Auto-generation of Least Privileges Access Control Policies for Applications Supported by User Input Recognition

Sven Lachmund and Gregor Hengst

DOCOMO Euro-Labs, Munich, Germany

Abstract. Applications are typically executed in the security context of the user. Nonetheless, they do not need all the access rights granted. Executing applications with minimal rights (least privileges) is desirable. In case of an attack, only a fraction of resources can be accessed. The state-of-the-art on application-based access control policy generation has limitations: existing work does not generate least privileges policies, policies are not always complete and the process requires complex manual interaction. This paper presents an almost fully automated approach which counters these limitations. It achieves this by (1) extending a static analysis approach by user input recognition, by (2) introducing a new runtime approach on user input recognition which is based on information tracking and Aspect-Oriented Programming and by (3) combining the other two contributions with some of the existing work. The combined approaches are integrated into the software development life cycle and thus, policy generation becomes practicable. A prototype of the runtime approach is implemented which proves feasibility and scalability.

1 Introduction

In today's mainstream operating systems applications are typically executed with the security context of the user. Since applications are used for a specific purpose, they do not need all the access rights of the user. Applications should rather be executed with only those access rights they actually need (least privileges [1]).

If the application has a vulnerability which can be exploited by an attacker, allowing the attacker to control the application, the attacker is able to access the resources which the application is permitted to access. If the application has restricted access, potential damage of the attack can be confined. Due to complexity and extensibility of today's software, applications typically have vulnerabilities [2,3].

To execute applications with least privileges, applications have to be assigned access rights individually, as the purpose of applications and the resources they need to access vary significantly. Generic policies and protection domains are not specific enough. If applications have their individual access rights, limited to the minimum, they can *execute normally*, i.e. they have all the access rights they need to carry out their operations, but not more. Applications access two categories of resources. The first category – the *system resources* – comprises those resources which the application accesses to carry out its operations independently of any user interaction. The second category – the *user resources* – comprises resources chosen by the user while interacting with the application. For example, a library that is loaded by an application to invoke a function represents a system resource and a text document accessed by a user in a text editor represents a user resource. It is not known prior to execution which user resources a user will access at runtime. In contrast, access to system resources can be derived from the application's code.

There is existing work, both on research level [4,5,6,7] and product level [8] (see Section 2), that automates generation of policies individually per application. While this existing work generates least privileges policies for system resources, it fails doing so for user resources. It fails due to permitting access to all user resources which might be accessed by the user during execution by adding *generic permissions* to the policy. An example for these generic permissions is to permit access to the entire home directory of a user. This overapproximation of access rights violates the principle of least privileges.

The objective is to reduce the set of access rights by discarding access rights to user resources in order to generate a least privileges policy. Treating system resources differently from user resources is the key. In this paper, access rights for system resources are collected and policies are generated using the existing work. However, for treating access to user resources, user input is identified and its propagation through the control flow of the application is analysed. If data that is input by the user is used as resource identifier at a permission check, the access is considered as access to a user resource. User-initiated resource access is determined that way. Permissions for accessing user resources are not added to the policy. The generated policy – the *application policy* – only consists of all the necessary access rights to system resources. The entire process is automated in order to minimise the involvement of the developer. Policy generation is performed at development time as a kind of side-task during implementation and testing. Technically, user input recognition is based on static analysis and runtime observations of the application's code.

The contributions of this paper are:

– Improving a static analysis-based approach by integrating user input recognition,
– introducing a new scheme for user input recognition based on user input tracking with aspect-oriented programming and
– Combining existing work on static analysis and runtime observation with the two other contributions.

Combining all these contributions eliminates major drawbacks of the existing work on policy auto-generation: overapproximation is eliminated, completeness of the policies is given and manual user interactions are eliminated. A prototype has been implemented which proves feasibility and scalability of the taint tracking approach.

Executing the application with this application policy being enforced at runtime would prevent the user from accessing any user resource in the application. Therefore the application policy is adapted on the target system where the application is executed. This can be done dynamically at runtime upon user interactions. Whenever a user chooses a resource in the application, the corresponding access right is added to the policy. Consequently, the application can only access user resources chosen by the user. This satisfies the principle of least privileges. The user perception is the same as in systems based on the object-capability security model (see Section 2.4). An alternative is to adapt the policy statically at deployment time or at load time, by specifying which user resources are accessible. We already elaborated various approaches for this adaptation. Some of them are similar to the user input recognition at development time (see Section 3). However, other approaches are beyond the focus of this paper.

Since the policy is generated by the developer and augmented on the target system, responsibilities are split: the developer defines the access rights the application needs and the target system only has to define which resources the user should be able to access. This results in policy generation being practicable for all the involved stakeholders. In contrast, existing work involves the user in complex manual tasks, as all the policy is generated on the target system. It is difficult for the user to determine which access rights to system resources an application needs. Consequently, existing work is not widely used in practice.

The paper is organised as follows. Section 2 discusses related work. Section 3 contributes distinction of access to system resources and user resources. It also presents the automated application policy generation process. The prototype implementation of the runtime observation approach is addressed by Section 4. Section 5 illustrates the contribution applied on an application. An evaluation of the contribution and the prototype is provided by Section 6. Section 7 addresses issues to be considered, such as embedding the presented work in the software development life cycle (SDLC) and future work. Section 8 concludes the paper.

This work has been carried out based on the object-oriented programming (OOP) paradigm [9,10]. It is assumed that the programming language and its execution environment are entirely object-oriented. Terms, such as *class*, *object*, *method*, *field*, *member*, *type* and *modifier* are used in their OOP context throughout the paper.

2 Related Work

This section is organised in line with the contributions. Section 2.1 describes the static analysis-based approach which is improved in this paper. Therefore it is discussed in depth. Section 2.2 covers relevant runtime observation approaches for generating application policies. Further approaches which are interesting but not used in this paper are briefly discussed in Section 2.3 Other related work is addressed thereafter.

2.1 Static Analysis by Call Graph

Policy generation by Koved, Centonze, Pistoia, et al. [4,5,6] is based on static analysis. It creates a call graph of applications written in Java which represents methods as nodes and method calls as edges. The call graph is used to determine which method calls result in permission checks. For each of these permission checks, the allocation site of the involved Permission class (representing access rights in Java [11]) is determined. Each Permission class contains an identifier which represents the accessed resource. The values of all of these identifiers are collected. They are put in the policy, as these resources will be accessed by the applications during execution. Libraries and applications are analysed differently.

For libraries a *summary* is created for all possible paths in the call graph of the library which start at any permission check node and end at any protected or public method. *Data flow analysis* [4,5] is applied to determine the paths. Each of the end point methods causes a permission check in the library when being invoked by the application. For the applications, these methods are entry points into the library. The summary contains all the required permissions for these calls.

Application analysis is limited to the paths in the call graph of the application that go from a start node to a node that is an entry point of a library for which a summary has been created. A set containing all the entry point nodes of all the libraries used by the application is created. It is partitioned in three subsets depending on properties of the resource identifiers. Paths are treated differently in the analysis, depending on the partition to which the entry point node where the path ends is assigned.

The first subset contains all those methods that use a string constant defined in the library to define the resource identifier. These methods are processed by a data flow analysis, like in the library analysis.

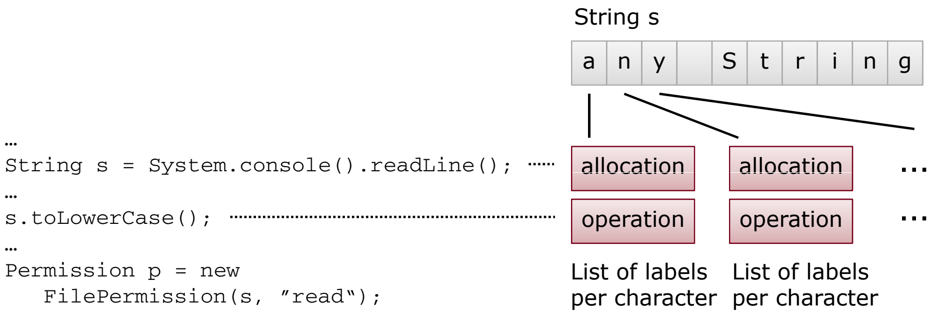
The second subset contains all those entry point methods that receive one or more String argument(s) when being invoked by the application. These arguments are allocated by the application and used as resource identifiers for the permission check in the library. These methods are processed with a more complex algorithm. The complete algorithm is presented in [6]. It starts with the *string analysis* described below to determine all possible resource identifiers in the application code that are used as arguments of entry point methods. *Slice*s [12] are created for each of these String arguments to determine their propagation through the application. The slices identify the propagation paths in the call graph that belong to the String argument, without introducing paths that do not exist in the application's control flow.

The *string analysis* is a processing step of the application analysis where String objects of the application are analysed. The string analysis is capable of tracking all instantiations and modifications (e.g. concatenation) of strings representing resource identifiers. *Transforms* [6] of String modifying operations are created to determine the output String when an input String is provided. The string analysis creates a context-free language representing possible values for input

Strings and output Strings, derived from all modifications that are applied to the given string in the application. String objects are labelled to document their allocation site and all subsequent modifications. The labels map nodes in the call graph of the application to literals in the context-free language. After the string analysis, each character carries its own history of modifications from allocation to the site where they are used as arguments for allocation of a Permission object. Fig. 1 illustrates this on an example. Each character of the string is assigned a list of labels, where each label describes the operation or allocation site. String analysis increases precision over data flow analysis, as resource identifiers can be determined in cases where the work in [4] and [5] only generate generic permissions.

The third subset contains all those entry point methods that have non-string arguments containing String objects. These String objects are used as resource identifiers. For each of these non-string objects rules are predefined that describe how to obtain the resource identifier from the contained String object(s). If no rule is predefined for a specific non-string object, a generic permission that matches the Permission type used for the corresponding permission check is used, which permits access to any resource of that type. Once the resource identifiers have been extracted, the analysis continues with the one for the second subset.

Once the analyses are done for an entry point, the permissions needed when calling the corresponding method are determined. They are added to the corresponding node in the call graph. After all entry points are analysed, all the collected permissions are propagated backwards through the call graph to the start nodes. In nodes which are join-points of paths, permissions are combined with set unions. After the backward propagation, the start nodes contain all the permissions the application needs. The policy is created from these permissions.



```
…
String s = System.console().readLine();
…
s.toLowerCase();
…
Permission p = new
   FilePermission(s, "read");
```

Fig. 1. Example of labelling characters of a string in the string analysis

The drawback of call graph-based analysis (as introduced in [4,5]) is that the call graph overapproximates [13]: it contains paths which do not correspond to program flow of the code. This overapproximation – we call it *call graph-based* *overapproximation* – results in access rights in the policy which the application actually does not need. Therefore the analyses which are sketched here are introduced in [6] in order to eliminate call graph-based overapproximation.

A different sort of overapproximation – we call it *indecisiveness overapproximation* – remains in [6], however. With all the analyses, many access rights which the application needs can be obtained statically. The rest of the access rights can only be determined on execution, as the involved resource identifiers are only defined at runtime. Thus they are unknown at the time of static analysis. In [5], static analysis is combined with dynamic analysis, but this is to overcome call graph-based overapproximation. For permissions that can only be determined at runtime, generic permissions are added to the policy in [6] (as well as in [5]). These generic permissions are added for all user resources and for those system resources which are not specified in the source code of the application (e.g. the file separator character in Java). Some resource identifiers are defined by operations at runtime. These operations receive various arguments which also may partially be available at runtime only. In such a case, overapproximation is countered in [6]: the string analysis provides transforms for operations which modify input that is used as resource identifier. These transforms are used to determine permissions statically. Generic permissions are only added if for certain operations no transforms are defined.

The improvement of the algorithm presented in Section 3.3 further reduces indecisiveness overapproximation of this approach by avoiding generic permissions for user resources in the generated policy.

2.2 Runtime Observation

Cowan's AppArmor [14,8] and Provos' Systrace [7] are runtime observation approaches. System calls are recognised and recorded, based on the Linux Security Modules (LSM). An application is executed several times in learning mode. All the performed system calls are written to a policy. Finally, the policy is examined manually and applied. Any future execution of the application is controlled within the bounds of this policy.

The drawback of the approach is the manual policy examination. It is left to the user. In addition, a general drawback of runtime approaches is that completeness of execution coverage cannot be determined. Only if all the functionality of the application is executed, a complete policy is generated. As a consequence, generated policies are likely to be incomplete. This is closely related to the need of generic permissions for user resources. They are needed as it cannot be determined whether during execution all possible user resources have been accessed. Due to these generic permissions, runtime observations overapproximate.

However, the approach is an important work on auto-generation of policies. It has been chosen for creating the observation records in Section 3.4. In order to benefit from this approach, it is combined with the other contributions of this paper. This eliminates the involvement of the user for manual policy examination and it avoids generic permissions for user resources by filtering permission checks which are performed by the user.

2.3 Further Analysis Approaches

Wagner and Dean [15] present a combination of static analysis and runtime monitoring. Using static analysis, a model of the application is created which is represented by an automaton. This automaton models the order in which system calls are made by the application. Each system call of the application initiates a state transition of the automaton. Any system call which the application makes when executing normally transfers the automaton from a valid state to another. Any system call that is normally not made by the application leads to an error state. Using this automaton at runtime allows recognising illegal state transitions of the application, which is used to terminate the application. The advantage is that this approach takes the history of system calls into consideration. However, while this approach keeps track of the application's control flow, it does not provide fine grained access control for resources. Once a system call is permitted, the resources on which the system call operates are not further restricted. Thus, this approach should always be combined with other access control models.

Polymer [16] also follows a two step approach. The bytecode of Java applications and libraries is instrumented with jumps into the Policy object, which performs policy enforcement. The policy is a compiled Java object. At runtime, the instrumented code and the Policy object act as runtime monitors to perform access control on the level of method calls. Polymer does not support generating the policies. This is a purely manual task. It also provides limited dynamic policy adaptation at runtime.

2.4 Other Related Work

Systems based on the object-capability security model [17] provide the application with a reference after the user has chosen a resource. The application itself does not have access rights for the resource, but via the reference, the application can access the resource. Thus, the user transparently provides the necessary access rights which the application needs. These object-capability-based systems and the contribution of this paper have the same user perception in common. It is the user's responsibility to carefully choose resources the application should operate on.

Taint tracking [18,19,20,21] is used to filter potentially dangerous user input before it is used for sensitive operations. There are different approaches, but they all have in common that user input is tracked when propagating through an application. Some work recognises intrusions when user input is used as arguments for certain sensitive operations, as these operations are normally only executed with arguments that are not specified by the user. If user input reaches such sensitive operations, the application will be terminated. In other work, control characters are filtered from user input. For an SQL statement, for example, characters, such as semicolon and quote characters, would be filtered. This prevents the user from rewriting the SQL statement. In this paper, taint tracking is used for tracking user defined resource identifiers.

3 Policy Generation with User Interaction Recognition

When analysing an application in order to generate an access control policy for it, all the control flow that leads to resource access is of interest. Each access to a resource initiates a permission check. The permission check determines whether access is permitted or prohibited. Control flow analysis starts at the function that starts the application and it ends at methods which perform a permission check. If all these control flows of an application are captured along with the corresponding access rights, a complete description of the application's access behaviour is available which can be used to generate an access control policy. Existing work does that by static code analysis [4,5,6], as described in Section 2.1 and by runtime observations [5,8,7] (see Section 2.2). Static analysis models stack inspection-based access control [22] statically and runtime observations collect all the access attempts of an executed application when they occur. This paper uses a combination of static analysis and runtime observations in order to eliminate either one's limitations. Since static analysis covers the entire code of the application, the generated policy is complete, i.e. it contains at least all the access rights the application needs. Runtime observations are incomplete, but they can determine permissions which cannot be determined statically prior to execution.

As explained in Section 2 and as further elaborated upon in Section 6.2, existing work suffers from different types of overapproximation. In order to eliminate overapproximation, the different types of overapproximation are to be treated differently. This requires distinguishing them. This distinction can be achieved by recognising user input. Indecisiveness overapproximation can be eliminated for user resources if all user-initiated access is discarded from the generated policy. Therefore this section integrates user input recognition into the policy auto-generation process. How to eliminate indecisiveness overapproximation for system resources is addressed by Section 6.2.

Integrating user input recognition into the policy generation process allows determining if an access is initiated by the user. Two promising approaches of user input tracking were found by the authors: (1) using information tracking along with Aspect-Oriented Programming (AOP) [23] to track user input during execution (known as *taint tracking*; see Section 2) and (2) extending the call graph-based static analysis. This paper uses both these approaches in combination. The contribution of this section extends the static analysis approach explained in depth in Section 2.1 and it uses ideas from the runtime observation approach by Cowan [14,8] (see Section 2.2).

Extending policy generation by user input recognition leads to the process depicted in Fig. 2. All the steps of the process are integrated in the development phase of the application. Policy generation itself takes place in step 5, where the policy is generated from all the input which is collected in the steps 1, 3 and 4. These four steps (marked by slightly darker background colour in Fig. 2) are discussed in this section.

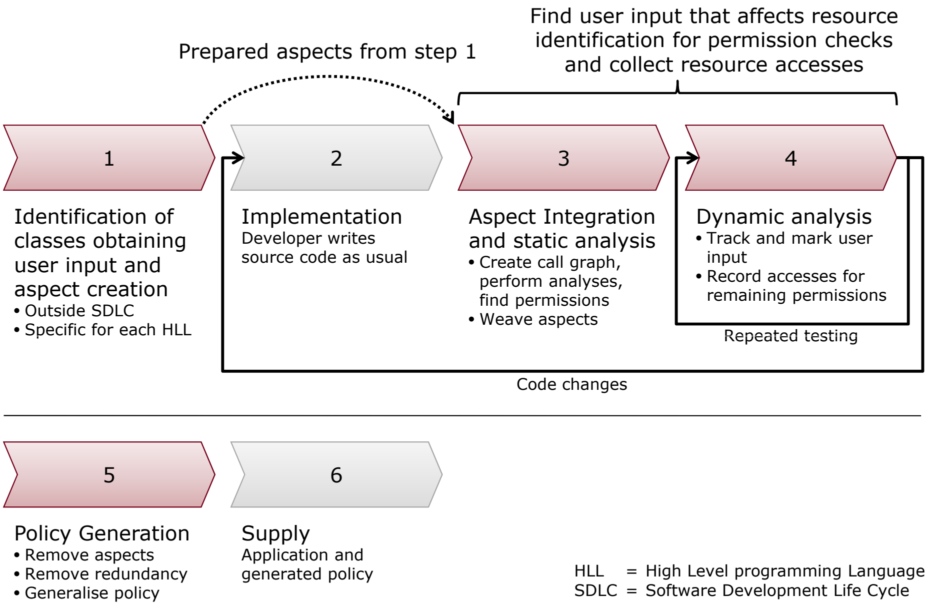

Find user input that affects resource identification for permission checks and collect resource accesses
Prepared aspects from step 1
1
2
3
4
Identification of classes obtaining user input and aspect creation
• Outside SDLC
• Specific for each HLL
Implementation
Developer writes source code as usual
Aspect Integration and static analysis
• Create call graph, perform analyses, find permissions
• Weave aspects
Dynamic analysis
• Track and mark user input
• Record accesses for remaining permissions
Repeated testing
Code changes
5
6
Policy Generation
• Remove aspects
• Remove redundancy
• Generalise policy
Supply
Application and generated policy
HLL = High Level programming Language
SDLC = Software Development Life Cycle

Fig. 2. The process of auto-generating the application policy including recognition of user interactions

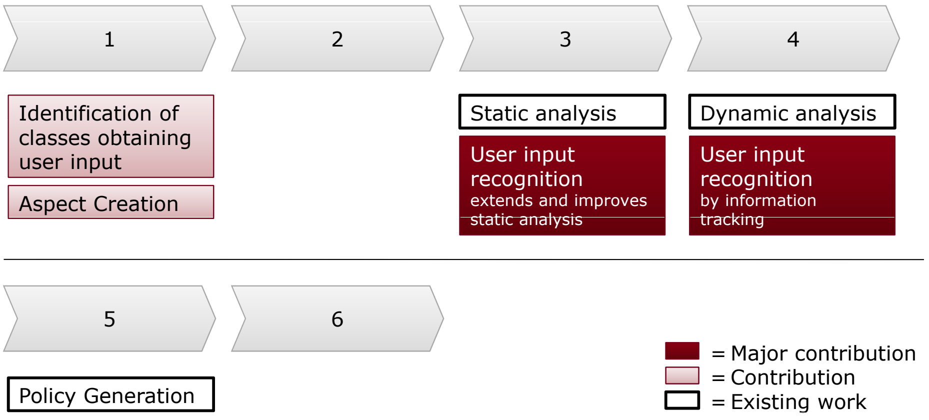

1
2
3
4
Identification of classes obtaining user input
Aspect Creation
Static analysis
User input recognition extends and improves static analysis
Dynamic analysis
User input recognition by information tracking
5
6
Policy Generation
= Major contribution
= Contribution
= Existing work

Fig. 3. Combination of existing work and contributions

Fig. 3 illustrates in which way existing work and contributions are combined in the steps of Fig. 2.

3.1 Classes Obtaining User Input

Tracking propagation of user input starts at those classes which obtain user input. These classes are collected in Step 1 of Fig. 2.

All the classes that come with a programming language and its execution environment – the so called *system library* – are well-known and finite in number and size. Thus, all the classes of the system library that obtain user input comprise a subset of the system library which is also finite. A list of all classes that obtain user input is compiled statically individually per programming language. The list is used as starting point for tracking user input through the application. If all entry points, such as console input, Graphical User Interface (GUI) input, network input, file input, database input and others, are taken into account, a complete list can be compiled. Some external libraries (e.g. windowing toolkits) are to be added separately if they are not part of the system library. As the object-oriented programming paradigm follows the idea of composing components, applications typically reuse components from the libraries to implement their functionality. Applications do rather not create own classes covering commonly used low level functionality, such as obtaining user input. Consequently, the list is not to be changed by the developer when implementing an application. It is only to be extended if an external library is used that does not provide its own list.

For some classes of the list, all instances obtain user input (e.g. a text box of a windowing toolkit). With other classes, only certain instances obtain user input (e.g. the input stream in Java that connects keyboard input to standard-in, but not necessarily any other input stream).

3.2 Aspects

In order to track propagation of user input through the application in the runtime observation approach, classes are augmented by aspects. This subsection discusses which classes are augmented and what the aspects do (Steps 1 and 3 in Fig. 2).

The aspects augment classes by a new field that stores *taint information.* When data is assigned to instance objects of these classes, the new field is set *tainted* if the data is obtained from user input. In any other case, the field is set *not tainted.* The aspects observe all operations that change the state of objects containing taint information. They update taint information accordingly. If data is propagated to other objects, taint information is also propagated by the aspects. Consequently, all the classes that are involved in user input propagation are augmented. The objects that perform permission checks, finally, receive the usual data they need for permission check, i.e. the resource identifier and the requested access right. They also receive taint information that is stored in the object containing the resource identifier. This allows aspects in the objects that perform permission checks to distinguish user-initiated access requests from application-initiated requests.

When classes deal with user input, apart from obtaining it, they can store, modify or transform this user input. Methods of the class perform this functionality. They are augmented by aspects to set taint information accordingly. A class that stores user input also needs to store taint information. A class that modifies user input also needs to modify taint information. A class that transforms user input into another type also needs to provide the target class with taint information. The target class needs to receive and store this taint information. Methods that perform other functionality that does not affect user input need no augmentation.

User input is data that is stored in a class either by calling a method of the class or by assigning the data directly to a field. In whatever way the state of the field is changed, taint information is to be set accordingly. If a method changes the state of the field, the method is augmented by the necessary aspect. If the field is accessed directly, the member class which represents the field is augmented.

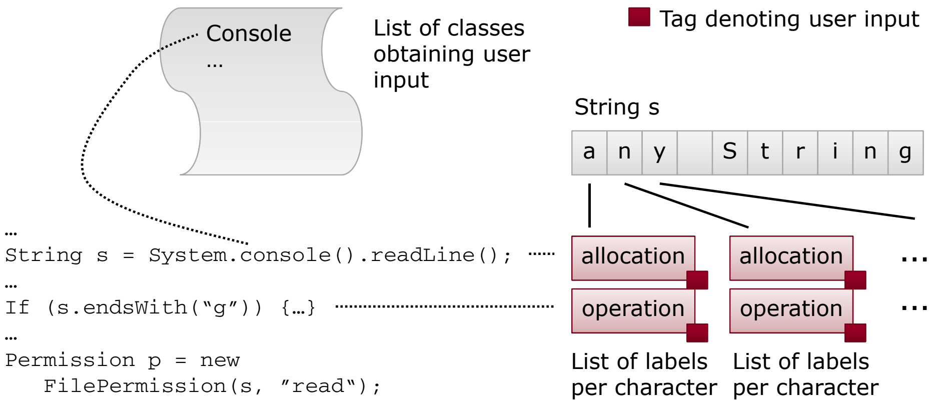
Aspects are prepared statically outside the process of policy auto-generation (Step 1 in Fig. 2). The prepared aspects are specific for each programming language. An example aspect for Java is listed in Section 4.2. The aspects are weaved into the application's code during development (Step 3 in Fig. 2). Many classes (mainly high level classes) are augmented with generic aspects, i.e. aspects with pointcut definitions which apply to a wide range of classes. Such a generic aspect applies, for instance, to all methods of all classes that return a value of type `String`. Some classes (mainly low level classes) are augmented with individual aspects to cover all their specific data propagation possibilities.

3.3 Call Graph and String Analysis

In order to distinguish user independent actions of the application from user interactions, values and allocation sites of resource identifiers are determined in the static analysis (Step 3 in Fig. 2). As soon as a resource identifier is allocated by a class from the list (see Section 3.1), the resource identifier is known to be defined by the user. The string analysis (see Section 2.1) is extended by integrating this distinction. The labels are analysed to find all the characters of which the resource identifier is composed and their allocation sites. Each of the allocation sites is looked up in the list of user input obtaining classes. If the allocation site is listed, the allocation label is tagged as originating from user input. All the other labels are analysed for their string operations. If the operations keep the original content obtained from the allocation site, their corresponding labels are also tagged. If the content is changed, e.g. by using a substring, the user input tag is discarded. This is to prevent the application from composing a resource identifier from parts of user input to gain access to arbitrary resources at runtime. The transforms (see Section 2.1) are extended by describing whether user input tags shall be dropped when the corresponding methods are applied.

If all the labels of a character get the user input tag, the character itself is tagged as user input. The string representing the resource identifier can either consist of (1) only characters that are tagged as user input, (2) characters that are tagged as user input and characters that are not tagged as user input and (3) no character that is tagged as user input. Treatment of those strings that only consist of tagged or untagged characters is easy: the strings are tagged according to their character tags. In the mixed case, the string is tagged as user input. In all potentially dangerous cases, the transforms of string operations removed the user input tag before. Thus, it is safe to treat the mixed case as user input. If the string is tagged, the permission object which uses the string as resource identifier is also tagged. The tagged permission object indicates that the permission it represents is defined by user input.

Fig. 4 depicts the extended analysis on an example: the allocated string originates from user input and the endsWith operation does not remove this property; thus, the characters used in the permission check originate from user input.


Console
...
List of classes obtaining user input
🟥 Tag denoting user input
String s
a n y S t r i n g
allocation
allocation
...
operation
operation
...
List of labels per character
List of labels per character

```
…
String s = System.console().readLine();
…
If (s.endsWith("g")) {…}
…
Permission p = new
   FilePermission(s, "read");
```

Fig. 4. Example of library-client application analysis extended by tagging labels of those characters of a string that originate from user input

The application analysis of Section 2.1 is extended by adding this user input analysis. Since subset 1 of the partitioned set of entry points is independent of user input, no extensions are applied. Both subset 2 and subset 3 entry points require extensions. After all the labelled strings are available, they are used for user input analysis. The extended algorithm collects permissions as before, but it tags those permissions that contain resource identifiers obtained from user input. This makes them distinguishable from the others which have resource identifiers not obtained from user input.

There is no need to extend the library analysis, as the application analysis considers all input that comes from outside the library when the library is used by an application.

The approach presented here further reduces overapproximation of the call graph-based approach discussed in Section 2.1. All the resource identifiers that are defined by user input, which cannot be determined prior to runtime, are not added to the application policy at all. Thus, there is no need for generic permissions as a result of the static analysis. Cases in which generic permissions are still required are limited to system values of the runtime (e.g. the file separator character in Java) and to resource identifiers that are read from other sources (e.g. database or file). Therefore overapproximation is reduced, but not eliminated. These cases of remaining overapproximation can be treated by defining sets of possible values to further reduce overapproximation, as discussed in Section 6.2.

3.4 Dynamic Analysis

During dynamic analysis (Step 4 in Fig. 2), the augmented classes are capable of tracking user input through the application. The application is executed repeatedly for software testing. During these executions, all access requests are recorded and stored during permission checks to generate the application policy from these records [5,8] (see Section 2). The records are stored together with the corresponding taint information for each resource identifier. If a resource identifier originates from user input, the corresponding record is discarded. For processing tainted data, content and semantics of the data is irrelevant. Only the propagation of data together with its taint information is relevant.

3.5 Policy Generation

The policy is generated in Step 5 of Fig. 2. After static analysis collected permissions and after execution has been finished, the access control policy is generated from all the acquired permissions.

For policy generation, redundant permissions are removed, as they are useless. Permissions with specific access rights that are implied by more generic permissions are removed as well. Finally, the policy is generated from the remaining records.

Since aspects are only needed for policy generation, they are removed in the policy generation step. The deployed application does not differ from an application for which no policy was generated.

4 Prototype Implementation

The prototype of the runtime observation approach with taint tracking [24] has been implemented in Java [25] and AspectJ [26]. Java's modular design allows replacing components easily. The access control model of Java [11] is flexible and advanced. Fine-grained access control is possible. We extend Java's access control facilities by replacing the default Policy Enforcement Point (PEP), i.e. the `SecurityManager` class. Access control is performed on the level of the Java Virtual Machine (JVM), based on the various subclasses of the `Permission` class.

The list of user input obtaining classes contains core classes like, for example, java.io.Console and java.io.InputStream. Direct user input is obtained from standard-in using these classes. To obtain user input from arguments given when starting the application, any class containing a main method is added to the list. Among the windowing toolkits only Swing is exemplified here. Text input fields (e.g. javax.swing.TextComponent) and dialogue boxes (e.g. javax.swing.JFileChooser) are added to the list. The classes Byte, Character, Integer and String are also added, as the InputStream uses the first three to store the input it gets from its data source. These classes are also involved in other String operations, processing user input until it is stored in a String representation.[1] In order to analyse tracked taint information, the SecurityManager.checkPermission method is added as well.

Since Java does not only know objects, but also support primitive types when adding aspects to classes, there are two categories of classes to distinguish. Category 1 represents all the low level classes that correspond to primitive types: they store data in their fields in primitive types. Category 1 classes access their fields directly. These classes are the end of the hierarchy of member classes. Category 2 comprises all the classes that store their data in fields that are classes by themselves. Category 2 classes need to call methods of their member classes whenever they store or read data therein. There can be an arbitrary hierarchy of member classes of category 2. All classes which are not category 1 are category 2.

Category 1 classes need to be augmented in any case. Each time, data is stored in these classes, taint information is to be set according to the origin of that data. If, for instance, data is obtained from the console (i.e. from System.in), it is known that this instance of InputStream always produces user input. The array of int in which the system library class InputStream stores the user input, needs to set its data tainted.

Category 2 classes do not necessarily need to be augmented. Their member classes refer to other objects which already may contain taint information. However, if a category 2 class is capable of transforming its content to another type, the corresponding methods need to set taint information of the target type according to the source type. Therefore such a category 2 class needs to be augmented.

Native methods are augmented using around advice. Taint is tracked on return values. When the method is called, its arguments are analysed and when it returns, its return value can be set tainted. This requires understanding the semantics of the method to some extent.

4.1 Implementation Details

We implemented a plug-in for the integrated development environment *eclipse* [27]. This plug-in accompanies the software development and testing process by augmenting classes with aspects and by auto-generating the policy. The plug-in

[1] Java actually uses primitive types in InputStream and other low level classes.

contains the aspects. The developer does not need to write their own aspects, unless the developer writes code that is capable of obtaining user input directly without using existing Java classes.

The application is always executed using the replacement SecurityManager – called ObservingSecurityManager – and a replacement policy provider – called PolicyObserver. The PolicyObserver always returns false for each permission check without consulting any policy. This causes the AccessControlContext, which is involved in policy enforcement, to throw a security exception [11]. This exception is caught by the ObservingSecurityManager and forwarded to a monitor class, which analyses and stores its contents. That way, the subject, the subclass of the Permission class, the resource identifier, the action, the entire call stack and the code base are obtained by the monitor class. Due to the aspect by which the ObservingSecurityManager is augmented, the monitor class also obtains taint information and stores it in the records.

The ObservingSecurityManager suppresses the caught exception and returns silently. Consequently, the application gets all access attempts permitted. This allows testing any application feature without being hindered by security constraints. As this takes place in the development phase, there is no threat for the system where the application is deployed.

The policy is generated from all the records. Filtering of user-initiated access and removal of redundancy is done as described in Section 3.

In order to keep the ObservingSecurityManager small and independent of analysing the exception, the ObservingSecurityManager sends the exception it caught on a permission check to a server process using RMI. This server process generates the policy.

4.2 Aspects

As described in Section 3.2, there are generic and specific aspects. For the prototype, it is advisable to classify them in three groups: group 1 consists of aspects that are needed to store, read and transfer taint information in classes. They provide methods to set and get taint information and they add a taint bit that stores taint information. All classes that obtain or process user input need to be augmented by these aspects. Classes are augmented by *inter-type declaration*, i.e. classes inherit from both the aspect and the Object class (or a sub-class). Augmented classes can then store and change their own taint information. Pointcuts of Group 1 aspects define which classes are to be augmented by that functionality. Aspects in group 2 are generic aspects that specify in which cases group 1 aspects' functionality is to be called in order to update the taint bit. This is the case when data in Group 1 classes is set or modified. These aspects are generic, as they apply to multiple classes satisfying some common properties. Group 3 aspects are all the specific aspects that deal with peculiarities of certain library classes. They have the same purpose as Group 2 aspects, but they are specifically written to track the taint bit in a particular class. Listing 1 shows one of the Group 2 aspects.

Listing 1. Advice augmenting main method

```
before(String[] args): execution(public static void *.main(String[]
    || String...) && args(arg) {
  for (String string : arg) ((Taint)string).setTainted(true);
}
```

5 Example

In the following, the prototype is used to generate the policy for the *UMU XACML-Editor* (Version 1.3) exemplarily. The results are compared to the state-of-the-art and evaluated. The UMU XACML-Editor [28] is a GUI-based XACML file editor written using Swing. At first, all the resource access has been collected using the ObservingSecurityManager. Table 1 lists all the access attempts which occur when the application is executed. Access attempts with numbers 3, 4 and 9 in the table are initiated by the user. When applying existing work, the same access attempts are collected, as the ObservingSecurityManager performs the same analysis. However, creating a policy from the collected access attempts is of little avail. It permits the application to access all the system resources it needs to execute normally, but it only permits the application to access those user resources (i.e. XACML files in the example) the user has chosen when the access attempts were recorded. Therefore existing work involves the user to manually inspect the policy. Thereby, the user ought to add a generic permission for file access which permits the application to access all the XACML files which may be opened in the XACML-Editor in the future. In order to prevent adding this generic permission, our prototype identifies user interactions. In the case of the UMU XACML-Editor, all user interactions are initiated via file dialogues. They are all identified and marked in the records. They are discarded for policy generation. Consequently, the generated policy does not contain access rights with numbers 3, 4 and 9 from Table 1, but all the other access rights. In contrast to existing work, the resulting policy does not overapproximate. Therefore it is the least privileges policy for the application. A modified file dialogue can then augment the policy at runtime upon user interaction (as described in Section 7.2).

6 Evaluation

6.1 Prototype

The AOP aspects of the prototype instrument the system library. They identify user-initiated resource access correctly. As depicted by Table 2, the overall number of necessary aspects is kept in a manageable range. This is due to the generic aspects which affect a large number of classes. For production, external libraries need to be augmented as well. From the feedback of the prototype implementation, this is a scaling task with respect to the size of the external libraries.

Table 1. All access attempts of the UMU XACML-Editor. Duplicates are omitted. Access attempts with numbers 3, 4 and 9 (bold font) are initiated by the user.

No.	Permission	Resource	Action
1	java.awt.AWTPermission	accessEventQueue	
2	java.awt.AWTPermission	showWindowWithoutWarningBanner	
3	java.io.FilePermission	/user/policy	read
4	java.io.FilePermission	/user/policy.xml	write
5	java.io.FilePermission	/UMU-XACML-Editor/bin/icons/cara.gif	read
6	java.io.FilePermission	/UMU-XACML-Editor/bin/icons/nube.gif	read
7	java.io.FilePermission	/UMU-XACML-Editor/bin/icons/target.gif	read
8	java.io.FilePermission	/UMU-XACML-Editor/bin/icons/verde.gif	read
9	java.io.FilePermission	/user	write
10	java.lang.RuntimePermission	exitVM	
11	java.lang.RuntimePermission	modifyThreadGroup	
12	java.util.PropertyPermission	elementAttributeLimit	read
13	java.util.PropertyPermission	entityExpansionLimit	read
14	java.util.PropertyPermission	maxOccurLimit	read
15	java.util.PropertyPermission	os.name	read
16	java.util.PropertyPermission	os.version	read
17	java.util.PropertyPermission	user.dir	read

Table 2. Number of aspects in the prototype implementation. The figures are limited to the packages java.lang, java.io, java.net and javax.swing. Aspects needed for compensating primitive types are not considered. For inter-type declarations, the number of affected classes is limited to directly affected classes. Through inheritance more classes become affected.

Group		Advice	Named Pointcuts	Affected Classes
Inter-type declarations	1	1	0	11
Generic aspects	2	8	8	41
Specific aspects	3	28	36	26

6.2 Elimination of Overapproximation

Existing work on policy generation suffers from limitations, as discussed earlier. Runtime observations are incomplete (see Section 2.2) and static analysis suffers from indecisiveness overapproximation (see Section 2.1).

Combining observations and static analysis (as done by Centonze et al. [5], see Section 2.1) counters the drawback of incompleteness of runtime observations. The combination can further reduce overapproximation of static analysis, as generic permissions can be more precise and minimised to a set of valid values in some cases, but they still remain.

The contributions of this paper further reduce overapproximation. By discarding user-initiated resource access, all access to user resources is omitted from the policy. There is no generic permission in the policy and the content of the policy does also not depend on the user resources which the tester has chosen during runtime observations. Access to user resources is the major cause for overapproximation in existing work. Thus, the primary source of overapproximation is eliminated.

There are still special cases where overapproximation remains: in cases where no transform is defined for an operation which is analysed by the string analysis, as well as in cases where no set of possible values for a resource identifier which is set by the execution environment is defined. Both these cases can be countered by ensuring that all the transforms and sets are defined. The policy generation process can be implemented in a way that it identifies missing transforms and sets. The sets can then be defined directly in the policy generation process, for instance by adding annotations to the code. Consequently, overapproximation is eliminated which results in policies that are complete and that also represent the least privileges of the corresponding application. This is a major benefit over the state-of-the-art. Table 3 summarises the differences and the gains by comparing the state-of-the-art, the individual contributions and the combined contributions of this paper.

Table 3. Comparison of state-of-the-art and contributions

	Static analysis [6] (see Section 2.1)	Dynamic analysis [7] (see Section 2.2)	Static analysis extended by contributions from Section 3.3	Dynamic analysis extensions by contributions from Sections 3.2 and 3.4	Combination of all contributions of Section 3
Over-approximation	Overapproximates on permissions only known on execution (indecisiveness overapprox.)	Overapproximates on user resources	Reduced indecisiveness overapproximation to system resources only	Does not overapproximate as no user resources are captured	Does not overapproximate
Completeness	Complete	Incomplete	Complete	Incomplete	Complete
Scalability	Scales due to separated library analysis	Scales due to the ability of combining permissions in include files	Scales due to separated library analysis	Scales as the number of aspects to be created is in a manageable range	Scales as the individual solutions combined here scale and as the combination does not add non-linear complexity
Automation	Automated to large extent	Manual inspection of generated policies required	Automated with a few exceptions	Aspects are created manually per programming language; Policy generation is fully automatic	Aspects are created manually per programming language; Policy generation is fully automatic
Requirements	Source code or object code	Complete test coverage	Source code	Complete test coverage	Source code

7 Discussion

7.1 Threat Model

The policy generation process contributed in this paper is meant for protecting systems against applications that misbehave due to programming errors and due to being exploitable by attackers. Since the policy is generated by the developer, a developer of a malicious application can generate a policy that permits all the malicious access. The contribution does not protect against malicious applications.

7.2 The Big Picture

If no further measures are taken in the deployment phase and/or in the execution phase, the generated policy is of little avail. If the application is deployed together with its application policy, manifold measures are advisable, as discussed next.

For deployment, the application policy can be checked against the policy of the system on which the application is to be deployed, to see if they do not contradict. This can be done manually by examining the policy or automatically. The European research project *Security of Software and Services for Mobile Systems* (*S3MS*) [29,30,31] provides means to prove that the application policy matches the application and that the application policy does not contradict the system policy.

As mentioned in Section 1, it is not sufficient to only apply the application policy at runtime. The policy needs to be adapted.

7.3 Future Work

The work presented here is limited to *volatile user input*. This is input that is only relevant for the currently executed instance of an application. It is used, for example, to open a file the user intends to edit in the application. We will also address *persistent user input* in our future work, which persists over multiple executions of the application. This is the case, for instance, if the user specifies the path and name of a configuration file which is read each time the application is started. Once specified by the user, the application should have access rights for future executions.

Means are needed to handle new classes that are capable of obtaining user input by themselves without relying on classes from the system library. In such a rare case, aspects are to be auto-generated from aspect templates. This way, the developer is not required to write aspects in order to apply user input tracking to these new classes.

The approach presented here relies on the availability of source code of the application. Applicability on intermediate language code (Java bytecode or .NET CIL) is to be elaborated.

In some cases, additional information is required for generating the policy. In these cases the developer needs to specify meta information. Investigations on integrating this meta information specification into the development process with little developer involvement are required.

8 Conclusion

This paper presents means to auto-generate least privileges access control policies for applications. While existing work is used for the process of retrieving the contents for the policy by static and dynamic analyses, this paper introduces a way to distinguish resource access performed by the application from resource access initiated by the user. This distinction allows generating the application policy that satisfies the principle of least privileges. The application policy does not contain any access right to user resources, whereas existing work permits generic access to user resources. Access rights users need to access resources in the application are later added on the target system.

Two approaches are presented here. The static analysis approach uses a call graph of the application and performs various analyses to determine and tag resource identifiers that are defined by the user. The policy is generated without adding permissions that are based on tagged resource identifiers. The runtime observation approach tracks user input through the application using taint tracking and aspect-oriented programming. If user input propagates to a permission check where the resource identifier is specified by the user, the corresponding access is treated as user-initiated. A prototype is implemented in Java. Its implementation shows that the total number of aspects is kept in a manageable range. It suffices to augment those classes by aspects which play a key role in processing user input. As a result, the approach is feasible, it scales with respect to the size of instrumented libraries and it reduces overapproximation of existing approaches significantly. However it requires a fully object-oriented programming language, as AOP cannot be applied on primitive data types.

If both approaches are combined, a complete and sound policy is generated and overapproximation is eliminated. As the policy is auto-generated, the effort for the developer is low. The resulting application policy can be used on the target system to execute the application in its bounds. The target system only needs to specify access rights for user resources. Thus, the effort is also low there. As a consequence, policy generation becomes practical. An outlook of three obvious possibilities to apply the contributions in practice concludes the paper:

The mobile phone is an appealing target, as the mobile industry controls most of the phases of the SDLC. Development environments can be extended by the analyses, the generated policy can be included in the supply chain of applications and the execution environments on the mobile phones can be adapted to perform policy adaptation. Controlling access on a mobile phone to user resources, such as the phone book or the agenda, means controlling access to personal data which not all the applications need. Effective tailored access control on a per-application basis is possible and practical that way.

Execution environments, such as the Java Virtual Machine and the .NET CLR, provide a fine-grained and flexible security architecture that allows enforcing tailored access control policies. The problem in practice, however, is that it is complex to write proper policies. Integrating the policy auto-generation process into the SDLC reduces this effort to a minimum.

The two execution environments .NET CLR and Android allow for defining the access rights an application needs in a configuration file which is supplied together with the application. This is used at install time in order to assign the right access rights. However, there is no support in collecting all the needed access rights. The policy auto-generation process can be used to close this gap by filling the section of required permissions in the configuration file.

References

1. Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
2. McGraw, G.: Software Security - Building Security. Addison-Wesley, USA (2006)
3. National Institute of Standards and Technology: National vulnerability database statistics, `http://nvd.nist.gov/statistics.cfm` (last checked: August 2010)
4. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for java. In: OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 359–372. ACM, New York (2002)
5. Centonze, P., Flynn, R., Pistoia, M.: Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies. In: Proceedings of the 23rd Annual Computer Security Applications Conference, ACSAC 2007, pp. 292–303 (December 2007)
6. Geay, E., Pistoia, M., Tateishi, T., Ryder, B.G., Dolby, J.: Modular String-Sensitive Permission Analysis with Demand-Driven Precision. In: Proceedings of the 31st International Conference on Software Engineering, pp. 177–187. IEEE, Los Alamitos (May 2009)
7. Provos, N.: Improving host security with system call policies. In: SSYM 2003: Proceedings of the 12th conference on USENIX Security Symposium, Berkeley, CA, USA, pp. 18–18. USENIX Association (2003)
8. Novell, Inc.: AppArmor, `http://en.opensuse.org/AppArmor/` (last checked: August 2010)
9. Goldberg, A., Kay, A.: Smalltalk-72 Instruction Manual. Technical Report SSL 76-6, Learning Research Group, Xerox Palo Alto Research Center, California, USA (1976)
10. Eckel, B.: Thinking in Java, 3rd edn. Prentice Hall, Nwe Jersey (2003)
11. Gong, L., Ellison, G., Dagenforde, M.: Inside Java 2 Platform Security, 2nd edn. Addison-Wesley, Reading (2003)
12. Horwitz, S., Reps, T., Binkley, D.: Interprocedural Slicing Using Dependence Graphs. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 35–46. ACM, New York (1988)
13. Shivers, O.: Control flow analysis in scheme. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 164–174 (1988)
14. Cowan, C., Wright, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux security modules: General security support for the linux kernel. In: Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA (August 2002)
15. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 22nd IEEE Symposium on Security and Privacy, pp. 156–169 (May 2001)
16. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), Chicago, IL, USA, pp. 305–314 (2005)

17. Miller, M.S.: Robust Composition - Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, MD, USA (May 2006)
18. Xu, W., Bhatkar, E., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: 15th USENIX Security Symposium, pp. 121–136 (2006)
19. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: 20th IFIP International Information Security Conference (SEC), pp. 372–382 (2005)
20. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21 (2003)
21. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
22. Wallach, D.S., Felten, E.W.: Understanding java stack inspection. In: Proceedings of the 1998 IEEE Symposium on Security and Privacy, pp. 52–63 (1998)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
24. Hengst, G.: Auto-generation of access-control policies - elaboration of an information tracking approach and its prototype implementation. Bachelor's thesis, Munich University of Applied Sciences (September 2009)
25. Sun Microsystems Inc.: Java Technology, `http://java.sun.com/` (last checked: August 2010)
26. Eclipse Foundation: Aspectj, `http://www.eclipse.org/aspectj/` (last checked: August 2010)
27. Eclipse Foundation: eclipse, `http://www.eclipse.org` (last checked: August 2010)
28. Dólera Tormo, G., Martinez Perez, G.: UMU XACML-Editor, `http://sourceforge.net/projects/umu-xacmleditor/` (last checked: August 2010)
29. S3MS project consortium: Security of Software and Services for Mobile Systems (S3MS), European research project, `http://www.s3ms.org/` (last checked: August 2010)
30. Dragoni, N., Martinelli, F., Massacci, F., Mori, P., Schaefer, C., Walter, T., Vetillard, E.: Security-by-Contract (SxC) for Software and Services of Mobile Systems. In: Nitto, E.D., Sassen, A.M., Traverso, P., Zwegers, A. (eds.) At Your Service-Oriented Computing From an EU Perspective, pp. 429–455. MIT Press, Cambridge (2009)
31. Aktug, I., Naliuka, K.: ConSpec - a formal language for policy specification. In: First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007), Dresden, Germany (September 27, 2007)