INTRODUCTION

Nowadays, Cyberspace is no longer science fiction; it’s a substantial reality. Millions of people depend on highly interconnected computing environments in order to conduct routine as well as critical tasks and transactions.

The evolution, from batch processing systems to machine-to-machine commerce and currently to distributed computing using component-based development and downloaded executable content, has grown in scope and magnitude the security and privacy risks and concerns of Web-based and wireless applications. Even more, the security and economy of every nation relies on the need for primary security services such as confidentiality, integrity, and availability and also authentication, access control, and non-repudiation since, when these systems perform badly or do not work at all, they put life, liberty, and property at risk.

In this context, trust and trustworthiness, the foundations of security, play a very important role. Home-owners trust lock manufacturers to create quality locks to protect their homes. Some locks are trustworthy; others are not. Businesses trust security guards to admit only authorized personnel into sensitive areas. Some security guards should be trusted; some should not. Common Gateway Interface (CGI) programmers trust users to provide valid inputs to the data fields on Web pages. Although most users can be trusted, some cannot.

Many corporations trust that their employees will not attack the information systems of the company or that their software developers will not leave back doors or other artifacts in their code that could potentially compromise the systems’ security. Even during the software development process, a development group may make the erroneous trust assumption that another group is handling the security of a component when in reality security is never codified.

The basis for these trust relationships and how they are formed can dramatically affect the underlying security of any system - be it home protection or online privacy.

All these prove that software that determines how every modern information system behaves is one of its essential elements. Today’s software problems, that are usually the root cause of all common computer security problems, lead to spectacular real world failures of many different kinds, including reliability and safety problems. Security holes and vulnerabilities are the result of bad software design and implementation. Buggy software and malicious (mobile) code make our computing environments untrustworthy. The need to avoid the “penetrate-and-patch” approach and build systems “with security in mind” has become an imperative. Software assurance for security has become extremely important.

However, building secure software and applications is not a straightforward task due to lack of expert knowledge by average software developers in writing functional software that is also secure, lack of education, as few classes cover this material and books that cover it well have just now started to appear, lack of coherent design time methodologies and tools for building software with security in mind, lack of programming languages with significant security architectures (with the exception of the Java security model), and structural difference between an application and the required security solution, since usually the security code is scattered throughout the functionality of the application.

Based on these, we clearly understand that security (which is usually a non-functional requirement) is a crosscutting concern difficult to be managed when it spreads throughout the functional code of an application with a higher risk of security bugs due to the code tangling effect. Aspect-Oriented Programming (AOP) is a programming paradigm/methodology that promises to tackle this problem by enabling the modularization of crosscutting concerns (like security). The research being conducted now in this area promises that AOP can lead to more reliable software and can help in building secure software.

In this special issue, we define the problem of secure computing in untrustworthy environments, identify the importance of software security and how AOP can help, and describe the current research challenges and areas.
The combination of AOP and software security forms a multidisciplinary approach adopting concepts from the areas of software engineering and security.

**THE NEED FOR SECURE COMPUTING**

Computing environments have changed significantly their face and shape as the computing revolution moved from one wave to the other in the past years.

The first wave constituted the mainframe era where universities established glass-house computing rooms for students, faculty, researchers and administrators to benefit from this invention. Our relationship to computers in this area was one of service: they were rare and expensive and we serviced them.

The second wave, in which we are still deeply immersed, is the personal computing era where, as the name proves, the relationship with the computer is personal. People use at least one Personal Computer (PC); more often they use several in several locations or use laptops for the everyday work.

The third wave, which is now on the move, is the era of ubiquitous computing (ubicomp in short) where computers can be found in watches, toasters, ovens, cars, wallets, etc. Inevitably, these computers have become pervasive, talk to one another, and form the invisible computational infrastructure of our lives. The relationship to computers in the ubicomp era is the inverse of the mainframe: computers have become common and inexpensive and they serve us.

People in the digital age have in their hands a large number of computing environments for conducting their transactions such as PCs, Network Servers (Web Servers, Mail Servers, FTP Servers, etc.), Personal Digital Assistants (PDAs), Mobile Phones, Set-Top Boxes (STBs), Laptops, Handheld devices, Notebooks and so many others. In addition, these devices have enhanced capabilities like extensibility via mobile code execution (ActiveX controls, plug-ins, helper applications, scripts, applets, etc.) and ability to communicate via wireless infrastructures in an ad hoc manner (IEEE 802.11, Bluetooth, etc.).

All these issues increase the security and privacy risks for the users and make them lose their trust on the devices. For example, if a user utilises a PC (home or public e.g. from an Internet cafe) for conducting e-banking transactions, for telemedicine (e-health), for e-voting, for e-commerce transactions paying via e-cash (e-wallet), etc. she cannot be sure that a Trojan Horse will not transmit her private information (credit card number, secret ballot, passwords, etc.) to a third party, or that a keystroke analyzer/recorder or keyboard/password sniffer is present and intercepts the information she gives or that even a malicious cookie is misbehaving.

In this context of uncertainty and untrustworthiness, it is obvious that the user needs assurance for secure computing and proper security services, such as confidentiality (prevention of unauthorised disclosure of information), integrity (prevention of unauthorised modification/tampering of information), availability (prevention of unauthorised outage/failure of computing resources), privacy/anonymity (the right not to reveal personal/sensitive information), authentication (the right to know to whom am I talking to), and non-repudiation/auditing (be sure that the other party/peer will not deny having participated in a transaction).

Users need support in order to increase their trust on the modern highly interconnected, extensible, complex, open, and distributed computing environments on which they depend. The security of computer systems should be ensured in four fronts: client, data transport, server and operating system security. A lot of security mechanisms have been described in the literature. In this special issue we will focus on mechanisms that provide software protection.

**THE AOP PARADIGM AS A SOLUTION**

The execution of malicious downloaded mobile code and buggy software increase the untrustworthiness of contemporary computing environments. From this fact it is clear that software security is an emerging and important concern for the smooth operation of our society, as it becomes more and more dependent and reliant on software systems.

However, since security is a non-functional requirement, it is not the utmost concern for system developers –even in systems where security threats might be easily perceptible. Hence, software system security is typically an afterthought and may be considered seriously if the functional requirements are met and the project is within the schedule and budgets, which is seldom the case.

Moreover, usually an application and the corresponding security logic have a structural difference. For example, code to log events in an audit trail/file, or code that implements an access control model is often spread and scattered among many different classes leading to code tangling and complexity. In that sense, security is a crosscutting-concern since it cuts across many classes or functions in the program. Since the primary goal of the AOP paradigm and the Aspect-Oriented Software Development (AOSD) methodology is the “modularisation of crosscutting concerns” it can be applied to software security in such a way that more reliable and secure software can be built by the developers.

In the following sections we will describe in more detail the AOP/AOSD concepts and techniques and their
application to software engineering for security. We will conclude by describing the research challenges and the current areas where research is being conducted regarding the application of AOP to security.

**AOP DESCRIPTION**

Aspect-Oriented Programming (AOP) is a new programming paradigm/methodology first introduced in 1997 from the Xerox Palo Alto Research Center (PARC) Software Design Area by Gregor Kiczales that explicitly promotes the notion of separation of concerns.

The separation of concerns (SoC) principle is one of the essential principles of software engineering. It was first introduced by Dijkstra and Parnas as an answer to control the complexity of ever-growing programs. This principle states that a given problem involves different kinds of concerns (i.e. particular goals, concepts, or areas of interest), which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility, and reusability.

In technology terms, a typical software system is a combined implementation of several core (business logic) and system-level concerns. For example, a credit card processing system’s core concern processes payments while its system-level concerns handle logging, performance, transaction integrity, data persistence, authentication, security, multithreaded safety, error checking, etc. The following figure describes this issue:

![Figure 1: A credit-card processing system as a set of crosscutting concerns.](image)

Development-process concerns, such as comprehensibility, maintainability, traceability, and evolution ease play also their role. But many such concerns - known as crosscutting concerns - tend to affect multiple modules. Using current programming methodologies, crosscutting concerns span over multiple modules, resulting in systems that are harder to design, understand, implement, and evolve.

The concern decomposition for a software system resembles a demultiplexer where the set of requirements can be seen as an interleaved stream passing through the concern-identifier demultiplexer, which separates each concern. The following figure depicts this issue:

![Figure 2: Concern decomposition - the demultiplexer analogy.](image)

The problem is obvious. Although crosscutting concerns span over many modules, current implementation techniques tend to implement these requirements using one-dimensional methodologies, forcing implementation mapping for the requirements along a single dimension. The single dimension tends to be the core module-level implementation. The remaining requirements are tangled along this dominant dimension. Thus, the requirements space is an N-dimensional space whereas the implementation space is one-dimensional. Such a mismatch results to code tangling, and code scattering that affect the software design and development by leading to poor traceability, lower productivity, less code reuse, poor quality, and difficulty to evolve.

AOP promises to better separate concerns than previous methodologies, thereby providing modularisation of crosscutting concerns. The idea is to implement individual concerns in a loosely coupled fashion and then combine these implementations to form the final system. AOP is not a substitute for Object-Oriented Programming (OOP) but it constitutes its logical evolution. OOP promotes modularised implementations of common business concerns while AOP promotes modularised implementations of crosscutting business and technical concerns. In OOP the common concern’s implementation is called a class, while in AOP the crosscutting concerns implementation is called an aspect.

A large community of researchers/practitioners are dealing with the AOP paradigm (AOSD). AOP involves three distinct development steps:

1. **Aspectual decomposition**: Decomposition of the requirements to identify crosscutting and common concerns. Separation of module-level concerns from crosscutting system-level concerns. For example, in the afore-mentioned credit card module example, three concerns are identified: core credit card processing, logging, and authentication.

2. **Concern implementation**: Implementation of each concern separately. For the credit card processing...
3. The developer would implement the core credit card processing unit, the logging unit, and the authentication unit.

4. Aspectual recomposition: In this step, an aspect integrator specifies recomposition rules by creating modularization units – the aspects. The recomposition process, also known as weaving or integrating, uses this information to compose the final system. For the credit card processing example, it would be specified, in a language provided by the AOP implementation, that each operation's start and completion would be logged. It would be also specified that each operation must clear authenticate before it proceeds with the business logic.

Just like any other programming methodology implementation, an AOP implementation consists of two parts: a language specification and an implementation. The language specification describes language constructs and syntax. The language implementation verifies the code's correctness according to the language specification and converts it into a form that the target machine can execute. In the literature, we can find a number of language implementations that support the AOP paradigm. These are:

- **AspectJ**: A freely available AOP implementation for Java from Xerox PARC, which forms a general-purpose aspect-oriented Java extension.
- **HyperJ**: A language that supports advanced multi-dimensional separation and integration of concerns in standard Java software.
- **Java Aspect Components (JAC)**: An AO and also OO development environment, which also offers an AO application server in Java.

In the context of the security field, the promotion of separation of concerns through AOP means that the main program does not need to encode security information; instead, it is moved into a separate independent piece of code, the aspect.

With such a modularisation between an application's business logic and its non-functional requirements (such as security), it is easier for the developer to concentrate on the core functionality of the application, while a security expert can analyse the security requirements and formulate the security policy in concrete aspects that will be woven into the application when needed. Moreover, if these security requirements are properly designed as aspects, they can be reused for other applications also.

### RESEARCH IN APPLYING AOP TO SECURITY

Thus, no security expertise is needed by the average programmer and even if the security expert is not there to provide assistance, the security features can shield the application and the developer will be able to reasonably secure the code written. This is one of the major benefits of AOP when applied to security.

Currently, a number of researchers are dealing with the issue of application of AOP to security and have provided concrete results. In the sequel, we will describe in more detail these approaches.

One team of researcher has developed an aspect-oriented extension to the C programming language. They designed a language for defining security concerns and implemented a weaver that generates and integrates security code into C programs. They started from the C programming language since it presents significant security risks.

Motivated by the fact that average developers do not have much security expertise and building systems “with security in mind” is order of magnitudes better than the “penetrate and patch” approach they built this language that can be used to: a) automatically perform error checking on security critical calls, b) implement the StackGuard technique of buffer overflow protection, inserting special code at function entry and exit, c) automatically log data that may be relevant to security, d) replace generic socket code with SSL socket code, e) automatically insert code at start-up that goes through a set of “lock-down” procedures that most programmers would not add to their programs, and f) specify privileged sections of a program and automatically request and return privileges when appropriate.

Similar to the above research, other researchers used AOP to address the concern of software security and built an Aspect-Oriented Security Framework (AOSF) in order to provide means for developing secure applications. The current implementation works for the C programming language but can be easily extended to other languages. The framework is simple, flexible and intended to be easily and seamlessly integrated into the build process for programs.

The idea is to add an extra step to the build process (the various source files being pre-processed and then compiled and linked to form an executable) by also integrating security aspects written by a security expert through an aspect weaver, which constitutes the core component of the framework. The framework has been used to address common security problems like buffer overruns, time-of-check-to-time-of-use (TOCTOU), and format strings. Two of the security aspects implemented are the Event Ordering and the Type Safety Aspect.

In another case it is argued that AOP can significantly improve the reliability of programs, particularly by modularising error-handling policies and allowing for easier maintenance and better reuse. AOP can be successfully used for exception handling in programs.
Another researcher developed a security aspect for the Java programming language, which is the result of merging the Java security model and the properties for typed applets. With this implementation, security holes caused by some bugs in the Java implementation can be avoided by integrating the aspect into the application code. Moreover, the security aspect helps programmers to deal with security concerns on two different levels: specifying policies related to domain’s entities and specifying restrictions at the code level by e.g. declaring sensitive classes and named types of the system.

The security aspect is clear and intuitive, facilitating the specification of policies by the programmer. The expressiveness of the security aspect language permits programmers to have access to the features of complex approaches without dealing directly with technical details. A prototype of the security aspect has been implemented and TXL, a generic framework for program transformations, has been used for the aspect weaver implementation.

Another team of researchers tried to answer the question “How can a security subsystem be designed such that it is reusable in many applications?”. In this context, they have developed security concerns (e.g. access control) in an aspect-oriented way using AspectJ and they tried to structure these security aspects in such a manner that they are reusable in many applications.

The result of this work was the implementation of an aspect framework for application security, which consists of generalised aspects for each of the security requirements. In this context, several concrete aspect implementations, depending on different underlying security mechanisms, may be included for the same security requirement. The deployment of the framework for a concrete application will then come down to choosing the appropriate aspects and define concrete pointcut designators for them. The corollary is that the combination of AOP and framework technology can allow the design of security subsystems that are reusable across many applications.

Finally, the access control and auditing concerns of an existing FTP server implementation (i.e. jFTFd) have been modularised through AOP providing a more practical experience on using AOSD for security.

The aforementioned research activities in the area of AOP and security prove that this programming paradigm can provide significant assistance in the development of secure software applications.

RELATED CONCEPTS

Concepts that have been introduced in the literature several years ago prove that the separation of concerns idea and the AOP paradigm were existent at that time but in another form. Indicatively, such research activities include:

The Java security model

Java provides a Security Manager that allows programmers to define a security policy for a system in such a way that the functional code (the base code) and the security code are largely separated. Therefore, the functional code is not affected and can be written quite independently from the security specification. In this way separation of concerns is accomplished.

Byte-code transformations

Transformations in AspectJ happen on the source code level. Other tools are available that work on the level of the byte code providing the advantage that aspects can be added even if the source code is not available but with the drawback that on the level of byte code a lot of application information is lost.

The Java Object Instrumentation Environment (JOIE) is a toolkit for constructing load-time transformations of Java classes. An enhanced class loader calls user-supplied transformers that specify rules/directives for transforming target classes. This allows the users to select transformations that add new features (e.g. in our case security) customise the implementation of existing features, and apply changes to all classes in the environment.

Wrappers

The Generic Software Wrappers approach uses wrappers (state machine specifications that listen for events and take appropriate actions) to bring components under the control of a security policy. The wrappers act as localised reference monitors for the wrapped components. The emphasis is on binary components and their interaction with an operating system via system calls.

Wrappers are defined using a Wrapper Definition Language (WDL) and are instantiated as components are activated. They monitor and modify the interactions between the components and the operating system in such a way that the security policy is respected. Generic policies for access control, auditing, intrusion detection, etc. can be specified using the WDL.

Policy languages for security

These are systems that allow a more declarative description of security properties for an application include. Indicatively, we name Naccio, Ariel, PolicyMaker, Keynote, and Deeds.
Meta-level architectures

Based on the principle of separating security policy and dynamically enforcing it on applications these techniques make use of behavioural reflection as a mechanism for implementing code modifications within an abstract framework based on the semantics of the underlying programming language.

Researchers have developed a reflective version of Java called Kava that uses byte-code rewriting techniques to insert predefined hooks into Java classes at load time. This makes it possible to specify and implement security policies for mobile code in a more abstract and flexible way. Their approach is based on the use of meta-level architectures and meta-object protocols to provide flexible fine-grained control over the execution of components. The meta-object protocol implements the security mechanisms that enforce security policies upon application code. This effectively allows security checks to be inserted directly into compiled code, thus avoiding the need to recode applications in order to add application specific security checks.

Language-based security

It is a set of techniques based on programming language theory and implementation, including semantics, types, optimisation, and verification brought to bear on the security question. The building blocks of language-based security are program rewriting and program analysis. By rewriting a program we can ensure that the result is incapable of exhibiting behavior disallowed by some security policy at hand. And by analysing a program, we ensure only those programs that cannot violate the policy are ever given an opportunity to be executed. Indicatively, we name the technologies of Inline Reference Monitors (IRMs) and Security Automata Software-Fault Isolation (SASI).

Code certification

A certifying compiler is a compiler that, when given source code satisfying a particular security policy, not only produces object code but also produces a certificate, machine-checkable evidence that the object code respects the policy. Proof Carrying Code (PCC), Typed Assembly Language (TAL) and Efficient Code Certification (ECC) fall in this category.

Based on the fact that software security is the cornerstone of this problematic situation we identified the Aspect-Oriented Programming (AOP) paradigm as a practical solution in providing secure software. We described in detail the concept and its evolution and identified the current research challenges and areas for further developments. Also similar concepts that exist in the literature were depicted, which prove that the concept evolved during the past years.

The most important research area identified from the literature is that there is a strong need for a more declarative description of security properties for an application. We need to express reasonable security policies (as layered sets of abstractions) that can be directly transformed into technology enforcement solutions. We need to think of the right abstractions the description of the security policy will consist of and, if possible, automate the generation of concrete pointcuts based on a simplified high-level description. Finally, we have to think of what class of security policies can be enforced using the separation of concerns approach.

CONCLUSIONS

In this special issue we have identified the problem of the untrustworthiness of the contemporary computing environments in the digital age, on which our society and citizens are highly dependent. We have also identified the need for secure computing that can be assured via security services.