Static Check Analysis for Java Stack Inspection

Byeong-Mo Chang

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

Abstract. Most static analysis techniques for optimizing stack inspection approximate *permission sets* such as granted permissions and denied permissions. Because they compute permission sets following control flow, they usually take intra-procedural control flow into consideration as well as call relationship. In this paper, we observed that it is necessary for more precise optimization on stack inspection to compute more specific information on *checks* instead of permissions. We propose a backward static analysis based on simple call graph to approximate redundant permission checks which must fail. In a similar way, we also propose a backward static analysis to approximate success permission checks, which must pass stack inspection.

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 [7, 16].

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 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.

Stack inspection may be expensive, since it must examine all the frames in the call stack. The run-time overhead due to examining stack frames may grow very high. Hence, effective improvement and optimization of stack inspection is a good research challenge.

There have been several works to optimize stack inspection by eliminating redundant checks with static analysis information [4, 12, 1, 2, 10]. Most of them approximate *permission sets* such as granted permissions and denied permissions [4, 12, 1, 2, 10]. Because they compute permission sets following control flow, they usually take intra-procedural control flow into consideration as well as call relationship [12, 1, 2].

In this paper, we observed that it is necessary for more precise optimization on stack inspection to compute more specific information on *checks* instead of permissions. We propose a backward static analysis to determine redundant permission checks. The static analysis is performed based on simple call graph. To determine redundant (dead) checks, we first approximate all live checks at every method by static analysis. A permission check $check(p)$ in a method m is *live* at a method n, if there exists a calling chain from the method n to the method m, along which the permission p is granted. We call it live at the method n because the stack inspection can go further across the method. Once live checks are computed at each method, dead checks (which are not live) can be computed easily at each method. Permission checks which

are dead at a starting method or a privileged action method must fail, because there exists no path, along which the permission is granted. So dead checks are redundant and can be eliminated. This idea can also be extended to other methods, even if they are not starting methods. Once a method is called during execution, its dead checks will certainly fail when they are executed. So, we can transform or optimize a program so as to utilize this property.

In a similar wary, we can also determine success permission checks, which must pass stack inspection. We also propose a backward static analysis to approximate success checks.

This paper is organized as follows. The next section reviews Java 2's stack inspection. Section 3 describes two proposed static analyses. Section 4 discusses applications of the static analyses. 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. An `AccessControlException` is thrown, if denied. 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 the `checkPermission(p)` 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. A method `M` performs a privileged action `A` by invoking `AccessController.doPrivileged(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 [16] :

```
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 Analysis

We define our static analysis 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 [14, 9]. 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 is *live* 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. We call it live because the stack inspection can go further across m.

We will approximate all live checks at every method by static analysis. Then we compute dead checks, which are not live. A live permission check is defined as follows:

Definition 2. *A permission check $check(p)$ in a method m is* live *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 *Live Check Analysis*, which gives a safe approximation of live checks at each method. The *Live Check Analysis* will determine:

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

The live check analysis is defined by the flow equation in Figure 1, where $LC_{entry}(n)$ includes $check(p)$'s in the method n or in $LC_{exit}(n)$ such that the permission p is granted by the method n. Note that only normal calls denoted by $n \rightarrow m$ are considered in the equation $LC_{exit}(n)$.

The flow equation in Figure 1 defines a transfer function $\mathcal{F}_{LC} : \mathcal{L} \rightarrow \mathcal{L}$, 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}. The least solution $(lc_{entry}, lc_{exit}) \in \mathcal{L}$ of the flow equation in Figure 1 can be computed by $lfp(\mathcal{F}_{LC})$ in finite time as in Theorem 1.

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

$$LC_{entry}(n) = \{check(p) | check(p) \in LC_{exit}(n), p \in Permissions(n)\} \cup gen_{LC}(n)$$

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

Fig. 1. Flow equation for live check analysis

Theorem 1. *The least fixpoint $lfp(\mathcal{F}_{LC})$ for the live check analysis can be computed in finite time.*
PROOF. Because the set *Check* of permission checks in a program is finite, the property space $\mathcal{L}$ is finite and so it satisfies the ascending chain condition. The transfer function $\mathcal{F}_{LC}$ is monotonic. So, the least solution can be computed by $lfp(\mathcal{F}_{LC}) = \mathcal{F}^n_{LC}(\perp)$ for some finite n by Tarksi theorem.

We prove the soundness of the live check 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.

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 granted by all the methods in the call chain, then $check(p)$ is in $lc_{entry}(n)$.*
PROOF. By the soundness of call graph construction, for every actual normal call chain, there exists a corresponding path in the call graph. If the permission p is granted by all the methods in the call chain, then it must be granted by all the methods in the path in the call graph. So, the check $check(p)$ is introduced into the solution $lc_{entry}(m)$ by the flow equation. It is propagated to the method n, and is included in the solution $lc_{entry}(n)$ by the equation, because it is granted by all the methods in the path.

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 live check analysis computes live checks for the entry of each method, which are shown in Figure 3. Note that $check(Pread)$ and $check(Pwrite)$ are live at `Shop.start()`, and $check(Pconnect)$ is live at `Robber.start()`.

A permission check is called *dead* at the entry to a method n, if it is not live at n. If a permission check $check(p)$ in a method m is dead 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 dead checks will certainly fail stack inspection when they are executed.

Once live checks $lc(n)$ at a method n have been computed, then dead checks $dc(n)$ at the method n can be simply computed as:

$$dc(n) = reachable(n) - lc_{entry}(n)$$

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

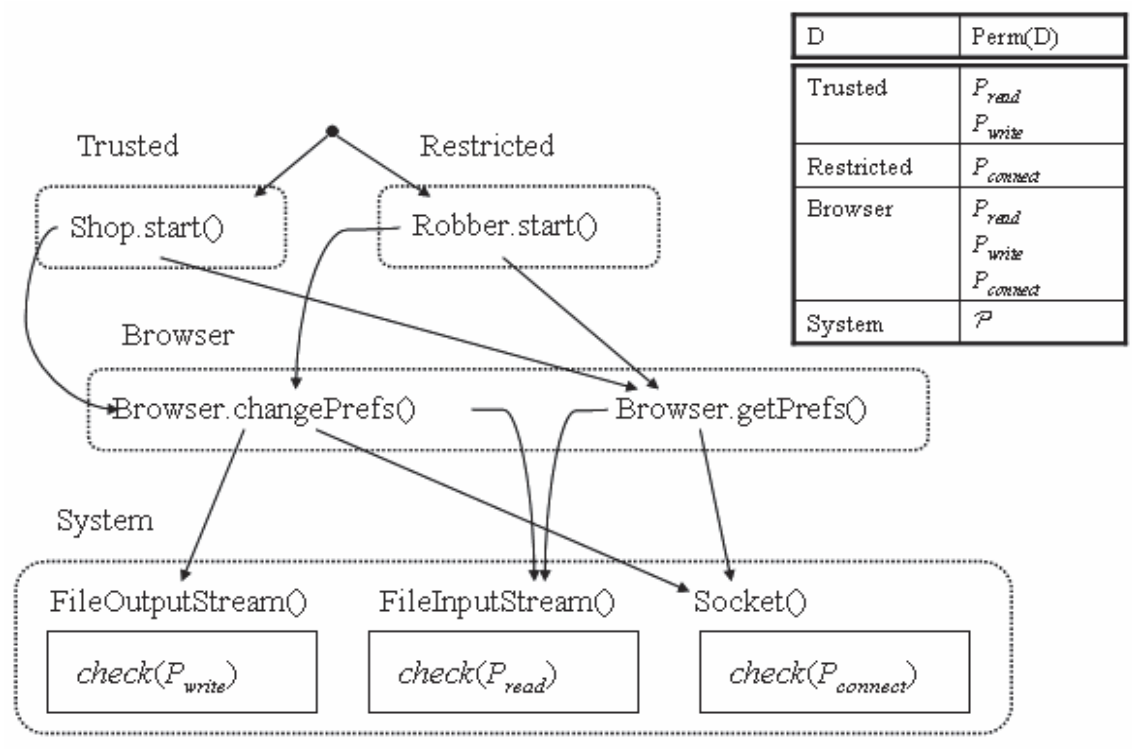

Fig. 2. Call graph and security policy for e-commerce application (client-side)

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

Note that only normal calls denoted by $n \to m$ 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 $check(Pconnect)$ is dead at `Shop.start()` and $check(Pread)$ and $check(Pwrite)$ are dead 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 started, then its dead checks must 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

Method	Live 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. Live checks

can succeed. Optimization of redundant checks based on the static analysis information is to be discussed in the next section.

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

Definition 3. *A check(p) in a method m is* successful *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 *Success Check Analysis* will determine:

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

Once a starting method(or a privileged action method) is started, then its success 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 successful at a node, then it *may fail* through some path from the check to the node. We first define *Failable Check Analysis* and then compute the success checks for each node n by the complement of failable checks with respect to $reachable(n)$. The *Failable Check Analysis* will determine:

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

The failable check analysis is defined by the flow equations in Figure 2, where $FC_{entry}(n)$ includes all the failable checks in $FC_{exit}(n)$ and new failable $check(p)$'s in $reachable(n)$ 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 $reachable(n)$. If a permission check is failable 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.

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

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

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

Fig. 4. Flow equation of failable check analysis

The flow equation in Figure 2 defines a transfer function $\mathcal{F}_{FC} : \mathcal{L} \to \mathcal{L}$. The least solution $(fc_{entry}, fc_{exit}) \in \mathcal{L}$ of the flow equation can be computed by $lfp(\mathcal{F}_{FC})$ in finite time as in Theorem 3.

Theorem 3. *The least fixpoint $lfp(\mathcal{F}_{FC})$ for the failable check analysis can be computed in finite time.* Proof. Because the finite property space $\mathcal{L}$ satisfies the ascending chain condition and the transfer function is monotonic, $lfp(\mathcal{F}_{FC})$ can be computed by $\mathcal{F}^n_{FC}(\bot)$ for some finite n.

A permission check $check(p)$ in the least solution $fc_{entry}(n)$ means there exists a path from n to the check, which doesn't satisfy the permission. We can prove the soundness of the failable check analysis.

Theorem 4. *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 $fc_{entry}(n)$.*

Proof. By the soundness of call graph, for every actual normal call chain, there exists a corresponding path in the call graph. If the permission p is not granted by some method k in the call chain, it must not

be granted by the method in the corresponding path in the call graph. So the check $check(p)$ is introduced into the solution $fc_{entry}(k)$ by the flow equation. Once it is introduced, it is propagated to n along the path, and is included in $fc_{entry}(n)$ by the flow equation.

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

Once the least fixpoint fc has been computed, the success checks sc at each node n can be computed by $sc_{entry}(n) = reachable(n) - fc_{entry}(n)$ for each node n. If a starting method(or a privileged action method) n is started, its success checks in $sc_{entry}(n)$ 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 success check at `Robber.start()` and $check(Pread)$ and $check(Pwrite)$ are success 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 algorith [11]. 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} [11].

4 Applications

In the example, we detect by the static analysis that 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.

Based on the static analysis information, the dead checks can be eliminated and replaced by the code throwing `AccessControlException`. This optimization can be done based on a starting point, which is an end point of stack inspection. There are several starting points with respect to stack inspection in Java programs. A `main` or `start` method are starting methods in case of Java applications or applets respectively. A privileged action method can also be considered as a starting point because stack inspection ends there.

If a starting method(or a privileged action method) is started, then its dead checks must fail stack inspection and throw `AccessControlException`. This is because there is no backward path from the check to the starting method(or the privileged action method), which can pass stack inspection. Therefore, permission checks which are dead at a starting method(or a privileged action method) are redundant and can be replaced by the code throwing `AccessControlException`.

This optimization can be justified as follows. Because we don't consider intra-procedural control flow, the check may or may not be executed. When the check is not executed, it is a kind of dead code, and so the elimination causes no problems. When the check is executed, it must fail, and so can be replaced by the code throwing `AccessControlException`.

This idea can also be extended to other methods, even if they are not starting methods. If a method is called during execution, its dead checks will certainly fail when they are executed. So, we can transform or optimize a program so that if a method is called, its dead checks can be skipped and replaced by the code throwing `AccessControlException`.

In a similar way, success check information can also be utilized for optimizing redundant permission checks. For example, $check(Pread)$ and $check(Pwrite)$ are success checks at `Shop.start()`. So, if the applet starts from `Shop.start()`, then $check(Pread)$ and $check(Pwrite)$ must pass stack inspection, so those checks can be skipped.

5 Related Works

There are several static analysis techniques for permission checks [4, 5, 1, 2, 10]. Most static analyses approximate stack inspection in terms of permissions. Our proposed analysis is unique in that it compute dead or

success information in terms of checks. By this type of analysis, we can determine whether each check is dead or not at each method. This can give more specific information for optimization of redundant checks. Every method has specific information about dead checks and partially context-sensitive optimizations can be done depending on method calls using this information.

There are some works on static analysis of security checks like stack inspection. 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. [10] 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.

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. They defined a security-dedicated program model that only contains procedure call and run-time security checks and propose a model checking method for verifying that an implementation using local security checks satisfies a global security property. In [4], they also presented a technique for inferring a secure calling context for a method, which is a pre-condition on the call stack sufficient for guaranteeing that execution of the method will not violate a given global property.

There are some other works on stack inspection such as semantics, type system and implementation. Wallach et al. [15] 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. [13] 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 [6] 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 [15]; the other is new and exhibits performance competitive with existing commercial JVM-resident implementations.

6 Conclusion

We have proposed two static analysis techniques, which can give more specific information for optimization of redundant checks. We have also discussed optimization of redundant checks using the static analysis information. In particular, every method has specific information about dead checks. This information can allow us to do partially context-sensitive optimizations depending on method calls.

We are currently implementing the static analysis, and transformation-based optimizations using static analysis information. We will also extend the proposed static analysis for fully context-sensitive analysis.

Acknowledgements

This Research was supported by the Sookmyung Women's University Research Grants 2005. The author would like to thank Yoonkyung Kim for useful discussions and helping to prepare this document.

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. U. Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. *2000 IEEE Symposium on Security and Privacy*, pp. 246-255.
7. C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. & Syst.* 25(3): 360-399 (2003)
8. J. Gosling, Joy, Steele, The Java Language Specification Second Edition, Addison-Wesley, 2002
9. D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. ACM OOPSLA 1997, pp. 108-124.
10. L. Koved, M. Pistoia, A. Kershenbaum. Access rights analysis for Java. *OOPSLA 2002*, pp. 359-372
11. F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999.
12. 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.
13. 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.
14. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. ACM OOPSLA 2000, pp 281-293.
15. 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.
16. http://java.sun.com/j2se/1.5.0/docs/api.