Type Annotations to Improve Stack-based Access Control

Tian Zhao
University of Wisconsin–Milwaukee
tzhao@cs.uwm.edu

John Boyland
University of Wisconsin–Milwaukee
boyland@cs.uwm.edu

Abstract

Java security architecture uses stack-based access control to protect security-sensitive resources. The architecture implements access control checks by inspecting the call stack to compute permission levels, which are used to decide whether to grant access to these resources. This implementation only considers the direct and indirect callers of sensitive methods that directly accesses the resources. However, it does not check the integrity of the variables used in the calls to these methods, nor does it help protect confidential values that might be returned by these calls. This paper proposes a type-based approach to strengthen stack-based access control. We use type annotations to track values originated from untrusted code such that these values will not be inputs to the sensitive methods when they are executed with high level of trust. We also use the annotations to protect confidential values from being accidentally revealed by trusted code. We give a static type system that checks these properties and augments existing dynamic stack-based checks. The hybrid approach is similar to so-called "history-based access control" without its run-time burden.

1. Introduction

In extensible systems such as Java Virtual Machine (JVM) and Common Language Runtime (CLR), trusted and untrusted components coexist and the complex interaction between these components make it difficult to protect security-sensitive data and resources. In these systems, programmers can use access control checks based on stack inspection to establish protection boundaries between the system routines that directly access the resources and the client code that calls them. The stack inspection mechanism includes configurable security policies to associate static permissions with each codebase based on its location and/or runtime authentications. It also associates runtime code with dynamic permissions obtained as a function of the stack [10, 21]. By default, the stack inspection algorithm scans the runtime call stack from the most to least recent stack frame and checks whether the codebase associated with each frame has the required permissions. The algorithm stops and raises security exception if it encounters a frame without the required permissions and it succeeds otherwise. Without necessary permissions, untrusted code cannot gain access to sensitive resources by directly or indirectly calling the system routines because the untrusted code can be identified in the call stack and the access will be denied at runtime.

However, stack inspection only provide limited protection against security attacks. In particular, it does not consider all the interactions between software components such as values exchanged between trusted and untrusted code, mutable states, side effects, inheritance, etc. [9]. For example, suppose there exists a native method that can accept a file object as argument and then delete the file specified by the path in the object. A piece of library code that provides public access to this method inserts an access control check before invoking it. Stack inspection can ensure direct and indirect callers of the native method have the required permissions but it cannot guarantee that the argument to the method is created by code with the necessary permissions. Therefore, if a thread consisting of all trusted code attempts to delete a file, it should not supply an argument variable that may be "tainted" by code without necessary permissions. Similarly, if trusted code obtains some secret values based on its own rights, it must ensure such values are not revealed to code without the required read permissions.

Past research in trust analysis [15, 17] statically tracks untrusted values by security labels so that they do not flow to dangerous operations without going through requisite checks. Also, extensive work on information flow control [8, 7, 19, 20, 12, 14, 2, 3] studied how to statically trace known confidential values so that they do not interfere with low security-level output. However, these approaches cannot be directly applied to the aforementioned problem because not all operations guarded by access control checks require only trusted arguments and not all their results are confidential. One alternative model to stack inspection (Abadi and Fournet "history-based access control" [1]) relies on execution history for access control decisions and it can prevent the problems listed above but it requires changes to the current access control algorithms in JVM.

In this paper, we motivate and describe a method of using type annotations to statically track values by their origin so that they can only be used in appropriate security contexts. For instance, values constructed by untrusted code may not be used for accessing resources when the dynamic context shows no trace of the untrusted code. In addition, we also use type annotations to protect possible confidential values obtained with the access rights of trusted code. This method can help strengthen the protection of the stack inspection mechanism without modifying the runtime system. We introduce a simple type system that can applied modularly to trusted code and we depend on Java's enforcement of visibility modifiers of the fields and methods of a class. The result is such that if a trusted program is typable in our system, then it can safely interact with untrusted code. For simplicity, we do not model implicit information flow; we only control direct flow of information between trusted and untrusted code.

The rest of the paper is as follows. In Section 2, we give two motivating examples and in Section 3, we informally describe our ideas of program annotations. In Section 4, we formalize the method with a simple language extended from Featherweight Java [13]. Section 5 discusses the limitations of our system, and related work is reviewed in Section 6. The paper concludes with Section 7.

2. Examples

The examples in this section illustrate the potential problems of using stack inspection for security protection. They are inspired by examples in earlier work [1, 4].

We assume two kinds of classes: trusted classes in the packages `system` and `local`, and the untrusted classes in the package `user`. The trusted classes have permission to access resources including all local files, while the untrusted classes only have limited permissions such as access to files in temporary directories. Suppose that the trusted class `FileSystem` implements direct access to the file system, while the `File` class provides interface to (untrusted) clients that needs to manipulate files (shown in Figure 1). The methods of the `File` class check the permissions of the callers before granting or denying the access to the file system. Untrusted code is prevented from directly accessing the file system since its methods are only package-accessible. (Alternatively, the class itself could be made package-accessible only.)

If any code in `user` package attempts to delete or open the password file by calling the delete or open method of the file object `new File("/etc/passwd")` then such access will be denied since the caller to the delete method of File class doesn't have the required permission to delete the password file.

```
package system;

public class FileSystem {
    void delete (File file) {     /* implement delete */     }
    FileHandle open (File file) {     /* implement open */  }
}

public class File {
    String path;
    public File (String path) { this.path = path; }

    public void delete () {
        AccessController.checkPermissions(
                    new FilePermission(path, "delete"));
        new FileSystem().delete(this);
    }
    public FileHandle open () {
            /* check permission before opening file */     }
}
```

Figure 1. Protecting file system access.

Figure 2 and 3, show two security flaws of the class `local.Task` that can be exploited by untrusted code in package `user`. The first flaw is to do with its method `cleanup`, which deletes a backup file specified by the method `getBackupFile`. In order to allow untrusted code to use the method `cleanup` to remove backup files, we use the `doPrivileged` method to assert the privilege of trusted code before calling the delete method on the file object. In Java, the access control algorithm scans the call stack starting from the top until the bottom or it reaches a frame marked as privileged. It returns with success if all frames it sees have the required permission.

However, since the method `getBackupFile` is not package-scoped, a method in the class `user.MyTask` can override this method and return the password file object instead. Consequently, the untrusted code in `user` package is able to delete the password file indirectly by invoking the method `MyTask.cleanup`.

The second flaw is caused by the method

```
package local;

public class Task {
    public File getBackupFile() {
            return new File ("/etc/backup");
    }
    public void cleanup() {
        AccessController.doPrivileged(...
                this.getBackupFile().delete(); ...);
    }

package user;

public class MyTask extends Task {
    public File getBackupFile () {
            return new File("/etc/passwd");     }
}

// the password file is deleted indirectly
new MyTask().cleanup();
```

Figure 2. Security defect that allows untrusted code to delete any file

```
package local;

public class Task {
    public void changePassword() {
        AccessController.doPrivileged(...
            this.updateFile(
                new File("/etc/passwd").open()); ...);
    }
    public void updateFile(FileHandle fh) { ... }

package user;

public class MyTask extends Task{
    public FileHandle fh;
    public void updateFile (FileHandle fh) { this.fh = fh; }
}

MyTask task = new MyTask();
task.changePassword();
// obtain the file handle to the password file indirectly
task.fh;
```

Figure 3. Security defect that permits untrusted code to open any file.

`Task.changePassword`, which also uses the `doPrivileged` method to assert the rights of trusted code to open the password file and then pass the (confidential) file-handle to the method `Task.updateFile`. Again since the method `updateFile` is not package-scoped, it is overridden by a method in `user.MyTask`, which stores the file handle in its field `fh`. We assume that access to the file-handle gives unfettered access to read the file, and thus untrusted code should not have access to the file handle for a security-relevant file. But in this case, the untrusted code can thus obtain the file handle to the password file by calling the method `MyTask.updatePassword`, and then retrieving the stored file-handle.

The examples demonstrate the insufficiency of stack inspection in that the decision to grant access to security-sensitive methods depends only on the direct and indirect callers but not on the integrity of the arguments to these methods. Moreover, stack inspection does not keep track of confidential values computed by security-sensitive code. Consequently, confidential values may accidentally be revealed to untrusted code.

We augment stack inspection by tracking the integrity and confidentiality levels of values via type annotations as described in the next section. Our analysis does not require a closed-world assumption and utilizes Java's accessibility rules to aid in the protection of trusted variables and confidential values. The two major limitations are (1) that, being static, the analysis will occasionally reject code that is actually safe in the way it handles values, and (2) that it does not prevent information flow via covert channels or indirect access. The first limitation, common to all static analysis techniques, can be viewed positively as a guideline for writing more secure code; the second limitation is beyond the scope of this paper in which we mainly focus on inadvertent errors in trusted code.

3. Tracking data dependencies

This section describes how we use type annotations to prevent the security defects shown earlier. We assume that each class is statically associated with some *static permissions* at compile-time/load-time according to security policies. For simplicity, we suppose that there are only two sets of permissions – high (H) and low(L) that are associated with trusted/untrusted classes respectively. We also assume that the classes in the same package have the same set of static permissions. At run-time, each execution unit is associated with some *dynamic permissions* at each point in time. The dynamic permissions are computed from the static permissions of the code in the current execution context. By default, the dynamic permissions are the intersection of the static permissions of all the callers on the call stack. If one of the callers is the `AccessController.doPrivileged` method. then the previous callers on the stack are ignored. The stack inspection mechanism makes access control decisions based on the dynamic permissions.

We associate each value (object or primitive) with a security level, which is the set of static permissions of the code where the value is created. We refer to the methods that provide direct access to security-sensitive resources as *sensitive methods*, to imply that calls to these methods should only be from contexts in which the dynamic permissions are checked.

3.1. Data integrity

As shown in Figure 2, calls to the sensitive method `FileSystem.delete` may not be secure if the security level of its argument (the file object) is low while the dynamic permission set of its execution context is high. Stack inspection does not give sufficient protection in this case because it only checks the dynamic permissions. Thus, we wish to maintain the invariant that the security level of the arguments to a sensitive method always include the permissions required for the operation. That is, the argument of the method `FileSystem.delete` should be created by code that has the permission to delete such a file. As an approximation, we require that the security level of the arguments to sensitive methods always contain the dynamic permissions. If the call to the sensitive method can proceed, then the invariant is apparently satisfied.

Methods are marked with the maximum security level in which they may be called. Using our two-level permission simplification, this means that if a method is marked L, it cannot be called if the dynamic permission set is possibly H. In other words, such a method may rely on stack-based access control. The reason why it may wish to is because it uses "tainted" value (created by untrusted code). The following fragment shows an example:

```
package system;

public class File {
...
    public void deleteMyBackup L () {
        File tmp = new MyTask().getBackupFile();
        AccessController.checkPermissions(
            new FilePermission(tmp.path, "delete"));
        (new FileSystem()).delete(tmp);
    }
}
```

Indeed the L annotation on this method is enforced by our type system because a "tainted" value is passed to a sensitive method, delete in class FileSystem. It is also enforced that this method cannot be invoked when the dynamic permission set could be H (inside of a "doPrivileged" call or a method *not* annotated L). In general, the security level of a method should not exceed the the dynamic permission set when the method is invoked.

We track the security levels of values by a security qualification α on each type t, written $t\langle\alpha\rangle$. This qualification represents the *exact* security level of the value; for simplicity we do not permit mixing high-security and low-security values. The reason for this restriction is that we use security levels to distinguish "tainted" from "untainted" values, as well as to distinguish "confidential" from "unconfidential" values. A value that could be either high or low security must be considered both possibly tainted (L) and possibly confidential (H). Rather than complicate our analysis here with two opposite lattice orderings, we chose an invariant order for values. Thus $t\langle\alpha\rangle \leq t'\langle\alpha'\rangle$ if and only if the type $t \leq t'$ and $\alpha = \alpha'$.

Instances of untrusted classes always have types qualified with L, while the security levels of the instances of trusted classes and primitive values can be either H or L depending on where they are created. Thus, a trusted class t is parameterized by a special variable X, which represents the security level of the instances of the class, which can also be used to label the types and methods used in the class. When an object of the class t is instantiated with security level α, then occurrences of the variable X in the class t are replaced by α.

The syntactic security level of values and methods thus ranges over H, L, and X. If a method is labeled with α, then we assume that the dynamic permission set when executing the method body is α. If a method m with security level X is called on a receiver object with security level α, then the security level of the method call is $[^{\alpha}/_{X}]X = \alpha$. For the purpose of method annotations, we define a partial order on H, L, and X such that $L \leq X \leq H$. When the dynamic permission set is α, a method call with security level α' is safe if and only if $\alpha \leq \alpha'$.

We wish to prevent security defects in trusted code such as system library and local applications so that they can safely interact with any untrusted code. We impose restrictions on all trusted code but we may not be able to verify that the untrusted code satisfies these restrictions. To avoid closed-world assumption and to check trusted code modularly, we rely on access modifiers of method and fields to protect variables and values of high security level from untrusted code. We use Java's package concept as the protection boundary for variables and values of high security level. That is, the types of public/protected fields and methods of a trusted class must not be qualified

H because these fields and methods may be accessible to untrusted code outside the defining package of the trusted class.[1] Packages of trusted code can be aware each other's the type annotations. A trusted class may only inherit from another trusted class. Also, a method of a trusted class with security level α may override another method with security level α' iff $\alpha' \leq \alpha$.

As shown below, we annotate the types and methods in the classes `system.File` and `local.Task` so that the security defect in Figure 2 can be detected.

```
package system;

public class File⟨X⟩ { . . .
    public void delete X () {
        // check file delete permission
        new FileSystem⟨X⟩ ().delete(this);
    }
}

package local;

public class Task⟨X⟩ {
    public File⟨X⟩ getBackupFile () {
        return new File⟨X⟩ ("/etc/backup");
    }
    public void cleanup () {
        AccessController.doPrivileged(...
            this.getBackupFile().delete(); ...);
    }
}
```

The method `File.delete` is labeled as X since it calls the sensitive method `FileSystem.delete` with argument of type `File`$\langle X\rangle$. The return type of the method `getBackupFile` is also labeled with X because the method is public and could be overridden by the method of untrusted subclass of `Task`. Thus, the security level of the method call `this.getBackupFile().delete()` is also X. However, the call is executed within the call to the `doPrivileged` method and the dynamic permission set is H and $H \not\leq X$. Thus, the method is not typable and should be rejected as unsafe. To fix this problem, we can let the method `getBackupFile` to be non-public and then we can annotate its return type with H.

3.2. Data secrecy

The defect shown in Figure 3 can be detected using qualified types as well. If a sensitive method is invoked when the dynamic permission set is H, then the value returned by the method can be high security value and if so, then it should be labeled with H. For example, the file handle of the password file returned by call to `FileSystem.open` should only be accessible to the trusted code, while the file handle of a temporary file can be accessed by any code. Thus, we should distinguish the two kinds of values by giving them types labeled with security levels H and L respectively. However, it is difficult to statically determine the security level of a value returned by sensitive method. As an approximation, we treat any results returned by a sensitive method when the dynamic permission set is H as confidential value and label its security level as H. The results returned when dynamic permission set is L can be safely labeled with L.

The code below shows how we can annotate the classes `system.File` and `local.Task` to protect confidential values.

```
package system;

public class File⟨X⟩ {
    // check permission before open file
    public FileHandle⟨X⟩ open () { . . . }
}

package local;

public class Task⟨X⟩ {
    public void changePassword () {
        AccessController.doPrivileged(...
            this.updateFile(
                new File⟨H⟩("/etc/passwd").open()); ...);
    }
    public void updateFile (FileHandle⟨X⟩ fh) { ... }
}
```

The return type of the method `File.open` is labeled with X because we expect the return value to have security level H or L depending the caller. Consider the second line in method `Task.changePassword`. The dynamic permission set of evaluating `new File`$\langle H\rangle$`("/etc/passwd").open()` is H because it is within a call to `doPrivileged` method. Thus, the security level of the file object must also be H because it is the same as that of the call to method `File.open`. Therefore, the return type of this call has security level H. However, the security level of the parameter type of the method `Task.updateFile` is X since the method is not package-scoped. Thus, the method `changePassword` is not typable.

4. Formalization

We formalize our ideas based on a simple language extended from Featherweight Java [13] and supple-

1 The requirement that a method annotated L may only be called when the dynamic permission level is L is automatically fulfilled when the call site is in untrusted code.

$$
\begin{array}{rcll}
\tau & ::= & t\langle\alpha\rangle & \text{qualified types} \\
d & ::= & \texttt{+} \mid \texttt{-} & \text{public/default access modifiers}
\end{array}
$$

$$
\begin{array}{rcll}
cls & ::= & \texttt{class}\ t\langle X\rangle \triangleleft \tau\ \{\ \overline{fd}; \overline{mth}\ \} & \text{classes} \\
fd & ::= & d\ \tau\ f & \text{fields} \\
mth & ::= & d\ \tau\ m^P_\alpha(\overline{\tau\ x})\ \{\,\texttt{ret}\, e;\ \} & \text{methods}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & x & \text{variable including "}this\text{"} \\
 & | & e.f & \text{field select} \\
 & | & e.f := e' & \text{field update} \\
 & | & e.m(\overline{e}) & \text{method call} \\
 & | & \texttt{new}\ \tau(\overline{e}) & \text{new object} \\
 & | & e; e' & \text{sequence} \\
 & | & \texttt{doPriv}\{e\} & \text{privileged computation} \\
 & | & \texttt{test}\ p\ ?\ e.m(\overline{e}) : e' & \text{check permissions} \\
 & | & t[e] & \text{framed expression} \\
 & | & \iota & \text{object reference} \\
obj & ::= & \{\tau, \overline{f} \mapsto \overline{\iota}\} & \text{object}
\end{array}
$$

Figure 4. Syntax

mented with some constructs for modeling permissions and access control checks (similar to those in [9] and [16]).

4.1. Syntax

The symbol p ranges over permissions, *Permission* is the set of all permissions, and $\mathcal{P}(Permission)$ is its power set. We only consider two sets of permissions: H and L, where $L \subseteq H$ and $L, H \in \mathcal{P}(Permission)$.

$$
\begin{array}{rcll}
p & \in & Permission & \text{permission} \\
S, D & ::= & H \mid L & \text{static/dynamic permissions} \\
\alpha & ::= & X \mid H \mid L & \text{security levels} \\
\beta & ::= & H \mid L & \text{ground security levels}
\end{array}
$$

The function $packof(t)$ returns the package name of the class t. Given an object of the type $\tau = t\langle\alpha\rangle$, α denote the security level of the object. The variable α ranges over H, L, and X. The occurrences of X in the code of an object of the type $t\langle\alpha\rangle$ are replaced by α. For example, if the class `File` contains a field `path` of the type `String`$\langle X\rangle$, then the `path` field of an object of the type `File`$\langle H\rangle$ has the type `String`$\langle H\rangle$. A value of the type `String`$\langle H\rangle$ can only be created and read by code with static permission set H.

The syntax is given in Figure 4. We use over-bar as abbreviation for arrays. For example, $\overline{\tau}$ represents an array of types $\tau_1, \tau_2, ..., \tau_n$. The symbol d ranges over `+` and `-`, which represents the public and default access modifiers of fields and methods. A field or method with the default access modifier is package-scoped, while public members of a class are accessible to any code. We use τ to represent qualified types of the form $t\langle\alpha\rangle$. We assume that each class has a default constructor for initializing all its fields. A method m's declaration includes a security label α, a method body, and a set P that contains a permission p if m is a sensitive method requiring p and P is empty otherwise. The variable e ranges over expressions such as variable, field access, method call, instantiation, and object reference, framed expression. The expression `doPriv`$\{e\}$ represents that the enclosing method asserts its privilege to evaluate the expression e. The expression `test` $p\ ?\ e.m(\overline{e}) : e'$ represents testing the dynamic permission set for the permission p and if succeeds, then evaluate $e.m(\overline{e})$, where m is a sensitive method that demands permission p, and evaluate e' otherwise. The type rule for method calls will make sure that any call to a sensitive method is proceeded by a proper access-control check.

A framed expression $t[e]$ represents the runtime expression e that is executed with its permissions reduced to the static permissions of class t. The variable ι ranges over object references and an object of the form $\{\tau, \overline{f} \mapsto \overline{\iota}\}$ is of type τ and has fields $\overline{f}$ that hold references $\overline{\iota}$.

4.2. Operational semantics

We give a small-step semantics for the language. An evaluation context as defined below is an expression with a single hole that can be filled in with an expression of proper type.

$$
\begin{array}{rcl}
E[\circ] & ::= & \circ \mid E[\circ].f_i \mid E[\circ].f_i := e \mid \iota.f_i := E[\circ] \\
 & & \mid E[\circ]; e \mid \texttt{new}\ \tau(\overline{\iota}, E[\circ], \overline{e}) \\
 & & \mid E[\circ].m(\overline{e}) \mid \iota.m(\overline{\iota}, E[\circ], \overline{e})
\end{array}
$$

The heap σ, appearing in all reduction rules, maps object references to objects. Each reduction rule is of the form $\sigma, e \rightarrow^S_D \sigma', e'$, where S, D are the static and dynamic permission sets during the reduction. The function $static(t)$ returns the static permission set of the class t.

Rule (Red Invk) reduces a method call of the form $\iota.m(\overline{\iota})$ to $t[e']$, where t is the defining class of m and e' is the method body e of m with X replaced by the security level of the receiver object, the parameters replaced by the arguments $\overline{\iota}$, and *this* replaced by ι. The function $mbody(m,\ \tau)$ (Figure 8) returns the parameter list $\overline{x}$, method body e, defining class t of the method m called on a object of the type τ, and the (possibly empty) permission set P required to call m.

(Red Invk)

$$\frac{\sigma(\iota) = \{\tau, \ldots\} \quad \mathit{mbody}(m,\ \tau) = (\overline{x},\ e,\ t,\ P) \qquad e' = [{}^{\iota}/_{\mathit{this}},\ {}^{\overline{\iota}}/_{\overline{x}}]e \quad P \subseteq D \qquad \forall i, \sigma(\iota_i) = \{t_i\langle\beta_i\rangle, \ldots\},\ P \subseteq \beta_i}{\sigma, \iota.m(\overline{\iota}) \ \to^S_D\ \sigma, t[e']}$$

(Red Select)

$$\sigma, \iota.f \ \to^S_D\ \sigma, \sigma(\iota)(f)$$

(Red Update)

$$\frac{\sigma' = \sigma[\iota \mapsto \sigma(\iota)[f \mapsto \iota']]}{\sigma, \iota.f := \iota' \ \to^S_D\ \sigma', \iota'}$$

(Red New)

$$\frac{\iota\ \mathit{fresh} \quad \tau = t\langle\beta\rangle \quad \beta \leq S \qquad \mathit{fields}(t) = (\_\ \overline{f}) \quad \sigma' = \sigma[\iota \to \{\tau, \overline{f} \mapsto \overline{\iota}\}]}{\sigma, \texttt{new}\ \tau(\overline{\iota}) \ \to^S_D\ \sigma', \iota}$$

(Red Ctx)

$$\frac{\sigma, e \ \to^S_D\ \sigma', e'}{\sigma, E[e] \ \to^S_D\ \sigma', E[e']}$$

(Ctx Frame)

$$\frac{\sigma, e \ \to^{\mathit{static}(t)}_{D \cap \mathit{static}(t)}\ \sigma', e'}{\sigma, t[e] \ \to^S_D\ \sigma', t[e']}$$

(Red Test)

$$\sigma, \texttt{test}\ p\ ?\ e_{true} : e_{false} \ \to^S_D\ \sigma, e_{p \in D}$$

(Red Frame)

$$\sigma, t[\iota] \ \to^S_D\ \sigma, \iota$$

(Ctx DoPriv)

$$\frac{\sigma, e \ \to^S_S\ \sigma', e'}{\sigma, \texttt{doPriv}\{e\} \ \to^S_D\ \sigma', \texttt{doPriv}\{e'\}}$$

(Red DoPriv)

$$\sigma, \texttt{doPriv}\{\iota\} \ \to^S_D\ \sigma, \iota$$

(Red Seq)

$$\sigma, \iota; e \ \to^S_D\ \sigma, e$$

Figure 5. Security-indexed Reduction Rules

If m is a sensitive method requiring permission p, then $P = \{p\}$ and the call to m can proceed only if p is in the dynamic permission set D and the creators of the arguments $\overline{\iota}$ have the permission p. A program is stuck if either condition is not met.

By Rule (Red New), an object of the type $\tau = t\langle\beta\rangle$ can be instantiated in an environment of static permission set S if $\beta \leq S$. That is, only trusted code can instantiate objects of security level H. The auxiliary function $\mathit{fields}(\tau) = (\overline{t}\,\overline{\mathit{fd}})$ returns the list of field declarations $\overline{\mathit{fd}}$ in τ and the defining classes $\overline{t}$ of the fields. The field declaration fd is of the form $d\,\tau\,f$, where d is the access modifier, τ is the type of the field f.

Rule (Ctx Frame) says that the framed expression $t[e]$, given static and dynamic permission sets S and D, reduces to $t[e']$ if e reduces to e' given static permission set $\mathit{static}(t)$ and dynamic permission set $D \cap \mathit{static}(t)$.

By Rule (Ctx DoPriv), a privileged expression $\texttt{doPriv}\{e\}$, given static permission set S, reduces to $\texttt{doPriv}\{e'\}$ if e reduces e' when both the static and dynamic permission sets are S.

By Rule (Red Test), an expression $\texttt{test}\ p\ ?\ e : e'$ reduces to e if p is in the dynamic permission set D, and reduces to e' otherwise.

4.3. Type Rules

The static constraints for protecting values of high security level are embedded in the typing rules (see Figure 6 and 7). Some definitions used in the typing rules are described below.

All types used in untrusted code are labeled with L. If $t\langle\alpha\rangle$ is well-formed and the class t is untrusted, then α must be L. In trusted code, a qualified type $t\langle\alpha\rangle$ is a subtype of $t'\langle\alpha'\rangle$ if the class t extends t' and $\alpha = \alpha'$. while in untrusted code, the security labels do not have to be the same. We make this distinction because we do not expect untrusted code to follow the subtyping constraints in trusted code but we can still make sure that the untrusted code can only write to low security variables and read low security values. The subtyping rule for runtime expression should be the same as the rule in trusted code.

$$L \leq X \leq H \qquad \text{partial order on labels}$$

$$\frac{\alpha \leq \mathit{static}(t) \quad \alpha \leq \mathit{static}(t')}{t' \vdash t\langle\alpha\rangle} \qquad \text{well-formed types}$$

$$\frac{\mathit{Class}(t) = \texttt{class}\ t\langle X\rangle \triangleleft t'\langle X\rangle \ldots}{t \ <:\ t'} \qquad \text{subtyping}$$

$$t \ <:\ t \qquad \frac{t \ <:\ t' \quad t' \ <:\ t''}{t \ <:\ t''} \qquad \text{reflexive/transitive rules}$$

$$\frac{t \ <:\ t'}{t\langle\alpha\rangle \ \preceq\ t'\langle\alpha\rangle}$$

$$\frac{\mathit{static}(t) = L \ \vee\ (t \ <:\ t', \alpha = \alpha')}{t \vdash t\langle\alpha\rangle \ \preceq\ t'\langle\alpha'\rangle} \qquad \text{subtyping of qualified types}$$

A type judgment is of form $\Gamma; \Sigma; t \vdash^P_\alpha e : \tau$, where the environment Γ maps variables to their types, the environment Σ maps object references to their types. If e is reduced from a call to method m, then α is the security level of the call and t is the defining class of m. The set P either contains a single permission p or is an empty set.

We omit the environment Σ or Γ in type rules if the environment is always empty. Rules (T-Invk), (T-Select) ensure that only public members of a class can be accessed by code in other packages, while Rule (T-New) ensures that only a class with all public fields can be instantiated by code in other packages. This restriction is not necessary if we have constructors that only instantiate public fields.

(T-Invk)

$$\frac{\begin{array}{c}\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e : \tau_0 \quad \mathit{mtype}(m,\ \tau_0) = (\overline{\tau}{\rightarrow}\tau, d, \alpha', t', P') \\ \Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \overline{e} : \overline{\tau'} \quad t \vdash \overline{\tau'} \preceq \overline{\tau} \\ d = \texttt{+} \ \vee\ \mathit{packof}(t) = \mathit{packof}(t') \\ \mathit{static}(t') = L \ \vee\ \mathit{static}(t) = L \ \vee\ (\alpha \leq \alpha',\ P' \subseteq P)\end{array}}{\Gamma;\Sigma;t \vdash^{P}_{\alpha} e.m(\overline{e}) : \tau}$$

(T-Test)

$$\frac{\Gamma;\Sigma;t \vdash^{\{p\}}_{\alpha} e.m(\overline{e}) : \tau \quad \Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e' : \tau}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \texttt{test}\ p\ ?\ e.m(\overline{e}) : e'\ :\ \tau}$$

(T-Seq)

$$\frac{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e : \tau \quad \Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e' : \tau'}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e;e' : \tau'}$$

(T-Var)

$$\frac{}{\Gamma;t \vdash^{\emptyset}_{\alpha} x : \Gamma(x)}$$

(T-New)

$$\frac{\begin{array}{c}t \vdash \tau \quad \mathit{fields}(\tau) = (\overline{t}\ \overline{d\ \tau\ f}) \quad \Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \overline{e} : \overline{\tau'} \\ t \vdash \overline{\tau'} \preceq \overline{\tau} \quad \forall i,\ d_i = \texttt{+} \ \vee\ \mathit{packof}(t_i) = \mathit{packof}(t)\end{array}}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \texttt{new}\ \tau(\overline{e}) : \tau}$$

(T-Select)

$$\frac{\begin{array}{c}\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e : \tau \quad \mathit{fields}(\tau) = (\overline{t}\ \overline{d\ \tau\ f}) \\ d_i = \texttt{+} \ \vee\ \mathit{packof}(t_i) = \mathit{packof}(t)\end{array}}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e.f_i : \tau_i}$$

(T-Update)

$$\frac{\begin{array}{c}\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e.f : \tau \\ \Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e' : \tau' \\ t \vdash \tau' \preceq \tau\end{array}}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} e.f := e'\ :\ \tau}$$

(T-DoPriv)

$$\frac{\Gamma;\Sigma;t \vdash^{\emptyset}_{\mathit{static}(t)} e : \tau}{\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \texttt{doPriv}\{e\} : \tau}$$

(T-Ref)

$$\frac{t \vdash \Sigma(\iota)}{\Sigma;t \vdash^{\emptyset}_{D} \iota : \Sigma(\iota)}$$

(T-Frame)

$$\frac{\Sigma;t' \vdash^{\emptyset}_{D\cap \mathit{static}(t')} e : \tau \quad t \vdash \tau}{\Sigma;t \vdash^{\emptyset}_{D} t'[e] : \tau}$$

Figure 6. Expression Typing Rules

By Rule (T-Invk), if a method m in a context with security level α, then either caller or callee is defined in untrusted class or α is no higher than the the security level of the call α'.

The auxiliary function $\mathit{mtype}(m,\ \tau_0)$ (Figure 8) returns the type signature $\overline{\tau} \rightarrow \tau$ of the method m called on a receiver object of type τ_0, and it also returns the method's access modifier d, security level α', its defining class t', and permission set P' required to invoke m.

Also, by Rule (T-Invk), if $\Gamma;\Sigma;t \vdash^{P}_{\alpha} e.m(\overline{e}) : \tau$ and t, t' are both trusted classes, then the set P must contain the permission set required for invoking m. The set P is empty if the method call appears in any context other than a test expression. By Rule (T-Test), $\Gamma;\Sigma;t \vdash^{\emptyset}_{\alpha} \texttt{test}\ p\ ?\ e.m(\overline{e}) : e'\ :\ \tau$ implies $\Gamma;\Sigma;t \vdash^{\{p\}}_{\alpha} e.m(\overline{e}) : \tau$. Thus, call to a sensitive method that requires permission p is typable only if the method is called in a test expression that checks for permission p.

By Rule (T-DoPriv), a privileged expression $\texttt{doPriv}\{e\}$ is well-typed in class t if e is well-typed given security level $\mathit{static}(t)$.

By Rule (T-Method), the security levels of the argument/return types of a public method are not H. This prevents untrusted code from casting low security values to high security types and vice versa. Moreover, if m in class t is a sensitive method that requires the permission p, then m cannot be public, t is a trusted class, and the security levels of the argument/return types of m are higher than security level α of m. The first two restrictions prevent untrusted code from accessing the sensitive methods directly. The last restriction ensures that the arguments and return values of m always have security levels containing p. The reason is that by Rule (Red-Invk), the current permission set of the execution context of m always includes the permission p, while by Rule (T-Invk), the security level of the method call contains the current permission set.

The predicate $\mathit{override}(m, t', \overline{\tau} \rightarrow \tau, d, \alpha, t, P)$ (see Figure 8) defines valid method overridding such that only public method can be overridden by methods in other packages, method overriding must not change argument/return types and the permission set P required to call m, and the security level of m is no lower than that of the overridden method. Note that the the type signature of a method in untrusted code are all labeled with L even if the method overrides another method in trusted code.

Rule (T-Class) requires that a trusted class can only extend another trusted class. Rule (T-Store) describes the condition of a valid heap σ given an environment Σ such that σ and Σ have the same domain, for each object reference, ι, the type of the object $\sigma(\iota)$ is the same as $\Sigma(\iota)$, and $\sigma(\iota)$ is well-typed by Rule (T-Object). If an object is well-typed, then the security level of each

(T-Method)

$$\frac{\begin{array}{c} \mathit{override}(m, t'\langle X\rangle, \overline{\tau} \to \tau, d, \alpha, t, P) \\ \overline{x} : \overline{\tau}, \mathit{this} : t\langle X\rangle; t \vdash^{\emptyset}_{\alpha} e : \tau' \\ t \vdash \tau' \preceq \tau \quad t \vdash \tau, \overline{\tau} \quad \tau = t_0\langle \alpha_0\rangle \quad \overline{\tau} = \overline{t\langle \alpha\rangle} \\ d = \texttt{-} \ \vee \ (\alpha_0 \neq H, \forall i, \alpha_i \neq H) \\ P = \emptyset \text{ or } (P = \{p\} \text{ and} \\ \mathit{static}(t) = H,\ d = \texttt{-},\ \alpha \leq \alpha_0,\ \forall i, \alpha \leq \alpha_i) \end{array}}{t\langle X\rangle \triangleleft t'\langle X\rangle \vdash d\ \tau\ m^P_\alpha(\overline{\tau}\ \overline{x})\,\{\,\texttt{ret}\,e;\,\}}$$

(T-Class)

$$\frac{\begin{array}{c} \forall i, t \vdash \tau_i \quad \tau_i = t_i\langle \alpha_i\rangle \quad d_i = \texttt{-} \ \vee \ \alpha_i \neq H, \\ t\langle X\rangle \triangleleft t'\langle X\rangle \vdash \overline{mth} \quad \mathit{static}(t) \leq \mathit{static}(t') \end{array}}{\texttt{class}\ t\langle X\rangle\ \triangleleft\ t'\langle X\rangle\ \{\ \overline{d\ \tau\ f}; \overline{mth}\ \}\ \text{ok}}$$

(T-Store)

$$\frac{\begin{array}{c} \mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma) \\ \forall \iota \in \mathrm{dom}(\sigma)\,.\,\sigma(\iota) = \{\tau, \overline{f} \mapsto \overline{\iota}\},\ \Sigma(\iota) = \tau,\ \Sigma \vdash \sigma(\iota) \end{array}}{\Sigma \vdash \sigma}$$

(T-Object)

$$\frac{\mathit{fields}(\tau) = (\overline{t}\ \overline{d\ \tau\ f}) \quad \forall i, \Sigma(\iota_i) \preceq \tau_i}{\Sigma \vdash \{\tau, \overline{f} \mapsto \overline{\iota}\}}$$

Figure 7. Typing Rules for Methods, Classes, Store

$$\frac{\begin{array}{c} \mathit{Class}(t) = \texttt{class}\ t\langle X\rangle\ \triangleleft\ t'\langle X\rangle\ \{\ \overline{fd}; \overline{mth}\ \} \\ m \text{ is not defined in } \overline{mth} \end{array}}{\mathit{method}(m, t) = \mathit{method}(m, t')}$$

$$\frac{\begin{array}{c} \mathit{Class}(t) = \texttt{class}\ t\langle X\rangle\ \triangleleft\ t'\langle X\rangle\ \{\ \overline{fd}; \overline{mth}\ \} \\ m \text{ is defined in } mth \in \overline{mth} \end{array}}{\mathit{method}(m, t) = (t, mth)}$$

$$\frac{\begin{array}{c} \mathit{method}(m, t) = (t', mth) \\ mth = d\ \tau\ m^P_{\alpha'}(\overline{\tau}\ \overline{x})\,\{\,\texttt{ret}\,e;\,\} \end{array}}{\mathit{mtype}(m,\ t\langle \alpha\rangle) = ([\,^{\alpha}/_{X}]\overline{\tau} \to [\,^{\alpha}/_{X}]\tau, d, [\,^{\alpha}/_{X}]\alpha', t, P)}$$

$$\frac{\begin{array}{c} \mathit{method}(m, t) = (t', mth) \\ mth = d\ \tau\ m^P_{\alpha'}(\overline{\tau}\ \overline{x})\,\{\,\texttt{ret}\,e;\,\} \end{array}}{\mathit{mbody}(m,\ t\langle \alpha\rangle) = (\overline{x},\ [\,^{\alpha}/_{X}]e, t', P)}$$

$$\mathit{fields}(\text{Object}) = ()$$

$$\frac{\begin{array}{c} \mathit{Class}(t) = \texttt{class}\ t\langle X\rangle\ \triangleleft\ t'\langle X\rangle\ \{\ \overline{fd}; \dots\ \} \\ \mathit{fields}(t') = (\overline{t'\ fd'}) \end{array}}{\mathit{fields}(t) = (\overline{t'\ fd'}, t\ fd_1, \dots, t\ fd_n)}$$

$$\frac{\mathit{fields}(t) = (\overline{t}\ \overline{fd})}{\mathit{fields}(t\langle \alpha\rangle) = (\overline{t}\ [\,^{\alpha}/_{X}]\overline{fd})}$$

$$\frac{\begin{array}{c} \text{if } \mathit{mtype}(m,\ t_0\langle X\rangle) = (\overline{\tau'} \to \tau', d', \alpha', t_0', P') \text{ then} \\ d' = \texttt{+} \ \vee \ \mathit{packof}(t) = \mathit{packof}(t_0') \\ (\overline{\tau'}, \tau', d', P') = (\overline{\tau}, \tau, d, P) \quad \alpha' \leq \alpha \end{array}}{\mathit{override}(m, t_0\langle X\rangle, \overline{\tau} \to \tau, d, \alpha, t, P)}$$

Figure 8. Auxiliary Functions

of its field f_i must be the same as that of the object referenced in f_i. Since the fields of a untrusted class are all low security level, objects of the untrusted class can never hold objects of high security level.

Note that we treat an object created in trusted code as a high security value, and this is true even if the object (*e.g.* a string) is constructed from both low and high security elements.

4.4. Properties

Following Featherweight Java, we assume an implicit class table *Class* and for any class t, we assume that $\mathit{Class}(t)$ is well-typed. The environment Γ is omitted in type judgments if it is empty.

The following lemma says that if an expression in class t is well-typed, then the type τ of the expression is well-formed in t as well. In other words, no expression in untrusted code can have a type with security level H.

Lemma 4.1 *If* $\Sigma; t \vdash^P_\alpha e : \tau$, *then* $t \vdash \tau$.

Proof Proof is by straightforward induction on the type derivation of e.

The subject-reduction lemma says that if a well-typed expression e of class t is reduced to an expression e', then e' is also well-typed. Also, if τ and τ' are the types of e and e' respectively, then $\tau' \preceq \tau$. Thus, an expression of low security type cannot be reduced to another expression of high security type. The lemma also proves that subject reduction preserves the typing of store, which implies that objects of untrusted class cannot have fields that hold references to objects with high security level.

Lemma 4.2 (Subject Reduction) *If* $\Sigma; t \vdash^P_D e : \tau$, $\Sigma \vdash \sigma$, *and* $\sigma, e \to^S_D \sigma', e'$, *where* $D \leq S = \mathit{static}(t)$, *then* $\exists\Sigma'$ *such that* $\Sigma' \vdash \sigma'$ *and* $\exists P' \subseteq D$ *such that* $\Sigma'; t \vdash_D^{P'} e' : \tau'$, *where* $\tau' \preceq \tau$.

Note that in the lemma above, if the expression e is a call to a sensitive method m that requires permission p, then the type of e' must have a security level β such that $p \in \beta$. Thus, confidential values retrieved from sensitive methods will be labeled with proper security levels so that these values not accessible to unauthorized code.

The progress lemma shows that a well-typed expression can make progress. In particular, the successful reduction of a call to the method m means that if m is a sensitive method, then the dynamic permission set must include the permission p required to call m and the arguments to m are created by code with the permission p. This is to prevent untrusted code from affecting security-sensitive operation that it is not authorized for.

Lemma 4.3 (Progress) *If* $\Sigma; t \vdash_D^P e : \tau$, $\Sigma \vdash \sigma$, $P \subseteq D$, *and* e *is not a reference value, then* $\exists\sigma', e'$ *such that* $\sigma, e \rightarrow_D^S \sigma', e'$, *where* $S = static(t)$.

The proof of the subject reduction lemma and progress lemma is included in the appendix.

We say that an irreducible expression e is stuck if it is not a value. Based on the subject reduction and progress lemma, we can show by induction that a well-typed program does not get stuck. Let $\Rightarrow_D^S$ be the reflexive and transitive closure of $\rightarrow_D^S$.

Theorem 4.4 *If* $\emptyset; t \vdash_S^P e : \tau$, $P \subseteq S = static(t)$, *and* $\exists\sigma', e'$ *such that* $\emptyset, e \Rightarrow_S^S \sigma', e'$, *then* e' *is not stuck.*

5. Discussion

We presented a simple way to strengthen the protection of security sensitive resources using stack inspection. However, we have to consider a number of issues for this approach to be more practical.

Generalization We have only modeled codebases with two sets of permissions H and L. In practice, applications often have security policies establishing multi-levels of trust. We believe that our type system may be generalized to handle such cases. For example, suppose we have an application with the set of permissions S and it interacts with a class t that belongs to a codebase with permission set S' where $S \not\subseteq S'$. Then we can annotate the objects of the type t with security level L, and we treat the types of the fields and methods in the class t as low security level as well. This kind of annotation guarantees that the application will not violate security policies by accidentally accessing system resources on behalf of untrusted code. The application may choose to trust code with more permissions such as those in system libraries (assuming that the libraries are properly annotated and the application is aware of those annotations).

Implicit flow We have not considered any implicit information flow from untrusted code to high security variables or flow from secret values to untrusted code via conditional statements or covert channels. For instance, a trusted class could contain methods that set the file name for deletion to some default value. The untrusted class may simply call these methods of the trusted class to affect which files get deleted by trusted code. In this case, untrusted code can have indirect impact on sensitive operations without modifying high security variables directly. There are abundant work [19, 20, 12, 14, 2, 3] in preventing implicit flow but most would impose strict conditions on programs.

Declassification and Untainting Sometimes, it is appropriate to supply low security arguments to the sensitive methods when dynamic permission set is H. We can check the input values to ensure that they satisfy appropriate constraints. For example, to read a font file for untrusted client, the trusted code asserts its privilege with `doPrivileged` method so that it can accept a string that specifies the path to a font file from a untrusted caller. The string is of low security level but after the trusted code has verified that the string actually is a legal path to a font file, it may *cast* the security level of the string object to H. Such casts should be rare and be used only after careful checking of low security values.

6. Related Work

Some of the past research on stack inspection has focused on its design and implementation [21, 18], static analysis to guarantee the success of stack-based access control checks [16], formal semantics and its effects on compiler optimization [6, 9], and alternative mechanisms [1].

The "trust analysis" of Ørbæk and Palsberg [15] uses types to track data flow of the trusted and untrusted values in the settings of higher-order functional language. The inputs to security-sensitive operations must pass necessary checks. The present work is similar in principle but we also consider the execution context of these operations.

Shankar *et al.* [17] proposed a static analysis approach to detect format string vulnerability with type qualifiers. The format-string attack to C programs exploits the misuse of variable-argument functions such as printf in C libraries. These functions may be called in a context with high level of trust and a knowledgeable adversary can pass maliciously constructed strings directly or indirectly to the problematic functions and write to arbitrary memory locations. Shankar *et al.* statically track the values from untrusted sources by "tainted" types and if such values are used as arguments to functions such as printf, a warning is generated. Their approach is similar to Perl's taint mode with the difference that in Perl, the tainted values are tracked by runtime taint propagation. In comparison to the present work, Shankar *et al.* only track tainted primitive values such as strings and their pointers. Also, they do not need to consider interaction with code from untrusted sources.

Abadi and Fournet [1] give an alternative mechanism to stack inspection in that access control decisions are based on execution history of a program. Their algorithm computes the dynamic permission by taking the intersection of the static permissions of all the codebases identified in the execution history. This approach eliminates the need to track tainted values as the creator of the tainted value must be recorded in the execution history. Similar to the dependency-tracking semantics in [9], history-based access control can be more restrictive than necessary. For example, some computation may interact with untrusted code without impacting security-sensitive operations. Nevertheless, the current permissions are decreased since the execution history includes the untrusted code. Because of this, constructs such as $\mathit{Accept}(P)\{B\}$ are used to restore some of the permissions in P lost after running the untrusted program B. However, this kind of explicit permission change increases the number of places where inadvertent errors may occur.

Besson *et al.* [5, 4] infer a precondition on the call stack based on the call-graph of library code to ensure that the execution of method call does not violate some global property. They developed a static analysis tool for reviewing the security of system libraries in CLR and to find possible security errors in programs that rely on stack-based access control. Their tool constructs permission-sensitive call graphs from trusted programs and does not check untrusted programs. Manual queries can be made on the call graph to identify "unsafe" flow paths where untrusted program can perform security-sensitive operations. We do not prevent such unsafe flow but our analysis may make these "unsafe" flow paths less harmful by preventing untrusted programs from passing "tainted" values to security-sensitive operations at the same time.

Guttman *et al.* [11] use model checking to analyze the access control policies of an operation system to determine whether information flow between untrusted and security-sensitive endpoints always traverses trusted programs in order to protect data integrity and secrecy. "Tainted" inputs from untrusted sources are always checked before they may be used for sensitive computation and outbound data is always filtered so that confidential data will not reach untrusted destination. Because a type of resource or process in a security policy may support operations that are not security sensitive, some direct information flow from/to untrusted endpoints is allowed as exceptions. The present work may help separate the information flow related to security-sensitive computation from the rest so as to verify the security properties for these exceptions.

Pottier *et al.* [16] use static type analysis to guarantee the success of stack inspection checks. They assume that the checked permissions are atomic values while in practice, permission values can be dependent on the input values. For example, the File permission of Java depends on the String object that specifies the file path. The type system in the present work does not need to compare permission values. (We do check whether a required permission has been tested before calling sensitive method but we do not need to determine whether the permission belongs to the static permission sets of the callers.)

The SLam calculus [12] associates the type of a value with direct (and indirect) readers and creators of the value to enforce more flexible access control for security policies of secrecy and integrity. The indirect readers and creators of a value may receive secure information implicitly and the dissemination of that information may be less restrictive. We do not separate the integrity and secrecy levels of an object for simplicity.

Banerjee and Naumann [3] presented a type system that can ensure non-interference properties for programs that use access-control constructs to protect confidential information. A method that includes an access-control check may return high or low security values depending on the enabled permissions of the callers. Thus, they gave multiple types to the method to cover every kind of return value and each type designates an upper bound of permissions that are not enabled by the callers of the method. Their type rules require knowing statically whether the enabled permissions of a caller include the permission required to pass the access-control check so that the high security value returned by the method (if the check succeeds) is not given to a low security variable in a caller that expects the check to fail. In contrast, we can annotate return type of a sensitive method using a polymorphic security variable so that the security level of the return value depends on the security level of the calling context. Thus, we do not have to statically determine whether the callers have certain permission, which can be helpful if the permission depends on runtime values.

7. Conclusion and Future Work

We have presented a simple type-based method for preventing security defects that allow untrusted code to circumvent stack-based control checks. Our approach is entirely static and only needs to be applied to trusted system library or local application. We make no assumption about the untrusted code and we rely on the access modifiers of the fields and methods to protect high security level variables and values. Those high security level variables may be safely used in a trusted environment for security-sensitive operations.

As future work, we would like to implement an algorithm to infer the type annotations for trusted classes. For example, we label the objects of untrusted classes with security level L and treat the types of the fields and methods in untrusted classes as low security level as well. In a trusted class, if the security label of the type of a field is α and the field is not package-scoped, then α must satisfy the constraint $\alpha \leq X$. If the field is assigned a value with security label L, then $\alpha \leq L$ and we must let α be L. If the field is used as an argument in a call to a sensitive method while the method call is executed with dynamic permission set H, then we must have $H \leq \alpha$ and in this case α should be H. We initially set the security labels of the types in the field and methods of all trusted classes to be X and then apply the inference rule to each trusted class iteratively until either we find a conflict among constraints or no more changes can be applied to the security labels.

Acknowledgments: We thank Jan Vitek and Jens Palsberg for helpful suggestions and the anonymous referees for valuable comments.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, February 2003.

[2] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. 15th IEEE Computer Security Foundations Workshop*, 2002.

[3] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *Proc. 16th IEEE Computer Security Foundations Workshop*, 2003.

[4] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From stack inspection to access control: A security analysis for libraries. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.

[5] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proc. 4th Conference on Principles and Practice of Declarative Programming*. ACM Press, 2002.

[6] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspections. In *European Symposium on Programming (ESOP)*, volume 2618 of *Lecture Notes in Computer Science*, pages 22–37. Springer Verlag, Apr. 2003.

[7] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–242, 1976.

[8] D. E. Denning and P. J. Denning. Certifications of programs for secure information flow. *Commun. ACM*, 20(7):504–512, July 1977.

[9] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2002.

[10] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, Reading, MA, USA, 2003.

[11] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. In *Workshop on Issues in the Theory of Security*, 2003.

[12] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[13] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–146, 1999.

[14] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[15] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, Nov. 1997. Preliminary version in Proceedings of SAS'95, International Static Analysis Symposium, Springer-Verlag (*LNCS* 983), pages 314–330, Glasgow, Scotland, September 1995.

[16] F. Pottier, C. Skalka, and S. Smith. A Systematic Approach to Static Access Control. *ACM Transactions on Programming Languages and Systems*, Nov. 2003. To appear.

[17] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., Aug. 2001.

[18] Úlfar Erlingsson and F. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000. IEEE Computer Society.

[19] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings TAPSOFT'97*, volume 1214 of *LNCS*, Berlin, Germany, 1997. Springer-Verlag.

[20] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[21] D. S. Wallach and E. W. Felten. Understanding java stack inspection. In *Proceedings of the 1998 Conference on Security and Privacy*, pages 52–62, 1998.

A. Proof for subject reduction and progress lemmas

Lemma 2 (Subject Reduction) *If* $\Sigma; t \vdash_D^P e : \tau$, $\Sigma \vdash \sigma$, *and* $\sigma, e \rightarrow_D^S \sigma', e'$, *where* $D \leq S = \mathit{static}(t)$, *then* $\exists \Sigma'$ *such that* $\Sigma' \vdash \sigma'$ *and* $\exists P' \subseteq D$ *such that* $\Sigma'; t \vdash_D^{P'} e' : \tau'$, *where* $\tau' \preceq \tau$.

Proof We prove by induction on the structure of e and there are eleven cases depending on which reduction rule applies to $\sigma, e \rightarrow_D^S \sigma', e'$.

Case (Red Test): suppose that $e = \texttt{test}\ p\ ?\ e_0.m(\overline{e})\ :\ e_0'$. From $\Sigma; t \vdash_D^\emptyset e : \tau$, we have that $\Sigma; t \vdash_D^{\{p\}} e_0.m(\overline{e}) : \tau$ and $\Sigma; t \vdash_D^\emptyset e_0' : \tau$. If $p \in D$, then let $e' = e_0.m(\overline{e})$ and $P' = \{p\}$ such that $\Sigma; t \vdash_D^{P'} e' : \tau$. If $p \notin D$, then $e' = e_0'$ and $\Sigma; t \vdash_D^\emptyset e' : \tau$.

Case (Red Invk): suppose that $e = \iota_0.m(\overline{\iota})$, $\Sigma(\iota_0) = \tau_0 = t_0\langle\beta_0\rangle$, and $\mathit{mbody}(m,\ \tau_0) = (\overline{x},\ e_0,\ t',\ P_0)$.

If $\mathit{mtype}(m,\ \tau_0) = (\overline{\tau} \rightarrow \tau, d, \alpha, t', P_0)$, then from Rule T-Method, $\Gamma; t' \vdash_{D'}^\emptyset e_0' : \tau''$, where $\Gamma = \overline{x} : \overline{\tau}, \mathit{this} : t'\langle\beta_0\rangle$, $t' \vdash \tau'' \preceq \tau$, $e_0' = [{}^{\beta_0}/_X]e_0$, and $D' = [{}^{\beta_0}/_X]\alpha$.

From $t' \vdash \tau$ (Rule T-Method) and $t' \vdash \tau''$ (Lemma 4.1), we have $\tau'' \preceq \tau$. Rule T-Invk gives

$$t \vdash \overline{\tau'} \preceq \overline{\tau} \tag{1}$$

We show that if $\Sigma; t \vdash_D^\emptyset \overline{\iota} : \overline{\tau'}$. then

$$\overline{\tau'} \preceq \overline{\tau} \tag{2}$$

$$t' \vdash \overline{\tau'} \tag{3}$$

If class t is trusted, then (1) implies (2). If class t' is also trusted, then $t \vdash \overline{\tau'}$ (Rule T-Ref) implies (3). If t' is untrusted, then by Rule T-Method, we have $t' \vdash \overline{\tau}$, which means that the security labels in $\overline{\tau}$ are all L. From (2), the security labels in $\overline{\tau'}$ are L and (3) also holds.

If class t is untrusted, then the security labels in $\overline{\tau'}$ and β_0 are L. In this case, (3) holds. If class t' is also untrusted, then the labels in $\overline{\tau}$ are also L. If class t' is trusted, then m must be public (t and t' are not in the same package) and from Rule T-Method, we know that the labels in $\overline{\tau}$ are L or X. Thus in both cases, (1) implies (2).

From (3) and Rule T-Ref, we have $\Sigma; t' \vdash_{D'}^\emptyset \overline{\iota} : \overline{\tau'}$. Let $e_0'' = [{}^{\overline{\iota}}/_{\overline{x}},\ {}^{\iota_0}/_{\mathit{this}}]e_0'$, Together with (2), we can show by induction that $\Sigma; t' \vdash_{D'}^\emptyset e_0'' : \tau'$ and $\tau' \preceq \tau'$.

At last, we need to show $\Sigma; t \vdash_D^\emptyset t'[e_0''] : \tau'$ or to show $\Sigma; t' \vdash_{D \cap \mathit{static}(t')}^\emptyset e_0'' : \tau'$ by Rule (T-Frame). From Rule (T-Invk), either class t or t' is untrusted, or $D \leq D'$. If the former is true, then $D \cap \mathit{static}(t') = L \leq D'$; otherwise, $D \cap \mathit{static}(t') = D \leq D'$. Thus, together with $\Sigma; t' \vdash_{D'}^\emptyset e_0'' : \tau'$, it is easy to show by induction that $\Sigma; t' \vdash_{D \cap \mathit{static}(t)}^\emptyset e_0'' : \tau'$.

Case (Red Select): if $e = \iota.f_i$, then $e' = \sigma(\iota)(f_i) = \iota_i$ and $\tau' = \Sigma(\iota_i)$. From Rules T-Select and T-Ref, we have $\Sigma; t \vdash \iota : \Sigma(\iota)$ and if $\Sigma(\iota) = \tau_0 = t_0\langle\beta_0\rangle$ then $\beta_0 \leq \mathit{static}(t)$. The only interesting case is when class t is untrusted but t_0 is trusted. Suppose $\mathit{fields}(\tau_0) = (\overline{t}\ \overline{d\ \tau\ f})$. Then $\tau = \tau_i$. By Rule T-Select, since t and t' are not in the same package, the field f_i must be public and by Rule T-Class, the type of this field must not be labeled with H. Since $\beta_0 \leq \mathit{static}(t) = L$, τ_i must be labeled with L. From Rule T-Object, $\Sigma(\iota_i) \preceq \tau_i$. Thus, the label of $\Sigma(\iota_i)$ is L as well, which implies $t \vdash \Sigma(\iota_i)$ and by Rule T-Ref, we have $\Sigma; t \vdash_D^\emptyset \sigma(\iota)(f_i) : \tau'$ and $\tau' \preceq \tau$.

The proof for the case (Red Update) is similar to the previous one.

Case (Red New): if $e = \texttt{new}\ \tau(\overline{\iota})$ and $\tau = t_0\langle\beta\rangle$, and $\sigma, e \rightarrow_D^S \sigma', \iota$, then $\sigma' = \sigma[\iota \mapsto \{\tau, \overline{f} \mapsto \overline{\iota}\}$ and $\beta \leq S$. Since $S = \mathit{static}(t)$, $t \vdash \tau$. Let $\Sigma' = \Sigma[\iota \mapsto \tau]$, then $\Sigma'; t \vdash_D^\emptyset \iota : \tau$. From Rule T-New, if $\Sigma; t \vdash_D^\emptyset \overline{\iota} : \overline{\tau'}$ and $\mathit{fields}(\tau) = (\overline{t}\ \overline{d\ \tau\ f})$, then $t \vdash \tau$ and $t \vdash \overline{\tau'} \preceq \overline{\tau}$. From Rule T-Class, we have $t_0 \vdash \overline{\tau}$ and from Rule T-New, for each i, either f_i is public or t and t_0 are in the same package.

If class t is trusted, then it is immediate that

$$\overline{\tau'} \preceq \overline{\tau}. \tag{4}$$

If t and t_0 are untrusted, then the security labels in $\overline{\tau}$ are all L and (4) is also true. If t is untrusted but t_0 is trusted, then $t \vdash \tau$ implies $\beta = L$ and since t,t_0 are in different packages, all fields of t_0 are public and the security labels in $\overline{\tau}$ are all L. Thus, (4) holds and by Rule T-Object, we have $\Sigma' \vdash \sigma(\iota)$. From Rule T-Store, we have $\Sigma' \vdash \sigma'$.

Case (Red Ctx): if $e = E[e_0]$ and $\sigma, e_0 \rightarrow_D^S \sigma', e_0'$, then $e' = E[e_0']$. By induction hypothesis, e_0' is well-typed and by simple case analysis, it easy to show that e' is also well-typed.

Case (Ctx Frame): if $e = t'[e_0]$ and $\sigma, e_0 \rightarrow_{D \cap S'}^{S'} \sigma', e_0'$, where $S' = \mathit{static}(t')$, then $e' = t'[e_0']$. By Rule

(T-Frame) and $\Sigma; t \vdash_D^{\emptyset} e : \tau$, we have $\Sigma; t_0 \vdash_{D \cap S'}^{\emptyset} e_0 : \tau$. By induction hypothesis, $\exists \tau'$ such that $\Sigma; t' \vdash_{D \cap S'}^{\emptyset} e_0' : \tau'$ and $\tau' \preceq \tau$. Thus, from Rule T-Frame, we have $\Sigma; t \vdash_D^{\emptyset} e' : \tau'$.

Case (Red Frame): if $e = t'[\iota]$, then $e' = \iota$. From Rules T-Frame and T-Ref, we have $t \vdash \tau = \Sigma(\iota)$ and thus $\Sigma; t \vdash_D^{\emptyset} \iota : \tau$ by Rule T-Ref.

Proof for the cases (Ctx DoPriv), (Red DoPriv), and (Red Seq) is trivial

Lemma 3 (Progress) *If* $\Sigma; t \vdash_D^P e : \tau$, $\Sigma \vdash \sigma$, $P \subseteq D$, *and* e *is not a reference value, then* $\exists \sigma', e'$ *such that* $\sigma, e \rightarrow_D^S \sigma', e'$, *where* $S = \mathit{static}(t)$.

Proof We prove by induction on the type derivation of e. Depending on the last rule used in the derivation, there are eight cases.

Case (T-Invk): if $e = e_0.m(\overline{e})$ then by Rule T-Invk, e_0 and $\overline{e}$ are well-typed. By induction hypothesis, e_0 or one of $\overline{e}$ can be reduced to other expressions. Thus, by Rule (Red Ctx), the expression e can make progress.

Suppose $e = \iota.m(\overline{\iota})$, $\sigma(\iota) = \{\tau, \ldots\}$, $\mathit{mbody}(m,\ \tau) = (\overline{x},\ e_0,\ t,\ P_0)$, and $\forall i,\ \sigma(\iota_i) = \{t_i'\langle\beta_i\rangle, \ldots\}$. We will show that $P_0 \subseteq D$ and $\forall i$, $P_0 \subseteq \beta_i$.

By Rule T-Invk, we have $P_0 \subseteq P$ and by assumption that $P \subseteq D$, we have $P_0 \subseteq D$. From Rule T-Method, if $\mathit{mtype}(m,\ \tau) = (\overline{\tau} \rightarrow \tau, d, \beta, t, P_0)$ and $\forall i,\ \tau_i = t_i\langle\beta_i\rangle$, then $\beta \leq \beta_i$. By Rule T-Invk, $D \leq \beta$. Thus, $\forall i$, $D \leq \beta_i$ Because D, β_i are either L or H, we have $D \subseteq \beta_i$. Therefore, $\forall i$, $P' \subseteq D \subseteq \beta_i$.

Thus, by Rule (Red Invk), $\sigma, \iota.m(\overline{\iota}) \rightarrow_D^S [{}^{\iota}/_{\mathit{this}}, {}^{\overline{\iota}}/_{\overline{x}}]e_0$.

Case (T-Seq): if $e = e_1; e_2$, then since e_1 is well-typed by Rule T-Seq, we can apply induction hypothesis to e_1 and then use Rule (Red Ctx). If $e = \iota; e_2$, then it reduces to e_2 by Rule (Red Seq).

Case (T-New): if $e = \texttt{new}\ \tau(\overline{e})$, then by Rule T-new, $\overline{e}$ is well-typed and thus we can apply induction hypothesis to show that one of $\overline{e}$ can be reduced to other expression, and then use Rule (Red Ctx) to show that e makes progress. If $e = \texttt{new}\ \tau(\overline{\iota})$, then by Rule T-New, we have $t \vdash \tau$ which means that if $\tau = t\langle\beta\rangle$, then $S = \mathit{static}(t) \leq \beta$, thus, by Rule (Red New), e can be reduced.

The proof for the cases (T-Select), (T-Update), (T-DoPriv), and (T-Frame) are similar to previous case.

Case (T-Test): if $e = \texttt{test}\ p\ ?\ e_{true} : e_{false}$. then by Rule R-Test, $\sigma, e \rightarrow_D^S \sigma, e_{p \in D}$.