call
curr-node = parent.child (key);
       //Child in the direction of key
        Rebalance (parent, curr-node);
   }
     Return result;
    }  //for while
}  //for begin
```

Table 5: Aspects and their node labels

Aspects	Node labels
Exception handling	1
Persistence	2
Security	3
Monitoring	4
Logging	5
Synchronization	6
Transaction processing	7

Case study 1: To construct lazy counting-based splay trees, aspects with the corresponding node labels given in Table 5 should be considered.

Injection of synchronization, persistence, transaction, exception, monitoring, logging and security aspects is shown in Fig. 5 to 8.

In an unbalanced tree, if the number of operations on *c* and on nodes in cLeft and cRight is greater than the number of operations on *p* and on nodes in pRight, then perform a zig-zag rotation. In other words, if Node+RightCnt>Parent.Self+rightCnt perform a zig-zag rotation. According to Fig. 6 and 3 (NodeRightCnt) >2 (Parent.Self+rightCnt) and therefore a zig-zag rotation is performed. While performing the zig-zag rotation, the counter values are updated as follows:

parent.leftCnt = rightChild.rightCnt
node.rightCnt = rightChild.leftCnt
rightChild.rightCnt+ = parentPlusRightCount
rightChild.leftCnt+ = nodePlusLeftCount

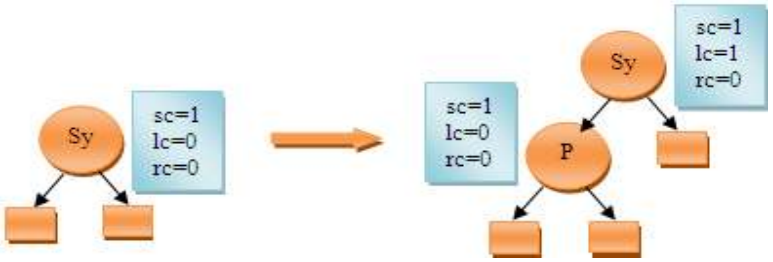


Fig. 5: Injection of synchronization and persistence aspects

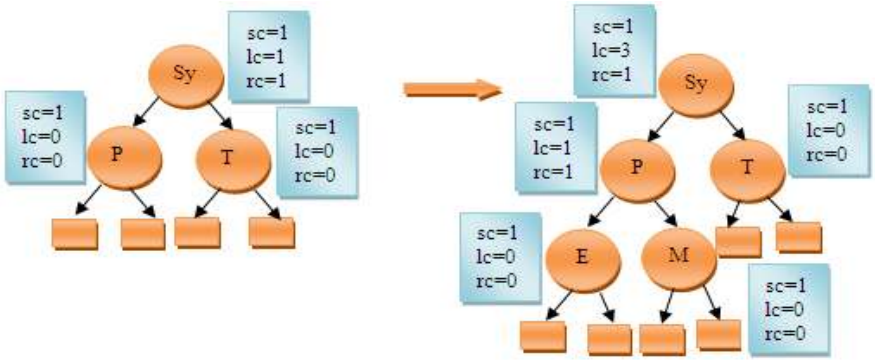


Fig. 6: Injection of transaction, exception and monitoring aspects

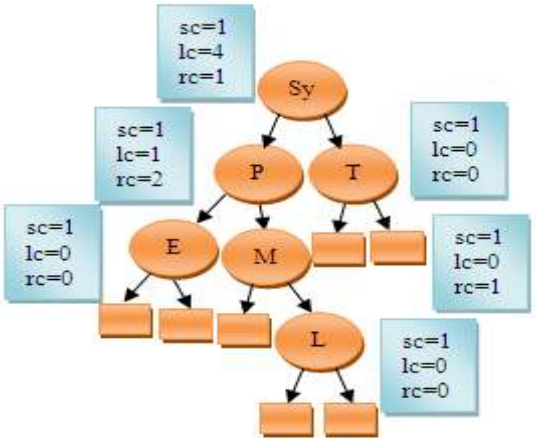


Fig. 7: Injection of logging aspect

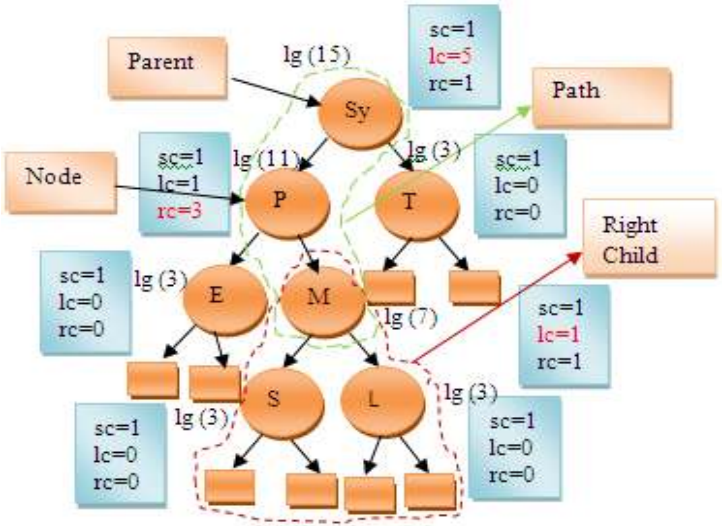
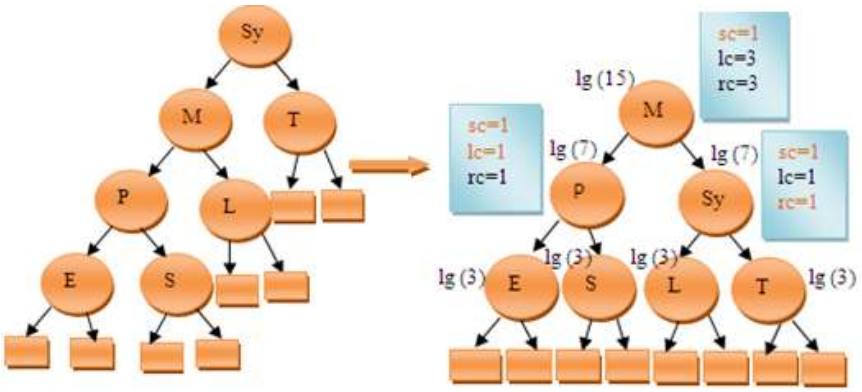


Fig. 8: Injection of security aspect



(a) (b)

Fig. 9: (a) During the zig-zag operations, (b) after the zig-zag rotations

Figure 7 shows the intermediate steps while performing a zig-zag rotation, while Fig. 8 illustrates how the most frequently accessed aspects are moved closer to the root node thereby leaving the tree in a balanced state. If the tree is in a balanced state, the cost of injecting an aspect is proportional to the depth of the node (Fig. 9).

The cost to splay node x at depth d is defined as follows. The number of comparisons increases for an unsuccessful search to avoid having to go for fictitious nodes (external nodes). The search terminates in internal nodes when the search is successful and in external nodes when the search is unsuccessful. If depth d is odd, we need to perform $d/2$ zig-zig or zig-zag rotation operations and d tree-rotation operations. Here depth d = 3 and thus, 3/2 = 1 zig-zag operations and 3 tree-rotation operations are performed. The cost to splay node x at depth d depends on the d tree-rotation operations involved.

The amortized cost to splay node x is d+Δ (r (T)). Δ (r (T)) refers to the change in balance and can be calculated using the difference between $r'(T)$ - r (T):

Before splay (M): r (T) = lg (15) + lg (11) + lg (7) + 4lg (3)

After splay (M): $r'(T)$ - = lg (15) + 2lg (7) + 4lg (3)

The amortized cost of splay (M) is 2.652076697.

Case study 2: To construct lazy counting-based splay trees, we consider aspects with the corresponding node labels given in Table 6.

Figure 10 shows the update of the counter and its organization after injecting aspects like synchronization, monitoring, logging, persistence, security and exception handling, which all come from the same origin. As before, the exception handling aspect is injected, after verifying its origin using same-origin policy techniques. If it indeed comes from the

Table 6: Aspects and their node labels

Aspects	Node labels
Exception handling	3
Persistence	4
Security	7
Monitoring	2
Logging	6
Synchronization	5
Transaction processing	1

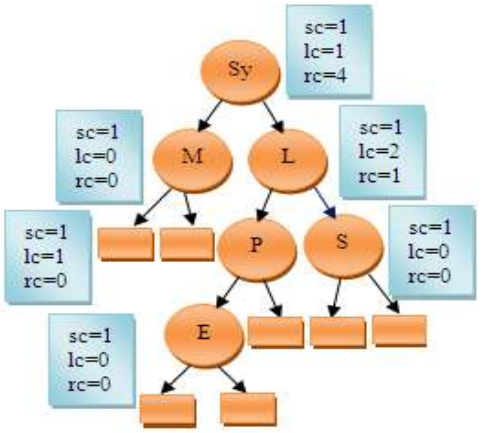


Fig. 10: Initial lazy counting-based splay tree with aspects

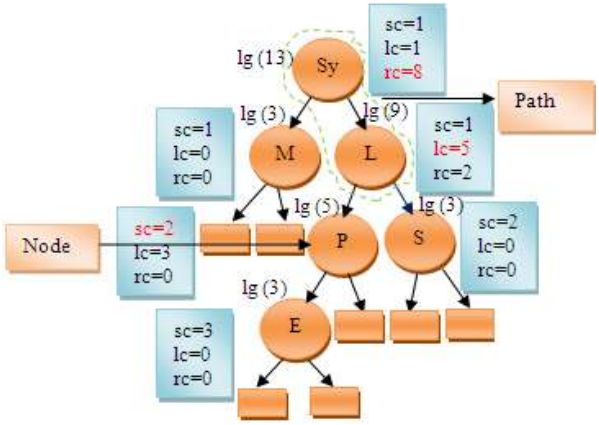


Fig. 11: Injection of persistence aspect

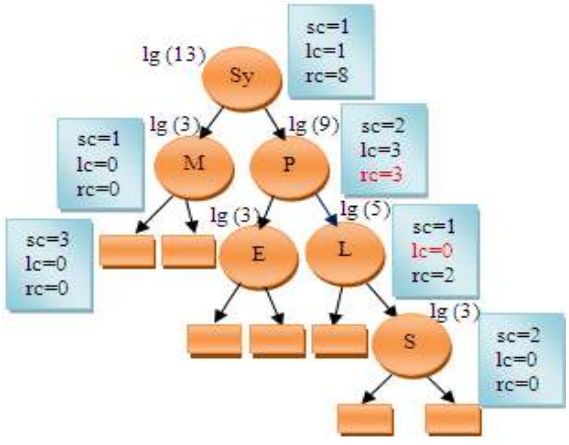


Fig. 12: After zig rotation

same origin, the exception handling self counter value is incremented by one and the rightCnt/leftCnt values on the path from the exception handling's parent to the root node are incremented by one depending on whether the exception occurred in the left/right subtree. Consequently, the synchronization aspect's rightCnt, the logging aspect's leftCnt and the persistence aspect's leftCnt values are all incremented by one. After updating the counter values, either semi-splaying or rebalancing operations are performed if balancing is needed. The same procedure is applied when injecting security, exception and persistence aspects. When injecting a persistence aspect, the tree becomes unbalanced as shown in Fig. 11.

In an unbalanced tree if the number of operations on *n* and nodes in nLeft is greater than the number of operations on *p* and nodes in pRight, then perform a zig rotation. That is, if Node+leftCnt> parent.Self+rightCnt then perform a zig rotation. From Fig. 11 we see that if (2+3) Node+leftCnt> (1+2) parent. Self+rightCnt we perform a zig rotation. While performing the zig rotation the counter values are updated as follows:

parent.leftCnt = node.rightCnt

node.rightCnt+ = parentPlusRightCount

After performing the zig rotation the counter values are updated as shown in Fig. 12.

The cost to splay node *x* at depth *d* is defined as follows. If depth *d* is even, we need to perform (d-1)/2 zig-zig or zig-zag rotations or 1 zig operation and *d* tree-rotation operations. Here depth *d* = 2 and thus, 1 zig operation and 2 tree-rotation operations are performed:

Before splay (P): $r(T) = \lg(13) + \lg(9) + \lg(5) + 3\lg(3)$

After splay (P): $r'(T) = \lg(13) + \lg(9) + \lg(5) + 3\lg(3)$

The amortized cost of splay (P) is 2.0.

Applying the lazy counting-based splay tree: Good security in a system reduces the chance of malicious or unintended actions outside the designed usage affecting the system and prevents the discovery or loss of information. Improving security can also boost the reliability of the system by reducing the chances of a successful attack that damages system operation. Securing a system implies defending the resources and avoiding illegitimate access to or alteration of the information. Composing a crosscutting concern into a requirements model may result in conflicts that have to be solved. It is possible that crosscutting concerns can cause contradictory situations in a system (Table 7 and 8).

Table 7: Contribution of response time

Name	Response time
Description	Period of time in which the system must respond to a service
Priority	Very important
Decomposition	None
Contribution	(-) to security and (-) to multi-access

Table 8: Contribution of security aspects

Name	Security
Description	Restricts access to the system and to the data by using a same-origin policy
Priority	Very important
Decomposition	Integrity and confidentiality
Contribution	(-) to response time and (+) to correctness

We have come across a situation where during the composition of crosscutting concerns with functional concerns, conflicting activities may occur. For example, response time and security are two crosscutting concerns that affect a system. When trying to compose these concerns, a conflict will occur, since both crosscutting concerns contribute negatively to each other. Thus, a tradeoff has to be found in terms of which crosscutting concern should have the highest priority and be composed first.

This contribution can be positive or negative. If two (or more) crosscutting concerns contribute negatively to each other, a conflicting situation occurs if and only if these crosscutting concerns influence the same set of requirements. To resolve these kinds of conflicts, which affect the whole system or parts thereof, a tradeoff is discussed with the stakeholders. In this situation, we propose the application of a lazy counting-based splay tree as this requires less time to check for operations like lookup () and attempt inject () and hence we need not perform tradeoff analysis between crosscutting concerns like response time and security.

METRIC EVALUATION AND DISCUSSION

The results are promising and show good potential for lazy counting-based splays, which is capable of analyzing the overall performance of splay trees versus lazy counting-based splay trees. Interesting results are obtained from this comparison. Both splay trees and counting-based splay trees use implicit caching by bringing the aspect to the root element and taking advantage of locality in incoming lookup requests for the aspect. Locality in this context refers to looking for the same aspect several times. A stream of requests exhibits no locality if every aspect is equally likely to be injected at each point. For our applications, locality does exist since aspects tend to be injected repeatedly (Fig. 13).

In lazy counter-based splay trees, according to the splay rotations involved, since most frequently accessed aspects move closer to the root, the depth of the tree is reduced and hence the delay in rotations tends to be stable (Fig. 14).

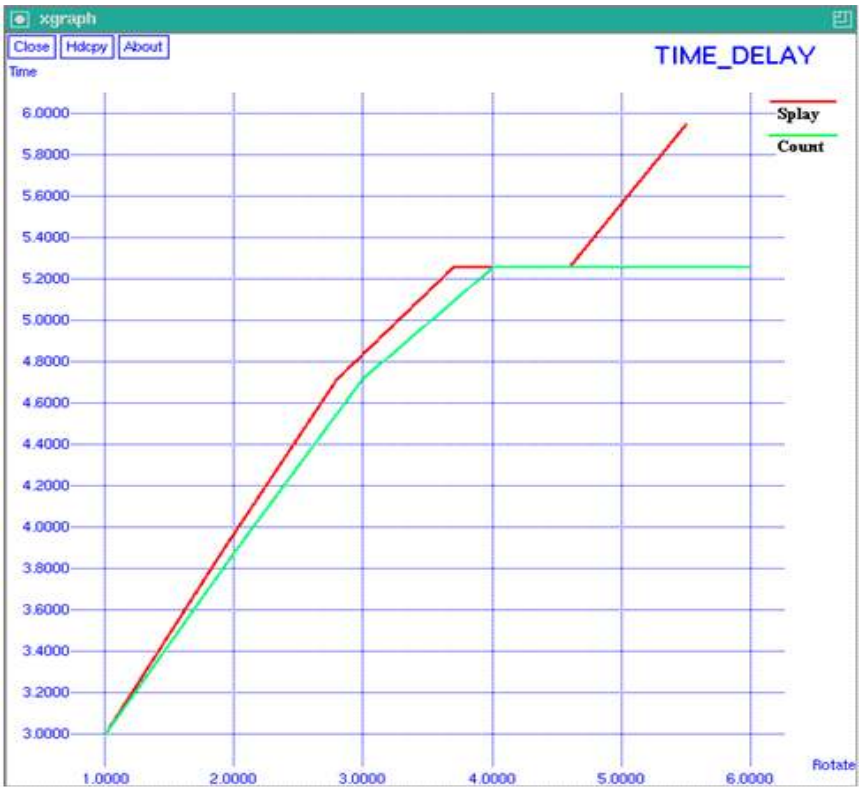


Fig. 13: Delay versus number of rotations

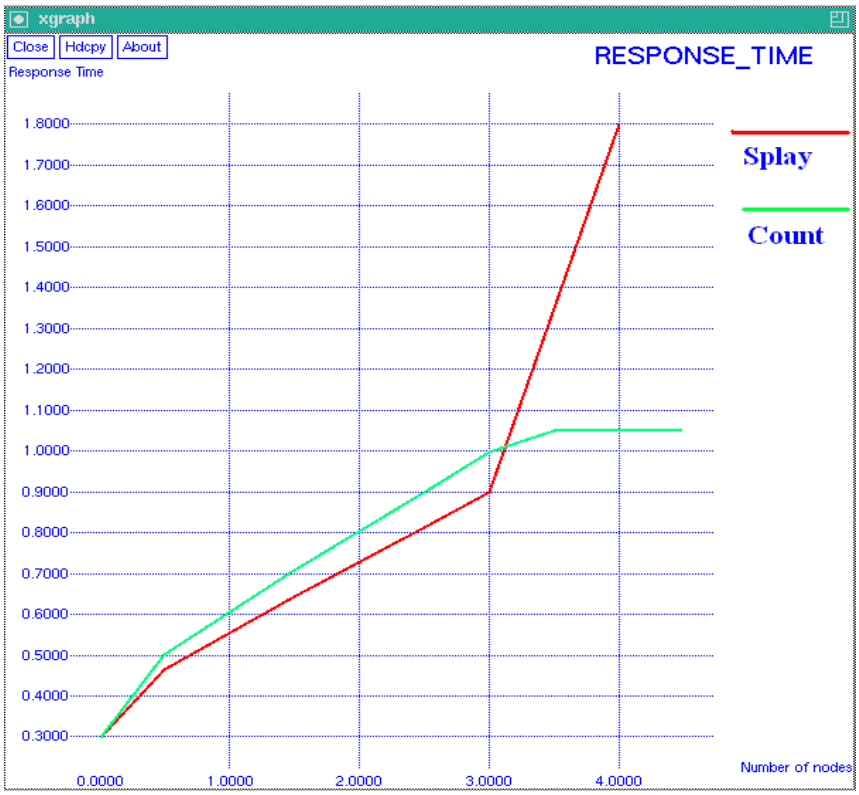


Fig. 14: Response time versus number of nodes

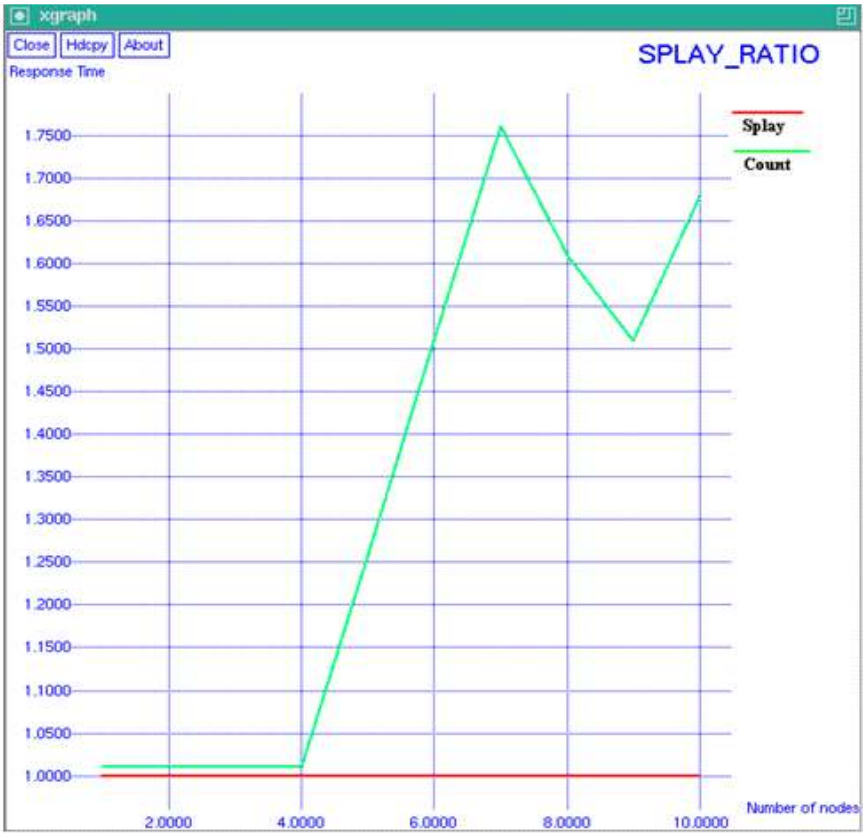


Fig. 15: Splay ratio versus number of rotations

In counter-based splay trees, over a period of time as the number of most frequently accessed nodes increases, the response time is reduced since most frequently accessed aspects are moved closer to the root (Fig. 15).

Comparing the splay ratio in both splay trees and lazy counter-based splay trees, by using the formula splay ratio = number of operations/number of splay, its values are in increasing order in counter-based lazy splay trees as the number of splay operations is largely reduced owing to the application of the counter-based technique.

CONCLUSION AND RECOMMENDATIONS

Conflicts between software quality attributes are common. Poor quality eventually affects cost and schedule because software requires fine-tuning, recoding, or even redesign to meet original requirements. Design flaws and policy errors or bugs are some of the sources of security flaws in software. This research study has been used to check whether pointcut aspects belong to the same origin and to apply the lazy counter-based splaying technique to reduce the time complexity needed to inject the legitimate pointcut aspects with the core functionality. This approach can be used effectively when implementing tradeoff analysis among aspects like security and response time. Security is a primary concern in software development and has generated a great deal of awareness amongst experts. Furthermore, few web evangelists argue that the same origin policy is too strict to block genuine third party aspects, which are essential for strict security in an application. As a future work, stack-based methods will be used to find fraudulently injected aspects even if they come from the same origin.

REFERENCES

Afek, Y., H. Kaplan, B. Korenfeld, A. Morrison and R.E. Tarjan, 2012. CBTree: A practical concurrent self-adjusting search tree. Proceeding of the 26th International Conference on Distributed Computing (DISC, 2012), pp: 1-15.

Bell, D.E., 2005. Looking back at the bell-la padula model. Proceeding of the 21st Annual Computer Security Applications Conference, pp: 15-351.

Boström, G., 2004. A case study on estimating the software engineering properties of implementing database encryption as an aspect. Proceeding of the 3rd International Conference on Aspect-oriented Software Development. Lancaster, UK, pp: 1-6.

Bronson, N.G., J. Casper, H. Chafi and K. Olukotun, 2010. A practical concurrent binary search tree. Proceeding of the 15th ACM SIGPLAN Symposium on Principals of Parallel Programming.

De Win, B., B. Vanhaute and B. Decker, 2001. Security through aspect-oriented programming. Proceeding of the IFIP TC11 WG11.4 1st Working Conference on Network Security Advances in Network and Distributed Systems Security. Leuven, Belgium, pp: 125-138.

De Win, B., B. Vanhaute and B. De Decker, 2002. How aspect oriented programming can help to build secure software. Informatica, 26(2): 141-149.

Hermosillo, G., R. Gomez, L. Seinturier and L. Duchien, 2007. AProSec: An aspect for programming secure web applications. Proceeding of the 2nd International Conference on Availability, Reliability and Security (ARES'07), pp: 1026-1033.

Huang, Y.W., F. Yu, C. Hang, C.H. Tsai, D.T. Lee and S.Y. Kuo, 2004. Securing web application code by static analysis and runtime protection. Proceeding of the 13th International Conference on World Wide Web, pp: 40-52.

Izaki, K., K. Anaka and M. Takizawa, 2001. Information flow control in role- based model for distributed objects. Proceeding of the 8th International Conference on Parallel and Distributed Systems. Kyongju City, Korea, pp: 363-370.

Kawauchi, K. and H. Masuhara, 2004. Dataflow pointcut for integrity concern. Proceeding of AOSD 2004 Workshop on AOSD Technology for Application Level Security (AOSDSEC).

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten and J. Palm, 2001. Getting started with aspect J. Commun. ACM, 44(10): 59-65.

Lee, J., K.H. Hsu, S.J. Lee and W. T. Lee, 2012. Discovering early aspects through goals interactions. Proceeding of the 19th Asia-Pacific Software Engineering Conference (APSEC, 2012), 1: 97-106.

Mourad, A., M.A. Laverdiére and M. Debbabi, 2008. An aspect-oriented approach for the systematic security hardening of code. Comput. Secur., 27(3-4): 101-114.

Ramachandran, R., D.J. Pearce and I. Welch, 2006. Aspect j for multilevel security. Proceeding of the 5th AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS). Bonn, Germany.

Sabelfeld, A. and A.C. Myers, 2003. Language-based information-flow security. IEEE J. Sel. Area. Comm., 21(1): 5-9.

Simic, B. and J. Walden, 2013. Eliminating SQL injection and cross site scripting using aspect oriented programming. In: Jurjens, J., B. Livshits and R. Scandariato (Eds.), ESSOS 2013. LNCS 7781, Springer-Verlag, Berlin, Heidelberg, pp: 213-228.

Sleator, D.D. and E.T. Robert, 1985. Self-adjusting binary search trees. J. ACM (JACM), 32(3): 652-686.

Stewart, J.M., E. Tittel and M. Chapple, 2005. CISSP: Certified Information Systems Security Professional Study Guide. 3rd Edn., Sybex Inc., San Francisco.

Viega, J., J.T. Bloch and P. Chandra, 2001. Applying aspect-oriented programming to security. Cutter IT J., 14(2): 31-39.

Wand, M., G. Kiczales and C. Dutchyn, 2004. Semantics for advice and dynamic join points in aspect oriented programming. ACM T. Progr. Lang. Sys. (TOPLAS), 26(5): 890-910.

Win, B.D., F. Piessens, W. Joosen and T. Verhanneman, 2002. On the importance of the separation-of-concerns principle in secure software engineering. Department of Computer Science, Katholieke Universiteit, Leuven.

Zdancewic, S., 2004. Challenges for information-flow security. Proceeding of the 1st International Workshop on Programming Language Interference and Dependence (PLID, 2004). Verona, pp: 1-5.