Challenges and design considerations for the authorization analysis of a software framework

Wook Shin Shinsaku Kiyomoto Kazuhide Fukushima Toshiaki Tanaka

KDDI R&D Laboratories, Inc.
2-1-15 Ohara Fujimino-shi Saitama 356-8502, Japan
{wookshin, kiyomoto, ka-fukushima, toshi}@kddilabs.jp

Abstract
We suggest an automated analysis tool design, where we can analyze the permission-based security of a software framework and test if an application execution can be completed with given permissions. For the analysis, we take two approaches. The one is theorem proving-based approach that formally specifies permission manipulation operations of the framework, defines security of the system, and then tests the logical correctness of the operations. The other is a simple simulation that partially executes a given application codes with given authorization.

1 Introduction

Software frameworks (such as Java or .NET-based mobile code systems and many of the current handset operating systems) are developed for providing high extensibility and portability in heterogeneous computing environments. They also publish application programming interfaces (API) so that users can easily benefit a variety of third party developed application software.
The typical security problem that those frameworks face is protecting local system resources from third party developed applications which inflow out of the security domain. Those applications are probable to violate security constraints of the local system.
The permission-based security is one of popular security countermeasure against the problem, which has been widely adopted for those frameworks, especially for Java language-based systems such as Java applets, J2ME MIDP, and Android. The permission-based scheme only allows an application to use system resource when the application has appropriate permissions. The scheme is very intuitive and easily understandable, but not easy to reason about security-related properties of the system.
We can think of two analysis requirements relating to the permission-based security enforcement in a software framework; Firstly, we need to confirm if the framework properly manages permissions while it behaves. If the framework fails to manipulate the authorization information properly, the security of the system cannot be guaranteed. Secondly, we also need to test if an application can be executed with a given authorization information. This could give a guideline to the security/usability question of asking how much permissions need to be granted and asked.
This paper proposes a design of an analysis tool that reasons about security of a framework and also notifies us when an application cannot be executed with a given authorization. The design takes two analysis approaches; the one is formally specifying the notion of security and the permission-manipulation operations of the system, and then verifying the logical correctness of the framework behavior. The other is partially executing an application and testing the application can be finished.
The rest of the paper is structured as follows; in the following Sect. 2 and Sect. 3, we briefly explain the ideas of our analysis approaches and suggest simple algorithms to accomplish the analyses. We also mention technical challenges and limitations of our design in Sect. 4, and conclude the paper in Sect. 4.

2 Background

This section briefly introduce our two analysis approaches: **framework analysis** and **application testing**, and introduce the Android as example target of our tool design.

2.1 Framework analysis

The framework analysis tries to prove the framework behavior correctly manipulates authorization information without any logical inconsistency. Since the proof will be verified with a formalism which is based on type system, we required to have formal specification of elements, behaviors, and security requirements of the framework.

For the analysis, we first describe types of elementary structure of the system where the framework is running. For example, the set of application identifiers can be represented as Set, the mathematical structure, to which the law of excluded middle is applied. It is possible to represent more complicated structure combining elementary types. Those types are used to define a state of a system.

Operations of the framework are described in the Hoare logic triples[1]. A Hoare triple is written as $\{\Phi\}\ P\ \{\Psi\}$, where Φ and Ψ are assertions that specify that *if the program statement P is executed at a state where Φ is satisfied, then it results in a state where Ψ is satisfied.* The Hoare logic also provides proof rules, called composition rules, which can be used to prove validity of a triple. Therefore, after we inspect framework source code and get pre-/post-conditions of operations, we can show validity of the framework behaviors by deriving $\{\Psi\}$ from $\{\Phi\}$ with program statement P using the rules.

We also identify security conditions of the permission-based scheme, and use the conditions to define a secure state combining them with the state definition.

The proof of framework behavior bases on the above logical expressions. We apply inference rules and see if each specified operation can logically derive a secure state from another secure state. If this is successful for an operation, then the operation preserves security when it occurs in the system. Consequently, if we verify the secure transition for all operations, then we can say the framework securely manipulates authorization information for an arbitrary system state. This guarantees that our target system stays secure when it starts its operation from a secure state.

Note that, however, we do not specify and verify every operation of the framework. Generally, there are too many operations defined in a framework for us to handle. Moreover, we rather try to concern security-related operations. We only focus on the following operations that affect the authorization status of the system: install, uninstall, start, and stop an application. Let us call those operations *permission-manipulation* operations. For more detail of the idea behind the framework analysis is explained in our previous work[2].

2.2 Application testing

The application testing checks the execution feasibility of an application with a set of granted permissions. Applications running on the software framework invoke API calls and benefit software services and hardware resources provided by underlying system. If a call leads to protected resources, the framework issues a permission checking process and check whether the caller has appropriate permissions to access the corresponding system resource. Let us name the functions that issue the permission check as *permission-consuming* operations. The result of the permission check depends on the given authorization at the moment, and the authorization changes as the permission-manipulation is executed in the framework.

For the testing procedure we propose, we need to figure out which permissions are necessary to invoke a particular function. That information can be obtained by looking at the framework source code. In case of a Java-based framework which conducts the permission check using the `SecurityManager` class or its child, we can look for where the `checkPermission()` function is called in the framework source and which permission is bound to the check routine. If an API call reaches to a function where the `checkPermission()` is invoked and `permA` is checked, we can conclude the API invocation requires the `permA`.

The application testing process needs the following two inputs: application code and authorization information given to the system. The authorization information should contain sets of permissions granted to applications. Based on this configuration information, the testing process simply emulates the application execution as follows: Firstly, we read through the application code from beginning and pick up permission-consuming operations and permission-manipulation operations, and make an ordered list of those operations. This is an abstracted form of the given application. In this procedure, we disregard the control structure of the application code. Secondly, we read the ab-

Figure 1: Overall structure

stracted code from the beginning again and do the following checks for each listed operation: if the operation is a permission-consuming operation, then check if the required permissions are granted to the application. If the operation is a permission-manipulation operation, then we apply the pre-/post-conditions of the manipulation operation and reflect the result to the current authorization information. If the operation is the consuming operation and also the manipulation, check the required permissions first.

2.3 An example framework

The Android framework is an example of a software framework that enforces permission-based security. In order to exemplify our idea more specifically, we take Android as our example target. The following sections will be explained in the context of the Android system.

In the Android system, a user can easily download and install an application from a remote site. At the installation time, the user is notified that the application requests a particular set of permissions for its successful run. If user decides to grant the requested permissions, then the granted permissions are stored in the framework. This can be obtained by `PackageSetting.grantedPermission()`. The granted permissions will be removed when the corresponding application is uninstalled from the system. Contrary to the J2ME MIDP (which is a popular Java-based framework for mobile devices), Android does not allow give/revoke permissions to/from an application during the application's runtime.

3 A Sketch of the Analysis Tool

A security analysis tool can be composed of the following sub-parts: **input, translation, specification, analysis, and output** (Fig. 1).

The **input part** includes: *the framework source code, security policy, application package, and authorization information.* The first two inputs are essential information for the framework analysis. The application analysis needs the last two inputs as well.

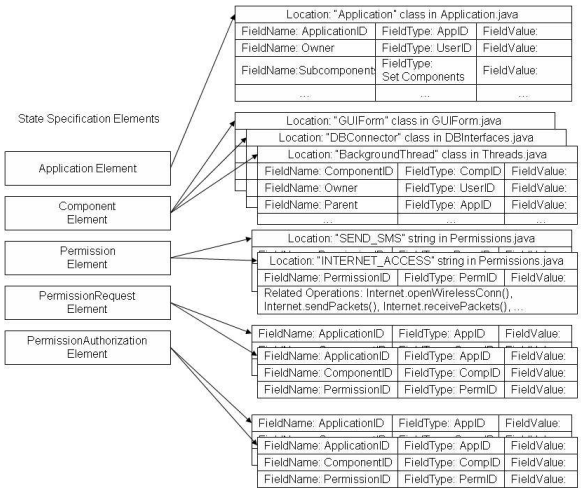

Figure 2: State Specification Elements

The **translation part** investigates inputs and generates specification elements. It includes several translators: *data structure translator, operation translator, policy translator, and application translator.* The *data structure translator* reads the framework source code and identifies abstracted data types. The *operation translator* generates pre-/post-condition expressions of permission-manipulation functions. It also identifies the necessary permissions to invoke framework functions. The *policy translator* converts security policy description to a set of security conditions that are written in predicate logic formula. The *application translator* reads application package which consists of application source codes and a package descriptor file. The package descriptor file contains a set of permissions that the application requests. Note that the data structures to represent an application are generated by the data structure translator. The application translator instantiates the data structures with the given application package information. The application translator also reads the authorization information and records the set of granted permissions for each application (Fig. 3).

The **specification part** includes data structures to represent the framework and application instances. The specification part is composed of *state specification elements, transition specification elements, policy specification elements, application specification elements.*

The *state specification elements* have the following sub-elements, which are essential to rep-

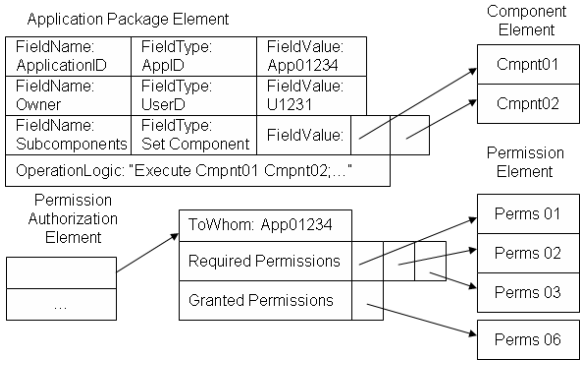

Figure 3: Application Specification Elements

resent authorization of the Android permission scheme: application element, component element, permission element, permission request element, and permission authorization element (Fig. 2). The application element is a composition of data types for depicting the structure of an application, which includes application ID, a list of sub-components, and permission request information. A component is a sub-element of an application, which includes component ID and parent application. A permission element has its own ID and metadata that includes the list of relevant functions. The metadata of a permission lists a set of functions that require the permission. A permission authorization is a list of tuples: (application ID, permission ID). It stores sets of granted permissions to applications.

The *transition specification elements* consist of install operation, uninstall operation, start operation, and stop operation. In Android, install and uninstall correspond to permission grant and revocation, respectively, so we do not specify grant/revocation operation separately.

The *policy specification elements* include a security condition element, and the given permission-based policy to our tool is a composition of multiple security conditions. We assume the conditions are connected by conjunction.

The *application specification element* has an application package element which specifies data structures of an installed application installed in the framework.

The **analyze part** consists of two processes: the framework analysis and the application analysis, which we describe in the next subsections in more detail. The **output part** displays results of the analysis.

3.1 Framework analysis

The steps of framework analysis we suggest execute the followings; (1) Let us assume the framework is in a state S, which is defined by the following state specification elements: the foreground component, installed applications, and granted permissions. (2) Assume that S is a secure state where all security conditions are satisfied. That is, when we have n number of security conditions, SC_i $(1 \leq i \leq \mathrm{n})$, each SC_i are satisfied at S. (3) When one of the transition specification elements is applied to the state, it results in another state S. A transition operation is described in terms of pre-/post-conditions. Therefore, we can add constraints to S by applying the precondition of the transition. Similarly, the resulting state S' can be obtained by applying the postcondition to the definition of a state. (4) Assume that S is also a secure state, which means every SC_i holds in terms of the foreground component, installed applications, and granted permissions of S'. (5) If we can logically derive S' from S using the inference rules, composition rules, and the well-known mathematical axioms and lemmas we added and proved, then we can prove that the operation can be applied to an arbitrary state of the system, and results in another state preserving security.

What the below steps try to do is the derivation process. Let us assume that we have n number of security conditions, SC_i $(1 \leq i \leq n)$ and m number of operations, Op_i $(1 \leq j \leq m)$.

[Step 1] Assume that we have a state S composed of *state_fgcmpnnt*, *state_package_tbl*, and *state_perm_table*. The state is secure, so we can denote every SC_i (*state_fgcmpnnt*, *state_package_tbl*, and *state_perm_table*) = **true**. Set j to 1.

[Step 2] if $j > m$, report to the **output part** that all operations can occur preserving the security, and terminate the test.

[Step 3] Assume that we have another secure state S where SC_i (*state_fgcmpnnt'*, *state_package_tbl*, and *state_perm_table*) = **true**. For an Op_j, try to logically derive S from S applying the list of inference rules and lemmas we have proven. If it can be driven, put the current proven rules to our lemma list, increase j by 1, and go back to the Step 2. Otherwise, report to the **output part** that Op_j cannot be occurred preserving security, and terminate the test.

3.2 Application analysis

The application test needs to maintain execution status of applications. When an application (denoted by App_t) is given as an input, an arbitrary state S_i (i is an integer number) of the execution system consists of *application ID, operation index, and a stack*. The application ID is the identifier of App_t. The operation index directs a term in the list of permission-consuming and permission-manipulation operation set of App_t. As we mentioned before, the application translator extracts permission-related operations from the application source code and generate an ordered list of those chosen operations. When App_t has n numbers of statements, we write jth term in the chosen operation list as $term_j$ ($1 \leq j \leq n$). The stack stores context information of an application, (application ID, #th term), when the application is temporarily switched out by dispatching another application. Overall, we denote the state S_i is represented by *StateTable* which is a triple *(AppID, CurrentTerm, StateStack)*, where the *StateStack* is a tuple of *(AppID, Term)*.

We suggest the application test for an application, App_t, carried out by the following steps:

[Step1] Construct an initial state, S_0. Set j to 1, and write the value of j into *StateTable.CurrentTerm*.

[Step2] If ($j > n$) and the *StateStack* is empty, then terminate the test process and report to the **output part** that App_t can finish its execution. If ($j > n$) and there is a data in the *StateStack*, pop the pair of *StateStack.AppID* and *StateStack.Term* from the *StateStack*, and replace the *StateTable.AppID* and *StateTable.CurrentTerm* with the *StateStack.AppID* and the *StateStack.Term*. Also, set j to the value of the *StateStack.Term*. Go to the Step 3.

[Step3] Assume that we are at a state S_i. Execute the current $term_j$. The $term_j$ should be one of permission-consuming operation or one of permission-manipulation operation.

If $term_j$ is a permission-consuming operation, then check if the App_t has enough permission by looking at the authorization table (but, in case of other frameworks where the dynamic permission revocation is allowed, we need to double check here if some of the authorized permissions are being on the revoked list). The $term_j$ can be executed only if the application has all the necessary permissions. If it is lack of permission, report it to the **output part** that the App_t cannot continue because of the lack of permission, and terminate the test. If the operation can pass the permission check, go to the Step 4.

If $term_j$ is a permission-manipulation operation, its execution result should be reflected to the current analysis. As we already have pre-/post-conditions of the operation, apply the operation descriptor to the current context. For example, if the operation is the start operation, it will push the current *StateTable.AppID* and the term index, j to the *StateStack.AppID* and *StateStack.Term*, respectively. And then, replace the current *StateTable.AppID* with the parameter of the start operation, set j to 1. Record the index j in *StateTable.CurrentTerm*. After we obtain the result of the operation, go back to the Step 2.

[Step4] After the $term_j$ is applied, check the security conditions. Assume that we have m condition elements. Apply all SC_k ($1 \leq k \leq m$) to the current state elements iteratively. If SC_k does not hold, then report that the security condition is violated, and terminate the test. If all Security Conditions hold, then increase the index j by 1, record the index in *StateTable.CurrentTerm*, and go back to Step2.

4 Discussion

The proposed design aims for an automated analysis tool, but still some embodiment challenges remain.

Firstly, some of the framework analysis process described in the sketch section cannot be fully-automated currently. In the Step 3 of the framework analysis, we try to prove the security property using theorem proving technique, but it is hard to be accomplished by a machine. Theorem proving generally known to need human expert's intervention, and our previous work[2] also was done by a human. Nevertheless, the automated analysis is not hopeless, because on the one hand, there are still ongoing researches of automatic theorem proving where they apply heuristic to the proving process. On the other hand, we have another type of formal technique, called model checking, which can investigate a particular property of a system automatically. Although the state space explosion problem hinders the model checking methods in large-sized com-

plex systems, it might be useful when we subdivide the analysis problem in manageable size. We are going to look at those approaches.

Secondly, it is hard to obtain some specification information. For the application test, we need to have the sets of required permissions of permission-consuming operations. The required permissions of a function can be grasped when the function contains the related permission string in their code explicitly. For example, the `BluetoothDeviceService` class of the Android framework has a function `checkPermissionBluetooth`, and it consequently calls `Context.checkCallingOrSelfPermission` to confirm if the caller has android.Manifest.permission.BLUETOOTH_ADMIN and android.Manifest.permission.BLUETOOTH. Therefore, we can conclude if any function calls those check functions, then it requires the latter two permissions. If the check function is found in the constructor of a class, any code that instantiates the class will require the related permissions. However, if the permissions are not written explicitly in the source code, but instead parameterized, then the required permissions are not easily gathered statically. We might need to run the framework and look at dynamic information the framework needs.

Moreover, if permission checking snapshots (or called Access Enforcement Function)are not placed in a certain point (like as security kernels usually place those functions at system call service routines), but are scattered over the framework source codes, the required permission gathering process gets more complicated. For example, in Android, when an activity tries to access internet using web browser, the android.permission.INTERNET permission will be checked at the constructor of the `WebKit.WebSettigs` class, and `verifyNetworkAccess` throws an exception when the caller does not have an appropriate permission. However, sometimes the internet access permission is checked and the exception is thrown at `OSNetworkSystem` of Apache Harmony which is a Java platform implementation that Android uses. It implies we might have to look at entire platform codes. However, the specification information still can be obtained anyway, and if we once collect the required permission information, we can proceed the application testing without obtaining the necessary permission information again. Therefore, the specification is worth having. We also utilize some instrumentation tools (Google's traceview and dmtracedump for Android), and it may reduce the specification difficulty.

Thirdly, the proposed design does not calculate the exact set of permissions for an application's execution. The fundamental reason is that our method is not a dynamic one, so we cannot know in advance how an application will be executed exactly. An application's execution sequence can be affected by runtime data it takes. Therefore, we try to estimate maximal set of permissions that an application would require, and tell whether the given authorization is sufficient or not for the application. When the test answers the given authorization is not sufficient, there is a still possibility that the given application can finish its execution. However, when the test tells the authorization is sufficient, the application cannot complete its execution and needs more permission. This can be rephrased as our testing method guarantees the sound authorization for an application's execution, but does not guarantee the complete authorization for an application's execution. The application testing is expected to be improved when the control flow analysis or the data flow analysis is applied.

5 Concluding Remark

We designed a software framework analysis tool that analyzes logical correctness of permission-manipulating operations of the framework and tests execution feasibility of an application with given authorization. Although some processes need manual assist currently, it is still feasible to embody the design. We also addressed the challenges of our design.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "Towards formal analysis of the permission-based security model for android," in *Proc. of International Conference on Wireless and Mobile Communications*, IARIA, 2009.