Visualization of Permission Checks in Java using Static Analysis

Yoonkyung Kim and Byeong-Mo Chang

Department of Computer Science, Sookmyung Women's University
Yongsan-ku, Seoul 140-742, Korea
{ykkim79,chang}@sookmyung.ac.kr

Abstract. The security manager in Java 2 is a runtime access control mechanism. Whenever an access permission to critical resources is requested, the security manager inspects a call stack to examine whether the program has appropriate access permissions or not. This run-time permission check called *stack inspection* enforces access-control policies that associate access rights with the class that initiates the access. In this paper, we develop a visualization tool which helps programmers enforce security policy effectively into programs. It is based on the static permission check analysis which approximates permission checks statically which must succeed or fail at each method. Using the visualization system, programmers can modify programs and policy files if necessary, as they examine how permission checks and their stack inspection are performed. This process can be repeated until the security policy is enforced correctly.
Keywords: Java, stack inspection, security, static analysis.

1 Introduction

Java was designed to support construction of applications that import and execute untrusted code from across a network. The language and run-time system enforce security guarantees for downloading a Java applet from one host and executing it safely on another. Bytecode verification is the basic building block of Java security, which statically analyzes the bytecode to check whether it satisfies some safety properties at load-time [8, 18].

While the bytecode verifier is mainly concerned with verification of the safety properties at load-time, the security manager in Java 2 is a runtime access control mechanism which more directly addresses the problem of protecting critical resources from leakage and tampering threats. Whenever an access permission to critical resources is requested, the security manager inspects a call stack to examine whether the program has appropriate access permissions or not. This run-time permission check called *stack inspection* enforces access-control policies that associate access rights with the class that initiates the access. A permission check passes stack inspection, if the permission is granted by the protection domains of *all* the frames in the call stack.

In Java 2, programmers implement a security policy of an application by writing its security policy file and checking whether an access request to resource should be granted or denied, before performing a possibly unsafe or sensitive operation. Programmers should examine whether the security policy is kept well in the program as expected. This examination usually requires a lot of effort, when programs are large and different permissions are needed for different classes. So, we need a tool to support this permission check examination to develop secure programs in Java 2 effectively.

In this paper, we develop a visualization tool which helps programmers enforce security policy effectively into programs. It is based on the static permission check analysis proposed in [6], which approximates permission checks statically which must succeed or fail at each method. We first implement the static permission check analysis. Based on the static analysis information, we implement a visualization system, which shows how permission checks and their stack inspection are performed.

Using the visualization system, programmers can modify programs and policy files if necessary, as they examine how permission checks and their stack inspection are performed.

This paper is organized as follows. The next section reviews Java 2's stack inspection. Section 3 describes two proposed static analyses. Section 4 describes implementation of the static analysis and its visualization. Section 5 discusses related works. Section 6 concludes this paper with some remarks.

2 Stack inspection

Java 2's access-control policy is based on *policy files* which defines the access-control policy for applications. A policy file associates *permissions* with *protection domains*. The policy file is read when the JVM starts.

The `checkPermission` method in Java determines whether the access request indicated by a specified permission should be granted or denied. For example, `checkPermission` in the below will determine whether or not to grant `"read"` access to the file named `"testFile"` in the `"/temp"` directory.

```
FilePermission perm = new FilePermission("/temp/testFile","read");
AccessController.checkPermission(perm);
```

If a requested access is allowed, `checkPermission` returns quietly. If denied, an `AccessControlException` is thrown. Whenever the method `checkPermission` is invoked, the security policy is enforced by stack inspection, which examines the chain of method invocations backward. Each method belongs to a class, which in turn belongs to a protection domain.

When `checkPermission(p)` is invoked, the call stack is traversed from top to bottom (i.e. starting with the frame for the method containing that invocation)

until the entire stack is traversed. In the traversal, the stack frames encountered are checked to make sure their associated protection domains imply the permission. If some frame doesn't, a security exception is thrown. That is, a permission for resource access is granted if and only if all protection domains in the chain have the required permission.

Privilege amplification is supported by `doPrivileged` construct in Java. By invoking `AccessController.doPrivileged(A)`, a method `M` performs a privileged action `A`; this involves invoking method `A.run()` with all the permissions of `M` enabled. This can be seen as marking the method frame of `M` as privileged: stack inspection will then stop as soon as a privileged frame (starting from the top) is found [2].

In Java, the normal use of the "privileged" feature is as follows [18] :

```
somemethod() {
     ...normal code here...
     AccessController.doPrivileged(new PrivilegedAction() {
         public Object run() {
             // privileged code goes here, for example:
             System.loadLibrary("awt");
             return null; // nothing to return
         }
     });
     ...normal code here...
}
```

This type of normal privileged call is assumed for simple presentation in this paper.

When inspecting stack, the `checkPermission` method stops checking if it reaches a caller that was marked as "privileged" via a `doPrivileged` call. If that caller's domain has the specified permission, no further checking is done and `checkPermission` returns quietly, indicating that the requested access is allowed. If that domain does not have the specified permission, an exception is thrown, as usual.

In summary, stack inspection checks the chains of method invocations backward until either the entire stack is traversed or an invocation is found within the scope of a `doPrivileged` call.

Java's stack inspection policy can also handles dynamic creation of threads. When a new thread T is created, T is given a copy of the existing run-time call stack to extend. The success of subsequently evaluating `checkPermission` in thread T thus involves permissions associated with the call stack when T is created.

3 Static Permission Check Analysis

The static permission check analysis is done based on simple call graph which can be defined as follows.

Definition 1. *A call graph $CG = (N, E)$ is a directed graph, where N is the set of nodes which represent methods, and $E \subseteq N \times N$ is the set of edges, which represents method calls.*

There are two kinds of edges in the call graph. A normal edge $n \rightarrow n'$ represents a normal method call from n to n'. Thread `start` is also considered as a normal method call to its `run` method. A privileged edge $n \rightsquigarrow n'$ represents a `doPrivileged` call from n to n'. This represents `doPrivileged` call to a privileged action n', which is usually a method `A.run()`, with all the permissions of n enabled. The soundness of call graph is shown in many literature [15, 10]. This call graph is unlike the call graph in [1], in that it doesn't contain any intra-procedural control flow.

In the following, we abbreviate `checkPermission(p)` by $check(p)$. We denote by $check(p) \in m$ if $check(p)$ occurs in a method m. The set of all permission checks in a program is denoted by $Check$. The set of permissions associated with a method m is denoted by $Permissions(m)$, which is determined by a policy file which associates *permissions* with *protection domains*, to which methods belong.

We can say that a permission check $check(p)$ in a method m *succeeds* at a method n, if the permission p is granted by all the stack frames from the method m to the method n by stack inspection. If a permission check succeeds at a method m, the stack inspection can go further across m.

We will approximate all checks that may succeed at each method by *static analysis*. Then we can compute all checks in a simple way which must fail.

Definition 2. *A permission check $check(p)$ in a method m* may succeed *at the entry to a method n, if there exists a path from the method n to the method m in the call graph, along which the permission p is granted by all the methods in the path.*

Based on the simple call graph, we first define a backward static analysis called *May-Succeed Check Analysis*, which gives a safe approximation of permission checks which may succeed at each method. The *May-Succeed Check Analysis* will determine:

for each node(method), which permission checks *may succeed* at the entry to the node.

The May-Succeed Check Analysis is defined by the flow equation in Figure 1, where $May - SC_{entry}(n)$ includes $check(p)$'s in the method n or in $May -$

$$May - SC_{exit}(n) = \begin{cases} \emptyset & \text{if } n \text{ is final} \\ \bigcup\{May - SC_{entry}(m) | n \to m \in E\} & \text{otherwise} \end{cases}$$

$$May{-}SC_{entry}(n) = \{check(p) \in May{-}SC_{exit}(n) | p \in Permissions(n)\} \cup gen_{May-SC}(n)$$

where $gen_{May-SC}(n) = \{check(p) | check(p) \in n, p \in Permissions(n)\}$

Fig. 1. Flow equation for May-Succeed Check Analysis

$SC_{exit}(n)$ such that the permission p is granted by the method n. Note that only normal calls denoted by $n \to m$ are considered in the equation $May{-}SC_{exit}(n)$.

The flow equation in Figure 1 defines a transfer function $\mathcal{F}_{May-SC} : \mathcal{L} \to \mathcal{L}$, where the property space $\mathcal{L}$ is a complete lattice $\mathcal{L}_{entry} \times \mathcal{L}_{exit}$ where $\mathcal{L}_{entry}$ and $\mathcal{L}_{exit}$ are total function spaces from N to 2^{Check}. We can compute the least solution $(may - sc_{entry}, may - sc_{exit}) \in \mathcal{L}$ of the flow equation in Figure 1 by $lfp(\mathcal{F}_{May-SC})$ in finite time, because the finite property space $\mathcal{L}$ satisfies the ascending chain condition and the transfer function is monotonic. See [6] for details.

We prove the soundness of the analysis in the following theorem. In the theorem, we only consider actual normal call chains which don't contain a privileged call, because stack inspection cannot go further across a privileged call. See [6] for details.

Theorem 1. *For every actual normal call chain from a method n to a method m which contains $check(p)$, if the permission p is granted by all the methods in the call chain, then $check(p)$ is in $may - sc_{entry}(n)$.*

As an example, we consider a client-part of small e-commerce example in [2]. As described in [2], the user agent runs a Java-enabled Web browser, which has the rights to access the local file system and to open a socket connection. `Shop` and `Robber` are client-tier components implemented as Java applets. The `Browser` class provides the applets with some methods to manage the user preferences: the `getPref()` method tries to retrieve the preferences from a local file if the applet has the rights to do so. Otherwise, it opens a socket connection with the remote server. The `changePrefs()` method first looks for the old preferences (either in the local disk or on the remote server); then it asks for the new preferences, which are thereafter saved on the local disk (if the applet has the rights to do so) or sent to the remote server.

Its call graph and the security policies are shown in Figure 2. Unlikely to [1, 2], our static analysis is based on simple call graph. The May-Succeed Check Analysis computes checks which may succeed at the entry of each method, which are shown in Figure 3. Note that $check(Pread)$ and $check(Pwrite)$ may succeed at `Shop.start()`, and $check(Pconnect)$ may succeed at `Robber.start()`.

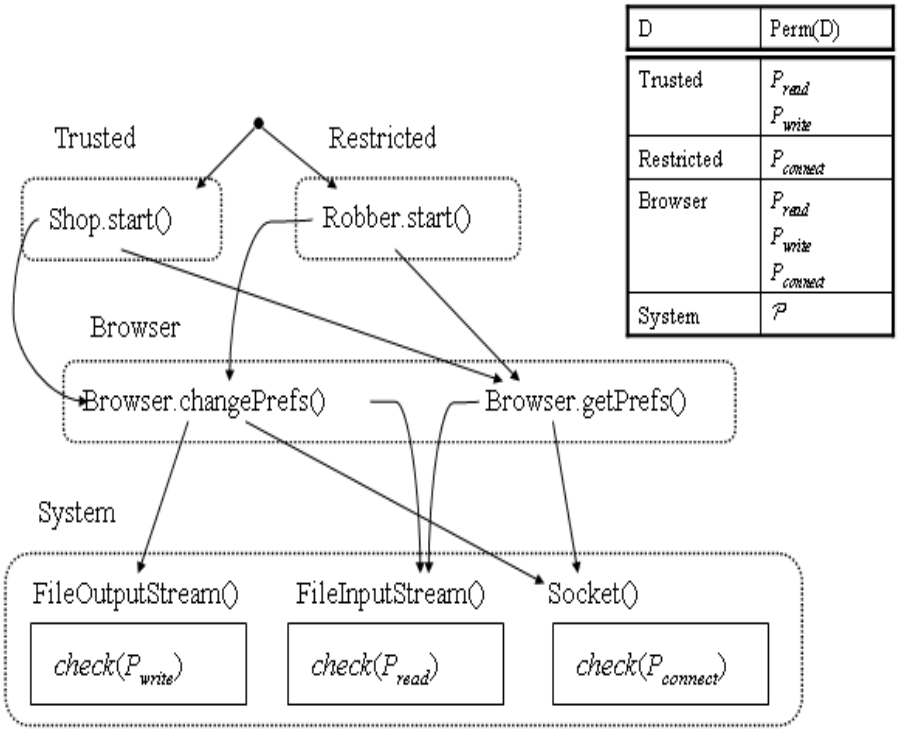

Fig. 2. Call graph and security policy for e-commerce application (client-side)

A permission check *must fail* at the entry to a method n, if it is not a may-succeed check at n. If a permission check $check(p)$ in a method m must fail at the entry to a method n, it implies that there is no path from n to m, which can grant the permission p. If a starting method(or a privileged action method) is started, its must-fail checks will certainly fail stack inspection when they are executed.

Once may-succeed checks $may - sc_{entry}(n)$ at a method n have been computed, then must-fail checks $must - fc_{entry}(n)$ at the method n can be simply computed as:

$$must - fc_{entry}(n) = rc(n) - may - sc_{entry}(n)$$

where $rc(n)$ is the set of all reachable permission checks to a node(method) n without considering permissions. $rc : N \rightarrow 2^{Check}$ is the least solution of the following equation:

$$RC(n) = \begin{cases} \{check(p) | check(p) \in n\} & \text{if } n \text{ is final} \\ \bigcup \{RC(m) | n \rightarrow m \in E\} \cup \{check(p) | check(p) \in n\} & \text{otherwise} \end{cases}$$

Method	may-succeed checks
`Shop.start()`	$\{check(Pread), check(Pwrite)\}$
`Robber.start()`	$\{check(Pconnect)\}$
`Brower.chagePrefs()`	$\{check(Pread), check(Pwrite), check(Pconnect)\}$
`Browser.getPrefs()`	$\{check(Pread), check(Pconnect)\}$
`FileOutputStream()`	$\{check(Pwrite)\}$
`FileInputStream()`	$\{check(Pread)\}$
`Socket()`	$\{check(Pconnect)\}$

Fig. 3. May-succeed checks

Note that only normal calls denoted by $n \rightarrow m$ are considered when computing reachable checks. A privileged call $n \rightsquigarrow n'$ is not considered, since stack inspection cannot go further across a privileged call.

In the example, all the three checks are reachable to the `Shop.start()` and `Robber.start()` methods. So $check(Pconnect)$ must fail at `Shop.start()`, and $check(Pread)$ and $check(Pwrite)$ must fail at `Robber.start()`. Therefore if the applet starts from `Shop.start()`, then $check(Pconnect)$ must fail, and if the applet starts from `Robber.start()`, then $check(Pread)$ and $check(Pwrite)$ must fail.

Once a starting method(or a privileged action method) is executed, then its must-fail checks certainly fail stack inspection and throw `AccessControlException` when they are executed. This is because there is no backward path from the check to the starting method(or the privileged action method) such that stack inspection can succeed.

Our second analysis is called *Must-Succeed Check Analysis*, which gives a safe approximation of permission checks which must pass stack inspection.

Definition 3. *A* $check(p)$ *in a method* m must succeed *at the entry to a method* n*, if, for every path from the method* n *to the method* m *in the call graph, the permission* p *is granted by all the methods in the path.*

The *Must-Succeed Check Analysis* will determine:

for each node(method), which permission checks *must succeed* at the entry to the node.

Once a starting method(or a privileged action method) is started, then its must-succeed checks must pass stack inspection when they are executed. This is because the permission p is granted for every (backward) path from the checks to the starting method(or the privileged action method).

If a reachable check is not a must-succeed check at a node, then it *may fail* through some path from the check to the node. We first define *May-Fail Check Analysis* and then compute the must-succeed checks for each node n by

computing the complement of may-fail checks with respect to reachable checks. The *May-Fail Check Analysis* will determine:

for each node, which permission checks *may fail* through a backward path from the checks to the node.

The May-Fail Check Analysis is defined by the flow equations in Figure 4, where $May-FC_{entry}(n)$ includes all the may-fail checks in $May-FC_{exit}(n)$ and new may-fail $check(p)$'s in $rc(n)$ such that the permission p is not granted by the method n. Note that if $check(p)$ occurs in n, then it is simply included in $rc(n)$. If a permission check may fail at the entry to a node n, it means that there exists a path from n to the check, which doesn't satisfy the permission.

$$May - FC_{exit}(n) = \begin{cases} \emptyset & \text{if } n \text{ is final} \\ \bigcup\{May - FC_{entry}(m) | n \rightarrow m \in E\} & \text{otherwise} \end{cases}$$

$$May - FC_{entry}(n) = May - FC_{exit}(n) \cup gen_{May-FC}(n)$$

where $gen_{May-FC}(n) = \{check(p) \in rc(n) | p \notin Permission(n)\}$

Fig. 4. Flow equation of May-Fail Check Analysis

The flow equation in Figure 4 defines a transfer function $\mathcal{F}_{May-FC} : \mathcal{L} \rightarrow \mathcal{L}$. The least solution $(may - fc_{entry}, may - fc_{exit}) \in \mathcal{L}$ of the flow equation can be computed by $lfp(\mathcal{F}_{FC})$ in finite time. See [6] for details.

A permission check $check(p)$ in the least solution $may - fc_{entry}(n)$ means there exists a path from n to the check, which doesn't satisfy the permission. We can prove the soundness of the May-Fail Check Analysis. See [6] for details.

Theorem 2. *For every actual normal call chain from a method n to a method m which contains $check(p)$, if the permission p is not granted by some method in the call chain, then $check(p)$ is in $may - fc_{entry}(n)$.*

In the example, $check(Pconnect)$ is a may-fail check at the entry to `Shop.start()` and $check(Pread)$ and $check(Pwrite)$ are may-fail checks at the entry to `Robber.start()`.

Once the least fixpoint $may - fc_{entry}$ has been computed, the must-succeed checks $must - sc_{entry}$ at each node n can be computed by $must - sc_{entry}(n) = rc(n) - may - fc_{entry}(n)$ for each node n. If a starting method(or a privileged action method) n is started, its permission checks in $must - sc_{entry}(n)$ must pass stack inspection when they are executed, because all paths from n to the checks satisfy the permission.

For example, $check(Pconnect)$ is a must-succeed check at `Robber.start()` and $check(Pread)$ and $check(Pwrite)$ are must-succeed checks at `Shop.start()`.

So, if the applet starts from `Robber.start()`, then $check(Pconnect)$ must pass stack inspection.

The fixpoint can be computed by worklist algorithm [12]. Basic operations in the worklist algorithm are set operations like union and membership. The worklist algorithm needs at most $O(|E| \cdot |Check|)$ basic operations where $|Check|$ is the number of checks and the height of the lattice 2^{Check} [12].

4 Implementation

We implement the visualization system for permission checks and Java stack inspection based on the static permission check analysis information.

We first implement the permission check analysis in Java based on Barat, which is a front-end for Java compiler. Barat builds an abstract syntax tree for an input Java program and enriches it with type and name analysis information. We can traverse AST nodes and do some actions or operations at visiting each node using a visitor, which is a tree traverse routine based on design patterns.

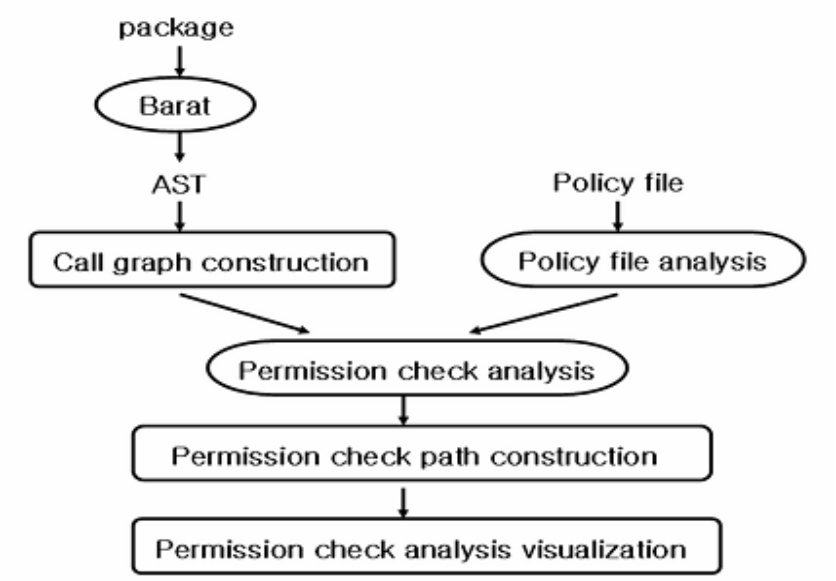

Fig. 5. System architecture

As in Figure 5, the permission check visualization system consists of five parts.

1. Policy file analysis collects, from input policy file, granted permission set for protection domain, which each method belongs to. This information is used to determine whether each permission check succeeds or fails.
2. Call graph construction constructs a call graph, where each method's callers are represented.
3. Permission check analysis computes permission checks which must succeed or fail at each method, based on policy file and call graph information.
4. Permission check path construction collects reverse call chains of permission checks from the call graph to trace stack inspection.
5. Visualization of permission check analysis visualizes permission checks and their stack inspection based on the static analysis information.

We extend Jipe, which is an open source IDE for Java to include the visualization system for permission check. Figure 6 shows the window executing Jipe and we can start the visualization system by selecting PermissionCheck browser menu item of Tools.

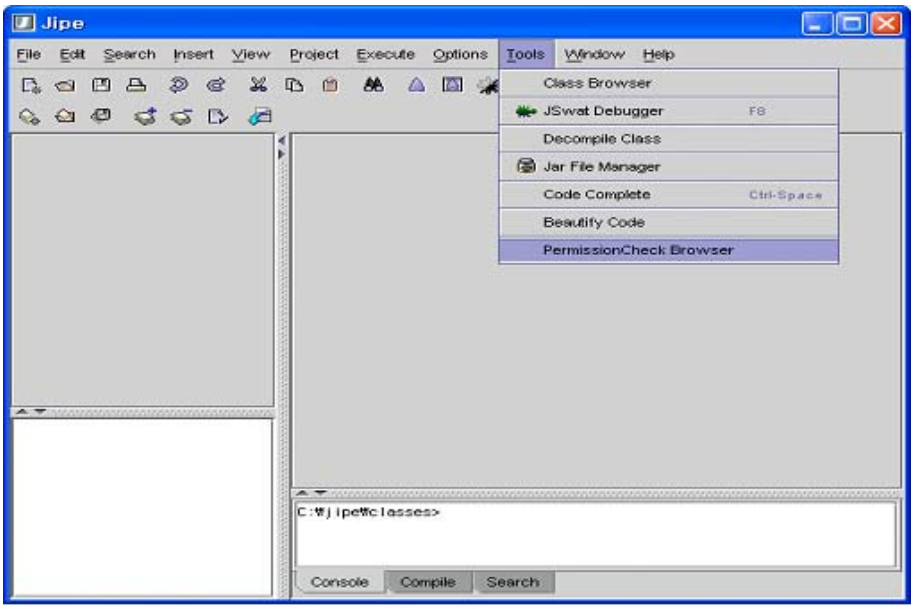

Fig. 6. Jipe and a menu for the Permission Check Visualization System

Figure 7 is a window, which shows permission checks visually based on static analysis information. The window shows a program `CountMain`, which creates `SecurityManager` class object, set it on system and counts characters in two files. Each numbered part is as follows:

In the part 1, users can select a display option. Information about permission check can be displayed in terms of methods or permission checks. The part 2 lists classes, methods and permission checks within a selected package. Users can select a method or a permission check of interest, depending on the option in the part 1. The part 3 displays contents of a selected policy file.

As in Figure 7, if a method is selected in the part 2, the part 4 displays all permission checks, which must fail, must succeed, or just may reach at the selected method. In the part 5, users can trace stack inspection procedure visually by following a calling chain from the selected method to a permission check. If a permission check must succeed at a method, it is displayed as green color. If a permission check must fail at a method, it is displayed as red color. Otherwise, it is displayed as yellow color.

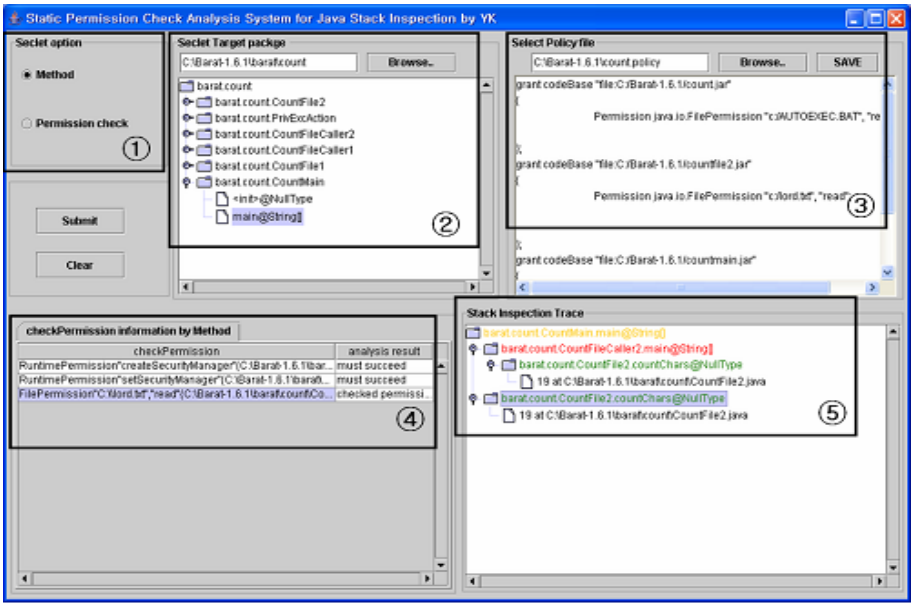

Fig. 7. Visualization of Permission Check Analysis: Method

As in Figure 8, if a permission check is selected in the part 2, the part 4 displays all reachable methods with analysis information that it must succeed, or must fail at each method. In the part 5, users can trace stack inspection procedure visually by following a calling chain backward from the selected permission check.

After examining permission check and stack inspection, programmers can modify policy files and programs if necessary. This process can be repeated until the security policy is enforced correctly.

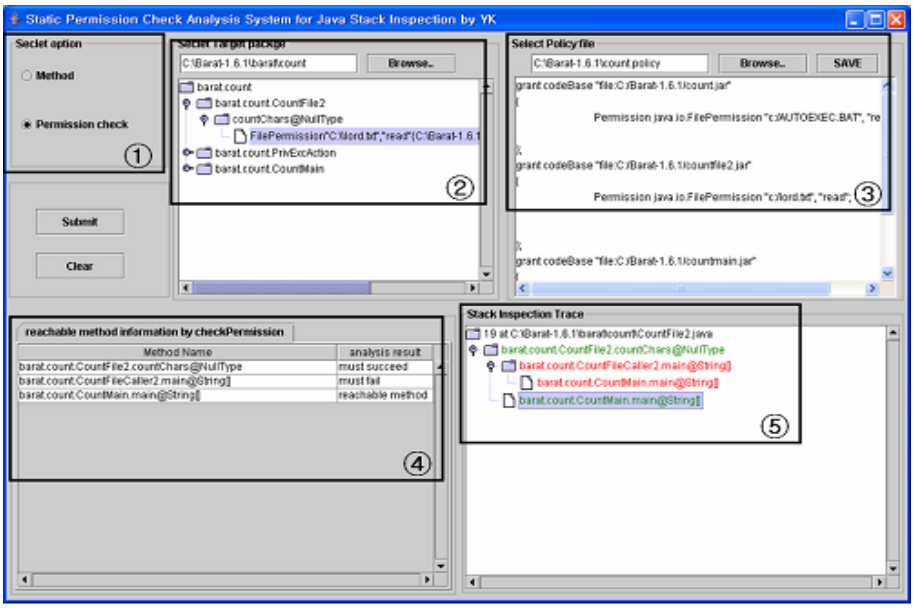

Fig. 8. Visualization of Permission Check Analysis: Permission Check

We experiment the system with five Java packages. The first program is `CountMain`. The second program `BankSystem` access to file which has client and account information, read and write new information. The third program `StringSearch` counts number of times which "string" appears in ten files. The fourth program `getProps` reads system information which are about system user, operating system, and file system. The last Client-Server is a client-server program, where server and client create socket, server sends the message which server reads from the local file and then client saves the received message from the server.

Table 1 shows the experimental results. The results can change according to the policy file. It shows the number of all checks within package, the number of permission checks which may reach to the main method, the number of must-fail or must-succeed checks at the main method. In case of `CountMain`, 1 check must fail and two checks must succeed at the main. There are no must-succeed or must-fail checks in `BankSystem`. In case of `StringSearch`, 5 checks must succeed and 5 checks must fail at the main. In `getProps`, 2 checks out of 9 must fail.

5 Related Works

There are some works on stack inspection such as semantics, type system, static analysis and implementation.

Package	All checks	Must-fail checks	Must-succeed checks	Reachable checks
CountMain	5	1	2	3
BankSystem	6	0	0	4
StringSearch	10	5	5	10
getProps	9	2	0	7
Server-Client	3	1	0	1

Table 1. Experimental result

Wallach et al. [16] present a new semantics for stack inspection based on a belief logic and its implementation using the calculus of security-passing style which addresses the concerns of traditional stack inspection. With security-passing style, the security context can be efficiently represented for any method activation, and a prototype implementation is built by rewriting the Java bytecodes before they are loaded by the system.

Pottier et al. [14] address static security-aware type systems which can statically guarantee the success of permission checks. They use the general framework, and construct several constraint- and unification-based type systems. They offer significant improvements on a previous type system for JDK access control, both in terms of expressiveness and in terms of readability of inferred type specifications.

Erlingson [7] describes how IRMs(Inlined Reference Monitor) can provide an alternative to enforcing access control on runtime platforms, like the JVM, without requiring changes to the platform. Two IRM implementations of stack inspection are discussed. One is a reformulation of security passing style proposed in [16]; the other is new and exhibits performance competitive with existing commercial JVM-resident implementations.

Walker [17] uses security automata to express security policies. Security automata can specify an expressive collection of security policies including access control and resource bounds. They describe how to instrument well-typed programs with security checks and typing annotations. The resulting programs obey the policies specified by security automata and can be mechanically checked for safety. This work provides a foundation for the process of automatically generating certified code for expressive security policies.

There are several static analysis techniques for permission checks [4, 5, 1, 2, 11].

Bartolleti et al. proposed two control flow analyses for the Java bytecode [1]. They safely approximate the set of permissions granted/denied to code at run-time. This static information helps optimizing the implementation of the stack inspection algorithm. They also developed a technique to perform program transformation in the presence of stack inspection [2]. This technique relies on the trace permission analysis, which is a control flow analysis and compute a safe

approximation to the set of permissions that are always granted to bytecode at run time.

Koved et al. [11] presents a technique for computing the access rights requirements by using a context sensitive, flow sensitive, interprocedural data flow analysis. This analysis computes at each program point the set of access rights required by the code. They implemented the algorithms and present the results of the analysis on a set of programs. This access rights analysis is also implemented into SWORD4J, which is a collection of tools for Java static analysis, and is available for the popular Eclipse IDE.

Besson et al applied constraint-based static analysis techniques to the verification of global security properties [5]. They introduces a formalism based on a linear-time temporal logic for specifying global security properties pertaining to the control flow of the program.

Most static analyses approximate stack inspection in terms of permissions [4, 5, 1, 2, 11]. Our proposed analysis is unique in that it compute success or fail information in terms of *permission checks*. This static analysis can approximate information about stack inspection for permission check. We also implemented visualization of stack inspection based on the static analysis information, which can help programmers examine stack inspection easily.

6 Conclusions

We have developed a visualization tool which helps programmers enforce security policy effectively into programs, based on the static permission check analysis. Using the visualization system, programmers can modify programs and policy files if necessary, as they examine how permission checks and their stack inspection are performed. This process can be repeated until the security policy is enforced correctly.

The static analysis information can also be applied to optimizing redundant permission checks. For example, stack inspection of a permission check can be skipped if it must succeed.

References

1. M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. *Electr. Notes Theor. Comput. Sci. 54*, 2001.
2. M. Bartoletti, P. Degano, G. L. Ferrari. Stack inspection and secure program transformations. *Int. Journal of Information Security*, Vol.2, pp. 187-217, 2004.
3. F. Besson, T. Blanc, C. Fournet, A. D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. *CSFW 2004*.
4. F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspsection. In *Proc. 4th Conference on Principles and Practice of Declarative Programming*. ACM Press, New York, 2002.

5. F. Besson, T. Jensen, D. Le Metayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security 9*, pp. 217-250. 2001.
6. B.-M. Chang, Static Check Analysis for Java Stack Inspection, *ACM SIGPLAN Notices* Vol. 41 No. 2, Feburary 2006.
7. U. Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. *2000 IEEE Symposium on Security and Privacy*, pp. 246-255.
8. C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. & Syst.* 25(3): 360-399 (2003)
9. J. Gosling, Joy, Steele, The Java Language Specification Second Edition, Addison-Wesley, 2002
10. D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. *ACM OOPSLA* 1997, pp. 108-124.
11. L. Koved, M. Pistoia, A. Kershenbaum. Access rights analysis for Java. *ACM OOPSLA 2002*, pp. 359-372
12. F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999.
13. N. Nitta, Y. Takata, H. Seki. An efficient security verification method for programs with stack inspection. *2001 ACM Conference on Computer and Communications Security*, pp. 68-77.
14. F. Pottier, C. Skalka, S. F. Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. & Syst.* 27(2), pp. 344-382, 2005.
15. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM OOPSLA 2000*, pp 281-293.
16. Dan S. Wallach, Andrew W. Appel, Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.* 9(4), pp. 341-378, 2000.
17. Lujo Bauer, Jay Ligatti and David Walker. Composing Security Policies in Polymer. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2005.
18. http://java.sun.com/j2se/1.5.0/docs/api.