Special issue: The future of software engineering for security and privacy

Research Paper

PORTAM: Policy, requirements, and threats analyzer for mobile code applications

Haruhiko KAIYA[1], Kouta SASAKI[2], and Kenji KAIJIRI[3]
[1,2,3]*Shinshu University*

ABSTRACT

Users and providers of an information system should clearly understand the threats caused by the system as well as clarify the requirements for it before they actually use or develop it. In particular, they should be more careful when certain components or services are provided by third-parties. However, few tools can help identify and highlight threats to the security requirements. In this paper, we present a support tool called "PORTAM" for such users and providers to better understand the threats and the requirements. Suppose some requirements cannot be satisfied when some threats are avoided, and vice versa. In such cases, they should decide whether the requirements could be satisfied or the threats avoided. The tool also helps them to decide these kinds of trade-offs. The current version of this tool handles Java mobile code applications, thus users of our tool can readily understand the existence of real threats. Although the current version deals with only Java components, the ideas behind the tool can be applied to software in general. We complete this report by discussing some experimental results to confirm the usefulness for pedagogical purposes.

KEYWORDS

Requirements analysis, security policy, mobile code application, tool

1 Introduction

Mobile code technology is useful because software services can be integrated on the fly, that is, codes can be downloaded and assembled at runtime. In addition, alternative codes can be selected for meeting ad hoc changes to requirements preferences because the mobile software components can be reused at a finer granularity. For example, suppose there are many alternative codes for data communication, and their efficiency and license costs are different from each other. Although an integrator will normally select a code that is not relatively fast but cheap, in an urgent situation they could replace the code on the fly with another that is very fast but expensive.

One of the more significant problems that arises when using mobile codes is the existence of malicious codes. If malicious codes are not restricted, valuable resources can be leaked and/or destroyed. For example, your credit card information could be stolen. In this paper we call such harmful effects by malicious codes *threats*. Therefore, we have to identify which requirements should be satisfied and which threats should be avoided when we integrate a mobile code application. In many cases it is impossible to both satisfy all the requirements and completely avoid all the threats. One has to make a compromise with software systems to deal with unsatisfied requirements and tolerable threats. Therefore, we have to also decide on the trade-offs between satisfying the requirements and dealing with tolerant threats. Unfortunately, a user does not explicitly understand the importance of specifying the necessary requirements and threats of an information system, even through we meet the actual threats every day via the Internet.

We have already proposed a method to identify the tradeoffs between the requirements and threats for Java mobile code applications [11], [12]. However, it would

Received September 13, 2007; Revised November 19, 2007; Accepted December 11, 2007.
[1)] *kaiya@acm.org*, [3)] *kaijiri@cs.shinshu-u.ac.jp*
DOI: 10.2201/NiiPi.2008.5.3

be difficult evaluate our method without supporting tools, because the method requires tedious human effort. In this paper, we introduce a supporting tool and some results for applying the tool into security education. The main contribution of this paper is to show how much our tool can support a security requirements analysis and security requirements education.

Security issues in requirements engineering have recently become the focus of much research effort due to increasing use of Internet applications [6]. Our tool is designed to be simple enough for students to learn the importance of security requirements in a classroom. We hypothesize that learners can meet real threats caused by an application and they can analyze the reasons by using our tool. Such activities can improve their understanding of security requirements. This hypothesis was partially confirmed in our case study. Although there are many complex and complete models/tools for security requirements, it is not so easy for learners to have experiences where they run malicious programs by themselves. For this reason we choose to use the Java mobile code applications.

The rest of this paper is as follows. In the next section, we explain the Java mobile code applications mechanism. Although Java systems are rather simple, they resemble typical security problems. Section 3 summarizes the requirements of our analysis tool based on its previous discussion. In Section 4, we introduce our tool in more detail. In Section 5, we report on a case study we performed to confirm the usefulness and educational benefits of our tool. In Section 6, our related works are discussed, and finally we conclude with our current results and show some of our future work.

2 Mobile code applications

In this section, we explain the security architecture of the Java system. Although the Java security architecture supports fundamental security issues, such as integrity, confidentiality, and availability, it is very easy to understand and suitable for educating people in the importance of security requirements. Therefore, the Java system is suitable for educating people in the importance of security.

The Java security architecture is based on the sandbox security model [17]. There are a lot of security related features in the Java security architecture, but we only focus on the permissions and security policies, because the access control mechanism specified by these items is one of the most important security issues. Each permission corresponds to the right to access system resource(s), such as files, network connections, running processes, and so on. To grant the pieces the right to a Java application, the security policy is given to the application. Fig. 1 shows an example of the environment for a Java application. An application in this figure consists of three pieces of codes and it accesses a file and a property, and establishes a network connection to another machine.

When inadequate policy is given to an application, malicious codes can be activated, thus security threats can be made. Examples of threats for each type of security issue are listed as follows.

- Integrity: Files or system properties are illegally modified because the security policy inadequately grants the right to files or system properties.
- Confidentiality: Data are illegally leaked because the policy inadequately grants the right to files and network connections.
- Availability: Calculation is illegally aborted because the policy permits the right to kill the calculation process.

To avoid the threats caused by malicious codes, the codes are distinguished with respect to both the site where a code was placed and the signature, and are restricted in different ways. If a code is downloaded from a trusted site or a trusted agent signed the code, we may believe the code does not cause any threats.

However, we cannot or do not in fact always use only trusted codes, e.g., different kinds of free software. In addition, even the trusted codes may cause security threats because of internal bugs or our inadequate usage. Therefore, application integrators and users have to investigate which requirements are satisfied and what kinds of threats can be caused by a mobile code application.

We assume a support tool is needed and would be useful for investigating such issues. The usage of our tool is explained in detail in Section 4.3, and we report on its evaluation results in Section 5. We also assume archiving such investigation information can improve their understanding about mobile codes and security. A part of a case study outlined in Section 5 was designed for checking this assumption.

3 Requirements for supporting tool

Based on our previous works [11], [12] and the discussion in the previous sections, we specify the requirements for the tool as follows.

1. It shall support the requirements analysis for a Java mobile code application.
2. It shall record both an application's requirements and its threats, which are inconvenient results caused by the application.
3. It shall record the permissions that are required to satisfy each requirement.

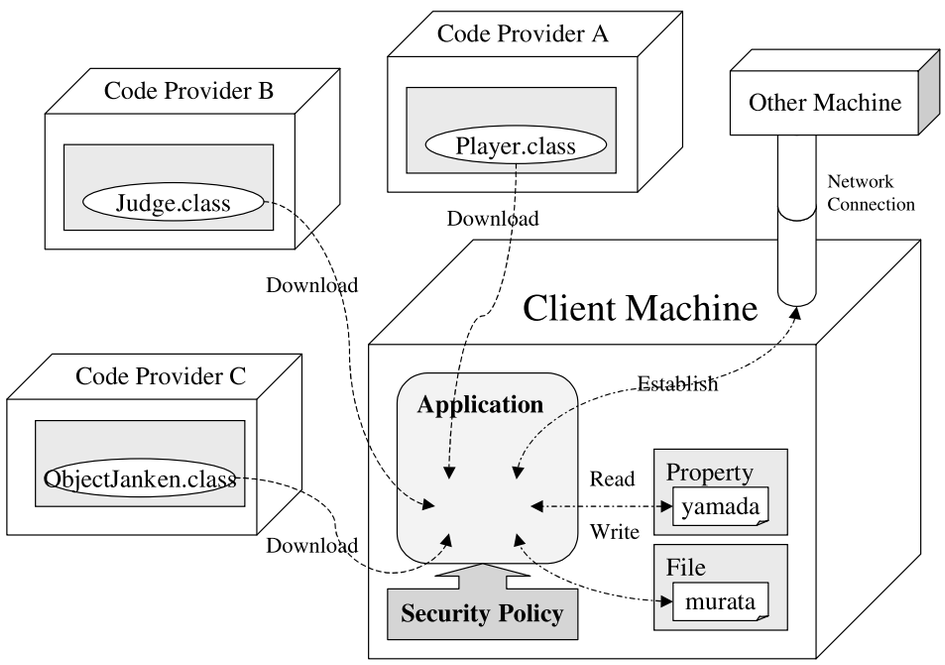

Fig. 1 Environment for a Java application.

4. It shall record the permissions that activate a threat.
5. It shall record the available mobile codes.
6. It shall be able to extract the security related permissions from each mobile code.
7. It shall be able to generate a security policy that grants permissions for the available mobile codes.
8. It shall be able to modify the security policy to satisfy the requirements and to avoid threats.
9. It shall be able to check whether the requirements are satisfied or not, and which threats are avoided or not under the security policy.
10. It shall allow its user to abandon some of the requirements and/or accept some of the threats if they choose to satisfy other requirements or avoid other threats.

4 Overview of PORTAM

This tool helps support application integrators and users in identifying which requirements are satisfied by a mobile code application. In addition, the tool also helps in identifying threats caused by the application. Since most mobile codes are intended for reuse, threats can be avoided by tightening up the security policies and/or by replacing the mobile code, including any malicious parts, with another compatible code. In some cases, some requirements cannot be satisfied because of tightened policies, thus we have to sometimes give up some of the requirements or to accept some threats. This tool also supports in finding such trade-offs.

4.1 Major functions

Our tool mainly provides the following six functions. By using such functions during a requirements analysis process, integrators and users can identify the achievements of the requirements and understanding the possibility of threats, and decide on the acceptable trade-offs between the requirements and threats.

4.1.1 Network deployment function

Our tool consists of several internal windows (Fig. 2). The top-left window, called the “Virtual Network Frame” window, is an analogical model of a deployment of computers, each of which provides mobile codes. In addition, the permissions that are required by such codes are semi-automatically extracted from the source or byte codes, and listed in a middle window called the “Permission Table” window. By using this function, users can understand what kinds of security related functions could be activated by each mobile code.

The extracted permissions are not always used in each run because the permissions are extracted by using a static source code analysis. In addition, the targets of each permission, such as the file names or host names, sometimes cannot be automatically extracted because they are sometimes represented as variables in the source codes. The tool’s user has to manually edit

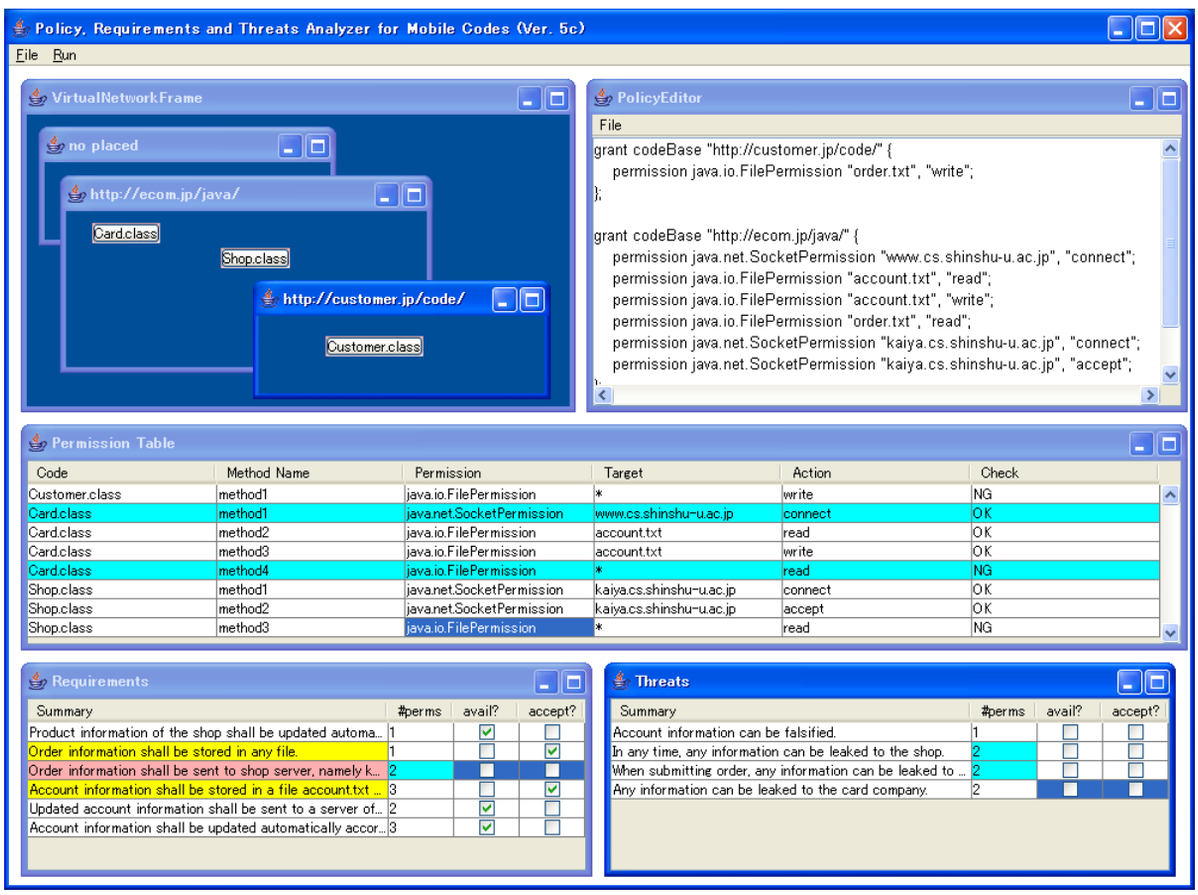

Fig. 2 A snapshot of PORTAM.

the targets in such cases.

4.1.2 Policy edit function

Based on the deployment of mobile codes shown in the "Virtual Network Frame" window (Fig. 2), our tool can automatically generate a security policy that grants all the permissions required by all codes. The generated policy is put in the right top window called the "Policy Editor" window, and users can freely edit the policy. The policy shown in this figure is already edited so as to revoke some permissions. Users may also edit a policy from scratch, but they can easily arrive at the intended policy by removing the granted permissions from the generated policy.

4.1.3 Policy check function

According to the policy in the "Policy Editor" window, each permission in the "Permission Table" window is automatically checked whether it works or not under the policy. The column labeled "Check" in the "Permission Table" window shows the results. For example, the first, fifth, and last permissions in Fig. 2 are marked "NG", thus they cannot work under the policy listed in Fig. 2. The outline of this algorithm is as follows.

```
foreach pol in (set of policy in PolicyEditor)
  foreach per in (set of permissions)
    per.Check="NG"
      if(pol.codeBase==per.Code.codeBase
        & pol.permission==per.Permission
        & pol.target_per.Target
        & pol.action_per.Action) then
          per.Check="OK"
      fi
    end
  end
```

In addition, we can easily identify which requirements or threats use the permission. By clicking a row in the "Permission Table" window, the requirements and threats in the "Requirements" and "Threats" windows are colored. For example, the tool in Fig. 2 tells us that the last permission in the "Permission Table"

window is used in the third requirement in the "Requirements" window and the second and third threats in the "Threats" window when we click a row in the "Permission Table" that corresponds to the permission. The "#perms" column of the rows in the "Requirements" and "Threats" windows corresponding to the requirement and threats is colored in blue.

4.1.4 Requirements and threats edit function

Users can list their requirements in the left bottom window labeled "Requirements" in Fig. 2. The requirements are simply itemed as shown in the figure. Users can also list their identified threats in the right bottom window labeled "Threats" in the same way. Each requirement or threat has the following properties.

- A list of permissions that are required by the requirement or the threat. A requirement is satisfied or a threat can be activated when all these corresponding permissions work. A tool user should manually specify the list of permissions because it is hard to automatically evaluate the meaning of such list. Because requirements and threats are written in natural language and our tool does not have the functionality for natural language processing, our tool cannot detect incorrect links between permissions and requirements/threats.

  As mentioned in Section 4.1.1, permissions are not always used in each runtime because of the static analysis. Therefore, the list of permissions does not always guarantee that a requirement is satisfied or a threat is activated. However, the list of permissions helps a requirements analyst to explore the possibility of threats, because such permissions are sometimes used in runtime.

- A truth value whether a requirement or a threat can be activated. If all permissions above are "OK", the value is true, otherwise it's false.

- A truth value whether the user abandons the requirement or accepts the threat.

These properties are shown on the GUI of PORTAM. For example, if the "Threats" window in Fig. 2 is the focus, we can see there are four threats listed. If we then focus on the last threat "any information can be leaked to the card company", we can see that the threat has two related permissions. The permissions are the second and fifth ones, and they are colored in the middle window "Permission Table". Because the fifth permission in the "Permission Table" is "NG", the threat cannot be activated. Thus, the check box "avail?" of the threat is not checked. We do not have to accept this threat, because the threat cannot be activated in this situation.

4.1.5 Requirements check function

Whether a requirement is satisfied or not is decided according to the status of permissions related to the requirement. A requirement has three states, which are shown by given colors.

- Satisfied status (white): The corresponding requirement will be satisfied, because all related permissions can be permitted.

- Accepted status (yellow): Although the requirement cannot be satisfied by the revoked permissions, the user decides to abandon the requirement.

- Unstable status (pink): The requirement cannot be satisfied now, but the user does not accept the fact. The user has to decide to accept the fact or to modify the security policy to change the status of the requirement.

To finish the requirements analysis, the requirements that are unstable should be completely eliminated. In Fig. 2, there are six requirements. The second and fourth requirements are colored in yellow, thus they are accepted. The third requirement is colored pink, thus it is unstable.

4.1.6 Threat check function

Whether a threat can be avoided or not is also decided according to the status of permissions related to the threat. There are also three statuses of a threat and the status is also shown by color on the GUI.

- Avoided status (white): The corresponding threat will be avoided because at least one of the related permissions cannot be permitted.

- Accepted status (yellow): Although the threat can be activated by all the granted permissions, the user decides to accept the threat.

- Unstable status (pink): The threat can be activated now, but the user does not accept the fact. The user has to decide to accept the fact or to modify the security policy to change the status of the threat.

In the same way as with the requirements check, the threats that are unstable should be completely eliminated to finish the requirements analysis. In Fig. 2, there are four threats, and they are all in the avoided status.

4.2 Technologies used in our tool

To extract permissions required by a code, we have to analyze the code, especially the method calls from the code to other codes. Security-related permissions

are required only when specific methods in Java standard API are called for. Thus, our tool is good enough to identify the method calls from the code to such API methods. Our tool is able to conduct a static source code analysis by using an XML-based representation for Java source codes (JavaML). Our tool invokes an extended version of a Java compiler called jikes[1] to convert a Java source code to a JavaML representation. The converter is not written in Java, thus our tool invokes it as an external program.

Our tool can handle Java class files (byte codes) by using a decompiler called jode.[2] Our tool was developed as a Java application and the decompiler jode was also written in Java, thus it is easily invoked from our tool.

4.3 Typical processes

One of the typical processes when using this tool is to explore threats and policy under given mobile codes and requirements. A case study in the next section is categorized in this way. According to the codes and their deployment, our tool can list the permissions in the codes in the "Permission Table". By using the requirements and threats edit function of our tool mentioned in Section 4.1.4, each requirement is related to several permissions. A user should explore the threats to browse the list of permissions and note that each threat is also related to several permissions. By using the policy edit function of our tool mentioned in Section 4.1.2, a security policy that grants all permissions can be generated by the "Policy Editor". By removing some lines from a policy in the "Policy Editor", the granted permissions are decreased in general, and vice versa. When granted permissions are changed, satisfied requirements and threats are also changed. The requirements, threats, and the policy are decided by repeating such changes. Finally, a user has to decide which requirements may be abandoned and which threats should be accepted with respect to the needs of the application users.

Another typical process is to explore suitable mobile codes and policy under several requirements and threats. To satisfy a requirement, several permissions are required. A user drops the existing mobile codes on "Virtual Network Frame" to explore the required permissions. To avoid a threat, some permissions should not be activated. A user writes the security policy for blocking the activation of such permissions.

[1] http://www.badros.com/greg/JavaML/

[2] http://jode.sourceforge.net/

5 Case study

The objective of this case study is to confirm the usefulness of our tool. If the following metrics are improved by using our tool, we may assume that our tool is useful in requirements elicitation.

- The amount of threats to be found.
- The amount of fatal threats to be found. Fatal threats are defined for each exercise and subjects off course do not know them before performing the exercise.
- The amount of wrong threats to be found. We decide if a threat is wrong when the threat cannot be activated under the given permissions. Because we cannot objectively decide that a threat is really inconvenient for the subject, we do not take such inconvenience into account.
- The efficiency of an elicitation task.

As explained in Section 4.1, our tool cannot automatically find threats, but only support an analyst in finding threats. Therefore, this kind of experimental study is required. Based on the assumptions above, we designed our experiment as follows.

Another objective of this case study is to confirm the educational effects by analyzing the requirements and threats. As mentioned in the introduction, we do not clearly understand the importance of specifying both the requirements and threats of an information system even though we meet the actual threats every day via the Internet. After performing the experiment, we sent out questionnaires to confirm the educational effects. The contents, results, and discussion are shown in Section 5.4.

5.1 Experimental design

We performed a comparative experiment by using several subjects. Each subject performed the following six exercises.

S1: Learn the mechanism of the security policy for executing a mobile code application. Since most of the subjects are unfamiliar with mobile code applications, they have to become familiar with such applications through this exercise.

S2: Write their own security policy to satisfy the given requirements. In this exercise, the existence of threats is secret before the answer is shown. From this experiment, the subjects can better understand the existence of potential threats in mobile codes.

S3: Write their own security policy to satisfy the given requirements and to avoid the given threats.

S4: Perform the same exercises using our tool. This exercise is simply used to help the subjects learn how to use the tool.

S5: Write a security policy to satisfy the given requirements and find any threats. Note that the codes, their deployment, and the policy that grants all the permissions in the codes are also given. Half of the subjects are permitted to use our tool, but the other half are not. Whether a subject is permitted to use a tool or not is decided at random.

S6: Perform the same kind of exercise as in S5 using other requirements and codes. The subjects using our tool in S5 are not permitted to use our tool, but the others are.

Only the data in exercises S5 and S6 are used in our analysis, because other exercises are basically used to help the subjects become familiar with mobile code applications and our tool. For each result from exercises S5 and S6 conducted by each subject, we counted the following values.

- The number of threats that are found by the subject.
- The number of fatal threats.
- The number of wrong threats.
- The time spent to solve the problems in the exercise.

By comparing the results using our tool to the results from the subjects without tool support, we confirmed the following hypotheses.

1. Subjects that used our tool could find more threats than those without it.
2. Subjects that used our tool could find more fatal threats than the others.
3. Subjects that used our tool could find fatal threats more frequently than the others. In other words, the ratio of fatal threats found by the subjects was higher than the ratio by the others.
4. Subjects that didn't use our tool wrote more wrong threats than the others.
5. Subject that didn't use our tool wrote wrong threats more frequently than others. In other words, the ratio of wrong threats by the subjects without our tool was higher than the ratio by the others.
6. Subjects that used our tool could perform the exercises more efficiently than the others.

5.2 Results

We were able to conduct this experiment through a course at our university. The course took five weeks, thus exercises S3 and S4 were performed in the same week. In each week, we spent three hours on classwork that included these exercises. To improve the understanding of mobile codes applications, attendance checks and answer submissions were achieved by mobile code applications. About 30 third-grade bachelor students participated in this course. They have already studied software engineering fundamentals and Java programming. We selected 20 students as our subjects based on the following conditions.

- The student was able to perform all six exercises.
- The student agreed to the fact that his/her data was to be used in our research.

In exercise S5, each subject analyzed a shopping system via the Internet. Five requirements were shown and this exercise had the following three fatal threats.

1. An account can be falsified.
2. User information can be leaked to shops.
3. User information can be leaked to a credit card company.

In exercise S6, each subject analyzed an e-learning system via the Internet. Six requirements were shown and this exercise had the following two fatal threats.

1. The correct answer can be leaked to other learners.
2. A learner can unfairly read the others' answers (cheating).

Tables 1 and 2 show the results. In each experiment, only half of the subjects used our tools. If a subject used our tool in S5, the subject could not use the tool in S6 and vice versa. The column labeled "tool" in the

Table 1 Results of S5 (Average per subjects).

	tool	manual	$\lvert t_0 \rvert$
Number of threats (X)	2.9	3.3	0.791
Number of fatal threats (Y)	1.4	1.0	0.801
Number of wrong threats (Z)	0.6	0.7	0.325
Ratio of fatal threats (Y/X)	0.48	0.30	–
Ratio of wrong threats (Z/X)	0.206	0.212	–
Spending minutes	167	168	0.248

Table 2 Results of S6 (Average per subjects).

	tool	manual	$\lvert t_0 \rvert$
Number of threats (X)	2.5	3.5	1.081
Number of fatal threats (Y)	0.7	0.8	0.341
Number of wrong threats (Z)	0.4	1.3	2.102
Ratio of fatal threats (Y/X)	0.28	0.22	–
Ratio of wrong threats (Z/X)	0.16	0.37	–
Spending minutes	161	153	0.774

tables is the data from the subjects using our tool. On the other hand, the column labeled "manual" is the data from the subjects that did not use our tool. Each result is the average of each type of subject. For example, in experiment S5, a subject found 1.4 fatal threats on average when using our tool. Because there were three fatal threats in S5, we can see that a given subject found only half the fatal threats on average. To confirm the statistical significance, we conducted a t-test. The third column in the table shows each t-value and the null hypothesis is that each number in the table is the same in both cases. Because $t0.005,18$ is 2.1009, only the number of wrong threats in S6 is significantly different in both cases.

5.3 Discussion

As shown in Tables 1 and 2, we were unable to confirm all the hypotheses in Section 5.1. In particular, the subjects without the tool ("manual" column) found more threats than the others. However, subjects without the tool tended to write more wrong threats than the others, as shown in the third and fifth rows in Tables 1 and 2. Thus, our tool seems to contribute to the accuracy for finding threats. Although the ratio of wrong threats is relatively low (0.2 and 0.16), we investigated into the reason for the wrong threats. When a subject with our tool finds and specifies a threat, the subject has to relate the threat to several permissions. We assume there is a gap between the threat and the permissions, because the permissions show the concepts at the implementation stage, but the threats do not. Our tool has to support the filling of such a gap.

The tool also seems to contribute to finding fatal threats slightly based on the fourth row, "the ratio of fatal threats", in the tables. By investigating the contents of the threats written by the subjects, the function making relationships between a threat and the permissions seem to contribute to finding fatal threats, because the combination of permissions causes fatal threats. We will discuss how to improve our tool based on the discussion here in the final section.

5.4 Questionnaires

Our questionnaires consisted of the following three kinds of issues. Each issue has several questionnaires as follows.

- Issues in a mobile code application.

  M1. Can you understand the running mechanism of mobile code applications? [yes/no]

  M2. Can you understand the role of a security policy? [yes/no]

  M3. Do you wish to use mobile code applications in the future? [yes/no]

  M4. Do you wish to develop mobile code applications in the future? [yes/no]

- Issues in the requirements analysis for mobile code applications.

  R1. Are you more careful when using libraries or programs provided by others? [yes/always did/no]

  R2. Do you understand the existence of threats in mobile code applications? [yes/no]

  R3. Do you understand that there are sometimes trade-offs between satisfying requirements and avoiding threats? [yes/no]

  R4. Do you think compromises are sometimes necessary in requirements and threats analysis? [yes/no]

  R5. Do you now think the users of an information system should identify both the requirements for a system and the threats by the system? [yes/always did/no]

  R6. Do you now think the developers of an information system should identify both the requirements for a system and the threats by the system? [yes/ever since/no]

- Issues concerning our support tool.

  T1. Do you understand how to use our tool? [yes/no]

  T2. Did our tool help you find threats and/or forgotten requirements? [yes/partially/no]

Table 3 Results of questionnaires.

	yes	ever since	no
M1	17		3
M2	18		2
M3	16		4
M4	12		8
R1	11	4	5
R2	20		0
R3	20		0
R4	19		1
R5	14	3	3
R6	16	4	0
	yes	partially	no
T1	17		3
T2	9	10	1

Note that "always did" means that the subject thought or understood the issue before this experiment. Table 3 shows the results from the questionnaires. Our subjects understood the mobile code application itself, but some of them worried about the unidentified threats. Another complained that he did not always connect to the Internet and he did not always use mobile code applications. There is a kind of mobile codes application that works under disconnected circumstances, but we did not mention it in this course. We have to introduce such an application to eliminate such complaints. Most subjects mentioned the easiness of updating software or the usefulness of it as positive reasons with respect to the users. With respect to the developers, more subjects gave negative answers. Typical reasons were as follows; it was hard to manage multiple versions of the codes or to take threats into account during development.

About the issues of requirements analysis, our questionnaires showed positive results, as shown in Table 3. One subject who answered "no" in R1 wrote a comment that it was difficult to avoid threats even if he could identify them. Thus, we have to provide an effective mechanism for avoiding threats in the next step. From the results of R5 and R6, our subjects assumed that the developers were more responsible for identifying the requirements and threats than the users. This assumption seems to be quite rational because the developers have more knowledge than the users. So, our tool is dedicated to the application developers or integrators rather than the users.

Finally, we reviewed the evaluation results of our tool used by our subjects. From the results of T1 and T2 in Table 3, their evaluation was not bad. It seemed to be easy for our subjects to learn our tool because less than three hours were spent to learn it and the result T1 tells us most of them could understand how to use it. Typical positive comments were as follows.

- By using color changes, it is easy to identify the permissions that are used in a requirement or a threat.
- In the same way, it is easy to identify the requirements and threats that depend on the same permission.
- Automatically generated minimal policy is useful. However, there were following negative comments.
- It is inconvenient to make the relationship between a requirement or a threat and the permissions.
- Because each permission is separated from the class where the permission belongs, it is difficult to understand the role of the permission.

5.5 Threats to validity

According to [19], we will discuss the threats to the validity of this experiment.

- Conclusion validity: As mentioned in Section 5.2, most of the differences between with or without tool support were not statistically significant.
- Internal validity: According to the results of the questionnaires, the usability of our tool was not so bad. However, the learning effects could have a bad influence on the result, that is, the subjects were not so familiar with our tool.
- Construct validity: Because our tool is expected to improve the security requirements definition, treatments in our experiment greatly reflected the theory of our tool.
- External validity: Our results may be generalized to the average student in a software engineering course in Japan. However, the results cannot be generalized to some industry people working in the security area.

6 Related work

We assume that information systems in this research field use fine-grained software components such as functions and/or classes. If these kinds of reuse are widely accepted, the variety of components selection available will largely increase and the markets of such components will soundly grow. There are already many researches concerning component selection and acquisition [8], [14]. However, threats caused by components compositions have rarely been discussed. This research and the tool directly handle such issues and partially support users and software integrators in dealing with such threats.

In general, requirements elicitation using an interview is costly and [3] argued that a method to make it more efficient is required. By using our tool, requirements analysts can enumerate the potential threats, and thus, the tool helps such analysts to ask and explore what should not occur in an information system to be developed and marketed. This function largely and efficiently contributes to eliciting requirements.

Threats in this paper are very similar to obstacles in KAOS [18]. The difference is that the threats do not have to obstruct existing requirements, but obstacles are basically identified by obstructing existing requirements or goals. Thus, threats in this paper are not

easily identified using the KAOS approach. The misuse case approach is also a useful method for identifying security requirements, but its weakness was argued in [16]. Our tool can partially overcome such weaknesses, for example, the process navigated by our tool is not open-ended, but systematically terminated if the user can compromise on a specific policy and its consequences; giving up requirements and/or accepting threats. A software fault tree [10] is also a systematic approach, but it is specialized for the requirements analysis of intrusion detection systems. A system called SoftwarePot [13] can also be applied to the problems we have focused on. In SoftwarePot, applications are executed in some kind of sandbox, and users have to decide whether access to the valuable resources should be granted or not each time. We think the SoftwarePot approach seems practical, but it does not contribute to improving the users' understanding of security-related problems.

Our research and tool focuses on one application used by one user rather than an information system used in an organization (many users). Thus, our research does not and cannot handle multiple users and/or roles in an information system, because the application we focused on basically has only one role. Taking such multiple roles into account, modeling techniques, such as those in [7] or [9] are required. With respect to the Java specification and implementation, we only focused on the so-called "code-centric style" for right now. Therefore, we are not concentrating on "who runs/executes a function" at this time. The Java system already has a mechanism called a "user-centric style" in the Java Authentication and Authorization Service (JAAS) framework, so we want to extend our tool by taking the roles into account when using the JAAS framework.

In a standard Java SDK, there is a tool called the "policytool" that is used to define security policy. The main differences between PORTAM and policytool are as follows. First, PORTAM can explicitly manage requirements and threats. Second, PORTAM can check whether a requirement is satisfied or a threat is achieved based on a static code analysis.

In [2], an organizational policy is handled, but our research is about the security policy for an application. Thus, the discussion and results in both researches cannot be simply compared. We think an organizational policy is a sum or product set of policies of applications in the organization. Thus, defining each application policy will sometimes contribute to defining the organizational policy.

In contrast to other researches about security requirements, our work is too simple. However, this is one of its advantages, because our tool can be easily and effectively applied to educational settings. As shown in the case study in the previous section, students could easily use our tool, and they could experience actual threat activations. In fact, the educational materials in our case study embedded real malicious codes such as stealing personal information, and the codes were sometimes activated during the course. By facing such real threats, the students could more deeply understand the importance of identifying the threats as well as the requirements. Tools and/or methods of other researches seem to be too complex to use in educational settings. There were a few researches about the requirements for engineering education [5], [20], but there is no research concerning the educational aspects of security requirements. As reported in the previous section, a lecture using our tool had a profound effect on the students with respect to better understanding requirements and threats.

In our research and tool usage, decision making, such as how to reach trade-offs and/or mitigate threats, is out of the scope. Existing research results, such as WinWin [1], DDP [4] and/or [15] can cope with our tool.

7 Conclusion

We introduce a tool in this paper called PORTAM. The tool supports users, software providers, and/or integrators in identifying the requirements, threats, and policies for mobile code applications. Because our tool handles Java mobile code applications, the users of our tool can easily find the threats caused by such applications during their analysis. We found from the results of our case study that our tool contributed to helping learners better understand the importance of threats as well as requirements. Our tool also contributed to finding significant threats.

Basically, threats can be caused by a combination of several permissions, Users of our tool currently have to manually find such combinations. We will extend our tool to propose possible combinations. It is difficult to decide whether a combination causes threats or not, but it is not difficult to enumerate possible combinations by using the dependencies and flows of data. Such combinations can also suggest some of the unidentified requirements of users. In addition, our tool has to support stopping the gaps between such a combination and the meaning of a requirement or a threat as discussed in Section 5.3. Such gaps can be stopped by using empirical knowledge such as the design patterns, because the semantic processing of a requirement or a threat is very difficult. Another plan is to provide a comparison mechanism of alternative codes. Currently, users have to manually replace a code with another, but the tool should recommend alternatives. If such a comparison mechanism is provided, we can also compare the

non-functional features, such as the costs or response of codes. The current version of our tool does not explicitly handle the priority among requirements and threats. The tool currently enables its user to put requirements and threats in order, respectively, but the ordering is not used formally in our tool. We will extend our tool by using such a priority e.g., deciding on the trade-offs. Finally, we have to explore the generalization of our tool. Our tool largely depends on the Java language and its policy language. By introducing more abstract representations of policies, we can handle more policies than now. In the case of programming and specification languages limited by the policies, it is a little bit difficult to generalize them because there are various kinds of languages.

References

[1] B. Boehm, P. Grunbacher, and R.O. Briggs, "Developing Groupware for Requirements Negotiation: Lessons Learned." *IEEE Software*, vol. 8, no. 3, pp.46–55, May/Jun. 2001.

[2] T.D. Breaux and A.I. Anton, "Analyzing Goal Semantics for Rights, Permissions, and Obligations." In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp.177–188, 2005.

[3] T. Cohene and S. Easterbrook, "Contextual Risk Analysis for Interview Design." In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp.95–104, 2005.

[4] S.L. Cornford, M.S. Feather, J.C. Kelly, T.W. Larson, B. Sigal, and J.D. Kiper, "Design and Development Assessment." In *Proceedings of the Tenth International Workshop on Software Specification and Design (IWSSD'00)*, pp.105–114, 2000.

[5] C. Coulin and D. Zowghi, "GONDOLA: An Interactive Computer Game-Based Teaching and Learning Environment for Requirements Engineering." In *REFSQ'04*, pp.113–126, 2004.

[6] R. Crook, D. Ince, L. Lin, and B. Nuseibeh, "Security Requirements Engineering: When Anti-requirements Hit the Fan." In *IEEE Joint International Requirements Engineering Conference, RE'02*, Essen, Germany, pp.203–205, Sep. 2002.

[7] R. Crook, D. Ince, and B. Nuseibeh, "On Modelling Access Policies: Relating Roles to their Organisational Context." In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp.157–166, 2005.

[8] X. Franch and J. Pablo Carvallo, "Using Quality Models in Software Package Selection." *Software*, vol. 20, no. 1, pp.34–33, Jan./Feb. 2003.

[9] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, "Modeling Security Requirements Through Ownership, Permission and Delegation." In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp.167–176, 2005.

[10] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, and R. Lutz, "A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System. *Requirements Engineering*, vol. 7, no. 4, pp.207–220, Dec. 2002.

[11] H. Kaiya, K. Sasaki, and K. Kaijiri. A Method to Develop Feasible Requirements for Java Mobile Code Application. *IEICE Trans. Inf. and Syst.*, E87-D(4):811–821, Apr. 2004.

[12] H. Kaiya, K. Sasaki, Y. Maebashi, and K. Kaijiri, "Trade-off Analysis between Security Policies for Java Mobile Codes and Requirements for Java Application." In *11th IEEE International Requirements Engineering Conference*, Monterey Bay, California, pp.357–358, Sep. 2003.

[13] K. Kato and Y. Oyama, "SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation." *Lecture Notes in Computer Science*, vol. 2609, pp.112–132, 2003.

[14] S. Lauesen, "COTS Tenders and Integration Requirements." In *12th IEEE International Requirements Engineering Conference (RE'04)*, pp.166–175, 2004.

[15] M.C. Robinson, S.E. Wallace, and D.C. Woodward, "Risk Mitigation of Design Requirements Using a Probabilistic Analysis." In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp.231–239, 2005.

[16] G. Sindre and A.L. Opdahl, "Eliciting security requirements with misuse cases." *Requirements Engineering*, vol. 10, no. 1, pp.34–44, Jan. 2005.

[17] Sun Microsystems, Inc. *Java Security Architecture (JDK1.2)*, Oct. 1998. Version 1.0.

[18] A. van Lamsweerde. "Elaborating Security Requirements by Construction of Intentional Anti-Models." In *Proceedings of ICSE'04, 26th International Conference on Software Engineering*, Edinburgh, pp.148–157, May 2004.

[19] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering An Introduction*. Kluwer, 2000.

[20] D. Zowghi and S. Paryani. "Teaching Requirements Engineering through Role Playing: Lessons Learnt." In *12th IEEE International Requirements Engineering Conference (RE'04)*, pp.233–241, 2004.

Haruhiko KAIYA

Haruhiko Kaiya is an associate professor of Software Engineering in Shinshu University, Japan. He is also a visiting associate professor in NII. http://www.cs.shinshu-u.ac.jp/~kaiya/

Kouta SASAKI

Kouta Sasaki was a graduate school student in Shinshu University, Japan.

Kenji KAIJIRI

Kenji Kaijiri is a professor of Software Engineering in Shinshu University, Japan.