2013

CONVERSION OF DOMAIN TYPE ENFORCEMENT LANGUAGE TO THE JAVA SECURITY MANAGER

JAMES WALKER
Michigan Technological University

CONVERSION OF DOMAIN TYPE ENFORCEMENT LANGUAGE TO THE JAVA SECURITY MANAGER

By

James Walker

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2013

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Report Advisor:	*Steven Carr*
Report Co-Advisor:	*Jean Mayo*
Committee Member:	*Xinli Wang*
Department Chair:	*Charles Wallace*

Table of Contents

Abstract

With today's prevalence of Internet-connected systems storing sensitive data and the omnipresent threat of technically skilled malicious users, computer security remains a critically important field. Because of today's multitude of vulnerable systems and security threats, it is vital that computer science students be taught techniques for programming secure systems, especially since many of them will work on systems with sensitive data after graduation. Teaching computer science students proper design, implementation, and maintenance of secure systems is a challenging task that calls for the use of novel pedagogical tools. This report describes the implementation of a compiler that converts mandatory access control specification Domain-Type Enforcement Language to the Java Security Manager, primarily for pedagogical purposes. The implementation of the Java Security Manager was explored in depth, and various techniques to work around its inherent limitations were explored and partially implemented, although some of these workarounds do not appear in the current version of the compiler because they would have compromised cross-platform compatibility. The current version of the compiler and implementation details of the Java Security Manager are discussed in depth.

Chapter 1. Introduction

1.1. Background

Designing and enforcing information security policies is a significant challenge. Today, system designers and administrators face a combination of ubiquitous Internet connections, heavy dependence on functional computing infrastructures, storage of massive amounts of sensitive data, and myriad technically skilled malicious agents. In this environment, maintaining a secure system is crucial.

Many tools and techniques have been developed to address this challenge. Among these tools are discretionary access controls and mandatory access controls. Discretionary access controls restrict access to entities based on the identity of subjects, such as users and processes, to which they belong. The owner can typically transfer permissions to other subjects as well, hence the term discretionary. An example of a discretionary access control is the familiar Unix permission system. In this system, every file defines read, write, and execute permissions for that file's owner, group, and everyone else. Typically, the owner of the file can change these permissions at will. For instance, the owner may grant himself the ability to read and write to a file, give his group read-only access, and not allow users outside the group to access the file at all.

In mandatory access control, the operating system or similar entity constrains access based on a set of rules. In this paradigm, subjects may not transfer access rights if they are restricted from doing so. In other words, it is the system (and the person responsible for maintaining the system), not end users, that controls file system and execution permissions. As an example, the system may divide users into various classes and restrict

users based on class from accessing the files in a given directory and its subdirectories. In this case, if users did not belong to the appropriate class, there is nothing they could do to access the restricted directory, nor could anyone grant them this permission except by changing the system's security configuration, or by moving them to another class (an action which itself would probably be restricted by the security configuration). The ability to configure universal security restrictions in this manner is a powerful tool.

One such mandatory access control is Domain Type Enforcement (DTE) [1]. The two most important concepts in DTE are domains and types. Domains define a security context in which processes operate, while types categorize paths in the file system's directory structure. Access modes, which include reading, writing, executing, and creating files, are restricted from domains to types and also between domains. Badger et al [2] have presented a formal definition for specifying DTE policies known as Domain Type Enforcement Language (DTEL). DTEL allows for the creation of more compact and maintainable security policies than standard type enforcement, which requires the development and maintenance of potentially enormous access control tables [3]. A DTEL policy file concisely describes all of the domains, types, and the access modes that are permitted between them on a given system.

Security is vital not only at the operating system level, but also at the application level. Safe programming languages include facilities for writing secure code. The Java programming language, in particular, is known for having robust security features. One such feature is the Java Security Manager (JSM). Java applications subject to a JSM check with the JSM when performing certain actions, such as file system access, and those actions are then permitted or denied based on the JSM's internal logic. A custom security policy can be implemented by overriding the SecurityManager class and its

relevant methods, then invoking the SecurityManager in the application to be restricted. The SecurityManager is implemented by overriding certain methods which are called whenever a particular action is performed; for example, checkRead and checkWrite [5] [14].

A fundamental similarity exists between DTEL and JSM. Both define security policies that restrict certain kinds of actions – especially file system access – based on explicitly defined criteria. However, they have many differences as well. DTEL's security configuration operates at the operating system level, while JSM operates at the level of an individual application. A system under the purview of a DTEL specification is subject to those restrictions whether it wants to be or not, whereas a Java application must willingly invoke a JSM class to be subject to its restrictions. Furthermore, DTEL security criteria are tightly defined and limited, while a JSM class can determine security restrictions based on any criteria that can be programmatically implemented in its internal logic. In this respect, JSM is more flexible in its ability to define security policies. However, certain limitations in JSM's implementation place considerable restrictions on its ability to implement certain kinds of security checks.

1.2. Motivation

As the design and implementation of good security policies is difficult, so too is teaching students how to understand computer security. To address this challenge, many pedagogical tools have been developed. Specifically as relates to this project, Carr and Mayo [4] have described using DTE to teach students the fundamentals of access control.

The Domain Type Enforcement Language to Java Security Manager Compiler (D2JC) is intended to expand the pedagogical potential of DTE as a learning tool. By using D2JC, students will gain a multifaceted understanding of DTE, the Java Security Manager, and access control in general. D2JC accepts only valid DTE specifications and includes thorough semantic checks, so in order to use the compiler, students must be able to produce well-formed DTE specifications and will be alerted to any semantic errors in their security policies. D2JC outputs Java code specifying a custom SecurityManager class, so in order to use the security policy within the context of a Java application, students must understand how to assign a custom security manager to their application. Finally, by seeing how the security manager interacts with their file system accesses, they will observe how JSM functions and can also test and explore the implementation of their security policy.

Together, it is hoped that these features will provide instructors with a useful tool for teaching the idiosyncrasies of access control, DTE, and JSM to their students.

1.3. Outcome

The current version of D2JC includes a robust parser that is capable of detecting and reporting a wide range of both syntactic and semantic errors in DTEL specifications. Simply by compiling their DTEL specifications, students will learn not only how to create well-formed DTEL syntax, but also the kinds of logical mistakes that might appear in their policies and how to avoid them. If compilation is successful, D2JC outputs code for a valid JSM class that maps certain restrictions (primarily simple file access controls) in the DTEL specification into JSM equivalents. This code can then be compiled and the resulting JSM class can be invoked by Java applications to implement the security

restrictions. This process will teach students about Java compilation, how to invoke the Java Security Manager, and provide the ability to test a subset of DTEL specifications written by the students (or provided to them by instructors).

Additionally, implementation details of the Java Security Manager itself were explored in detail. Some surprising limitations (discussed in detail in subsection 3.2.3) were discovered in the JSM's implementation that prevented a full mapping of DTEL to JSM restrictions. To summarize, it was discovered that the JSM's ability to check the execution of system commands is impaired; the Java FileDescriptor class does not contain enough information to check path-based file system accesses without complex workarounds; and the JSM is subject to its own security restrictions which can throw it into an infinite loop. The opportunity was taken to research potential solutions to these issues. Some of these solutions were partially implemented, but they are not included in the final product because they would hamper cross-platform compatibility. Despite these limitations, D2JC still has considerable value as a pedagogical tool.

Chapter 2. Related Work

2.1. Domain Type Enforcement

The flexibility of DTE has attracted a significant amount of attention in research. Badger et al [2][1] formulated DTEL as an expression of DTE policies and have explored potential applications. Tidswell and Potter [15] proposed a dynamically configurable variant of DTE. Hallyn and Kearns [6] have explored the implementation of DTE in Linux. Kiszka et al [10] applied DTE to a security model divided into real-time and non-real-time components and predicted emerging applications and system responses to expected attacks.

2.2. Security Visualizations

D2JC was built upon an existing DTEL parser. The base code has been used for other projects besides the D2JC, such as DTEvisual by Li et al [12]. Expanding on the pedagogical uses of DTE, DTEvisual accepts a valid DTE specification as input and outputs a graphical representation of the access control policy. DTEvisual is used for educational purposes such as modifying policies during classroom lectures.

Because humans are adept at interpreting data visually, security visualizations have a high potential to improve understanding of security policies and even real-time security events. Recognizing this, other researchers have also developed security visualization tools. Hallyn and Kearns [7] have developed a tool called DTEView to aid the construction of sound DTE policy files through visual representation. Marty [13] describes techniques for using visualization to extract meaningful information from network security logs. Other examples of security visualizations include NVisionCC, a

tool developed by Yurcik et al [16] for visualizing security events on high performance clusters, potentially allowing for much better security maintenance of high-node clusters than traditional command-line tools; and the Intrusion Detection Toolkit by Komlodi et al [11], a visualization tool for detecting intruders on a network.

Chapter 3. The DTEL to JSM Compiler

3.1. The DTEL Parser

D2JC was built upon an existing DTEL parser. The parser was written using the Java Compiler Compiler (JavaCC), “a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar” [8] and converts valid DTEL specifications into Java data structures for other uses such as the DTEvisual tool [12].

In implementing D2JC, several minor bug fixes were applied to the existing code base and semantic error checking was added. If a semantic error is detected, the program terminates compilation and prints an appropriate error message. Errors detected by the semantic checker include the following:

- Multiple types defined with the same name.
- Multiple domains defined with the same name.
- Assigning to a nonexistent type.
- A domain and type sharing the same name.
- The same path is assigned to multiple types.
- No generic type is assigned.
- The initial domain is not a domain.
- Permissions are applied to something that is not a type (e.g., a domain).
- Exec or auto is applied to a non-domain entity.

If the scanner reads the DTEL specification successfully and the parser finds no semantic errors, the program reports that parsing was completed successfully, and compilation is allowed to continue. If the user specified the “-jsm” command line option when invoking

the compiler, the compiler proceeds to convert the DTEL Specification into Java Security Manager code.

3.2. Restrictions and Limitations

There is not a 1:1 mapping between DTE and JSM. DTE is a general-purpose mandatory access control specification, whereas JSM intercepts certain kinds of operations invoked from a Java application and either permits or denies those operations. Because of this, there are some aspects of DTE which have no JSM equivalent, and vice-versa.

3.2.1. DTE-to-JSM Non-Equivalencies

UNIX signals. Because a Java application could potentially be running on any operating system, implementing controls for UNIX signals (e.g., sigkill, sigpause, etc.) would be unnecessarily restrictive and eliminate cross-platform compatibility.

Domain transitions. The DTEL specification allows for controlling domain transitions via auto and exec. This involves the creation of a new process. In order to be meaningful, the new process must be subject to the same security restrictions as the process that spawned it. The JSM clearly cannot enforce its own restrictions on any non-Java processes that are spawned. Even if a new Java process is spawned, potential techniques for transferring the JSM's security restrictions to the new process were deemed unacceptable, as described under subsection 3.2.3, “Limitations of the Java Security Manager.”

3.2.2. JSM-to-DTE Non-Equivalencies

Sockets. The JSM allows security restrictions on a Java application's socket connections. Extensive research did not uncover equivalent restrictions configurable through DTE.

Threads. The JSM allows security restrictions on thread access. DTE allows inter-process restrictions via subject access rights, but no equivalent was found for threads.

Java-specific components. The JSM allows security restrictions on the Java class loader, package access, and properties access. As these are Java-specific security concerns, DTE contains no equivalent.

3.2.3. Limitations of the Java Security Manager

Even among those security concerns which are shared by DTE and the JSM, not all of them could be implemented due to limitations in the JSM's design. These limitations are described below.

Transferring JSM access restrictions. Ordinarily, a Java application must explicitly install a custom security manager in order to be subject to its security restrictions. Initially, the D2JC project assumed that a rough simulation of a complete mandatory access control scheme might be achieved by forcing the initial application to invoke the DTE-specified JSM and then transferring the JSM's security restrictions to any subsequent Java applications invoked by the initial application. This approach assumes that forcing the JSM upon subsequently invoked applications is feasible.

Subsequent research revealed this to be unworkable. It is possible to assign a security manager to a Java application via the command line. The initial approach was to intercept command-line calls initiated by the initial application. Thus, if the initial application launched another Java process with a call such as

```
java ApplicationToInvoke
```

the security manager would intercept that call and replace it with

```
java -Djava.security.manager=SecMgr ApplicationToInvoke
```

thereby transferring its properties to the new application.

However, this approach is thwarted by the fact that JSM's method for checking system calls, `checkExec(String command)`, receives only the first word of the call. Using the previous examples, the parameter `command` would contain the string `"java"`, nothing more. This is insufficient information to apply meaningful security restrictions to system calls.

Due to this limitation, D2JC does not transfer JSM restrictions, nor provide an implementation for the `checkExec` method.

Reading and writing files with `FileDescriptors`. The JSM includes multiple variations of the `checkRead` and `checkWrite` methods, including methods which accept `FileDescriptors` as parameters. This is problematic because the `FileDescriptor` class contains no path information, which DTE requires to perform access checks.

Extensive research revealed a possible workaround for this issue. The `FileDescriptor` class (obtained by downloading the Java source) contains the fields `fd` and `handle`. These are private fields, but they may be accessed using reflection [9], as follows:

```
Field privateField = FileDescriptor.class.getDeclaredField("fd");
privateField.setAccessible(true);
int fd = (int)privateField.get(filedescriptor);
```

Thus armed with the value of the file descriptor, the JSM could invoke an operating system tool such as `lsof` to obtain a list of open files, compare them to the obtained filedescriptor to find the file in question, get the path information, and finally apply DTE restrictions.

This solution was partially implemented before it was deemed too operating system-dependent. The current version of D2JC simply denies all file system accesses attempted with `FileDescriptors`.

The Java Security Manager is subject to its own restrictions. For example, if the JSM attempts to open a file as part of a security check, it calls its own `checkRead` method to see if the access is allowed. Combined with certain other Java design decisions, this has the effect of creating situations where infinite recursive calls of security checks are unavoidable.

In particular, this behavior interferes with the enforcement of file and directory creation permissions. All of the standard Java file output operations work by automatically creating the file being written to (as well as requisite path structure) if it does not already exist. In order to implement file/directory creation checks, it is necessary to first check if the file being written to does not yet exist; and if it does not, to check the relevant permissions.

However, checking for the existence of a file in Java involves creating a new `File` object and then checking for its existence; i.e.,

```
File file = new File(path);
if(!file.exists()) { /* file creation permission check */ }
```

This is a problem because the instantiation of the `File` class causes the security manager to invoke its own security checks, initiating an endless loop which quickly floods the call stack and results in the termination of the application.

This behavior might be avoided by invoking native code and performing the file existence check from there (an option that was explored in some depth), but this would severely hamper cross-platform compatibility, a limitation deemed unacceptable in the implementation of this project.

Because of this behavior, the current version of D2JC is unable to enforce these permissions.

3.3. The DTE to JSM Converter

Although JSM can make only limited use of the DTE specification, D2JC outputs JSM code that contains a complete internal representation of all aspects of DTE which are currently supported by the parser. It also overrides all variations of the `checkWrite` and `checkRead` methods to implement those file system checks which it is able.

The JSM generated by D2JC employs the use of five internal classes for converting the DTE permissions to a usable internal representation. The full code of these classes is given in Listing 3.1.

Listing 3.1. Internal classes of the D2JC-generated JSM.

```
class Permission {
      public ArrayList<String> types;
      public boolean read = false;
      public boolean write = false;
      public boolean exec = false;
      public boolean dir = false;
      public boolean create = false;
}
class Transition {
      public boolean auto = false;
      public boolean exec = false;
      public ArrayList<String> domains;
}
class Domain {
      public String name;
      public ArrayList<String> entryPoints;
      public ArrayList<Permission> permissions;
      public ArrayList<Transition> transitions;
}
class Type {
      public String name;
      public ArrayList<TypeAssignment> assignments;
}
class TypeAssignment {
      public boolean recursive;
      public boolean staticOpt;
      public ArrayList<String> paths;
}
```

When invoked to output JSM code, the compiler uses the information stored by the DTEL parser to generate a constructor that instantiates objects of the classes given in listing 3.1, assigns their values, and ultimately places them in `ArrayLists` of `Domains` and `Types`. It then assigns its own domain as the `initial_domain` defined in the DTEL

specification and determines the current working directory of the application that invoked the JSM.

For executing its file system permission checks, the D2JC-generated JSM implements the following helper methods:

`String convertPath(String path):` Converts Windows paths into Unix paths. Unix paths are returned unaltered.

`String combinePaths(String left, String right):` Extrapolates a single absolute path from the `left` path which is used as the "base" (in practice, the current working directory of the application that invoked the JSM) and the `right` path which is a relative path from the base. It is intelligent enough to parse the `../` character sequence to move up the directory structure of the base path. If the `right` parameter is an absolute instead of relative path (i.e. it is preceded by a slash `/`), the `left` parameter is ignored and the `right` parameter is returned unaltered.

`ArrayList<String> getTypes(String path):` Returns a list of all `Types` that contain the `path` supplied.

`boolean checkPermission(String type, int permission):` Checks if, under the current `Domain`, the given `type` permits `permission`, which is a coded parameter. Values of 0 through 4 correspond to the permissions create, read, write, execute, and directory, respectively.

`boolean filesystemCheck (String type, int permission):` A generalized method that contains code common to all file system checks, called on behalf of the JSM's

`checkRead` and `checkWrite` methods, which supply the relevant `permission` to be checked.

Finally, the D2JC-generated JSM overrides the following SecurityManager methods, which use the helper methods described above:

- `checkDelete(String filename)`
- `checkRead(FileDescriptor filedescriptor)`: Always throws a SecurityException (see subsection 3.2.3).
- `checkRead(String filename)`
- `checkRead(String filename, Object executionContext)`: The `executionContext` is irrelevant to the DTE check and is ignored.
- `checkWrite(FileDescriptor filedescriptor)`: Always throws a SecurityException (see subsection 3.2.3).
- `checkWrite(String filename)`

A more robust implementation was planned and partially implemented, but numerous features were cut from the final version of the project for reasons described in subsection 3.2.3.

Chapter 4. Tests and Performance Analysis

4.1. Semantic Error Checking

In order to demonstrate the abilities of the semantic error checker, the following malformed DTEL specification was created:

Listing 4.1. Malformed DTEL specification.

```
type same_name, same_t, same_t, dup_assign;

domain same_name =      (/sbin/init),
                        (rd->same_d),
                        (auto->same_t);

domain same_d =         (/usr/bin/login),
                        (crwd->same_t),
                        (exec->same_d);

domain same_d =         (/usr/bin/{sh, csh, tcsh}),
                        (crwxd->same_name),
                        (rwd->same_t);

initial_domain = same_t;

assign -r    same_name  /usr/var, /dev, /tmp, /test;
assign -r    same_t     /etc;
assign -r    dup_assign /dev;
assign -r    non_existent /fakepath;
assign -r -s same_t     /dte;
```

The DTEL specification given in Listing 4.1 contains the following semantic errors:

- The initial domain is assigned to a type.
- There is no generic type.

- The name `same_name` is defined for a type and a domain.
- There are two domains called `same_d`.
- There are two types called `same_t`.
- A domain tries to assign `auto` to a type.
- Access permissions are applied to a domain.
- Attempts to assign to a nonexistent type (`non_existent`).
- Attempts to assign the path `/dev` to multiple types.

When attempting to compile the DTEL file, the compiler reports each of these errors:

```
Type 'same_t' has multiple definitions.
Domain 'same_d' has multiple definitions.
Type and domain lists both contain identifier 'same_name'
There is no generic type defined.
initial_domain set to 'same_t' which is not defined as a domain
Permissions tried to reference undefined type 'same_d'
Attempted exec or auto transition to 'same_t' which is not defined
as a domain
Invalid identifier 'non_existent' with assign statement.
Path '/dev' assigned to multiple types
```

Note that each of these errors corresponds to one of the semantic checks described in Section 3.1, so this comprises a thorough test of the semantic checker's ability to detect all of the errors defined for this version of D2JC. Fixing each of these errors yields the DTEL specification given in Listing 4.2.

Listing 4.2. Corrected DTEL specification.

```
type same_name, same_t, diff_t;

domain diff_name =      (/sbin/init),
                        (rd->same_t),
                        (auto->same_d);
```

```
domain same_d =         (/usr/bin/login),
                        (crwd->same_t),
                        (exec->same_d);

domain diff_d =         (/usr/bin/{sh, csh, tcsh}),
                        (crwxd->same_name),
                        (rwd->diff_t);

initial_domain = same_d;

assign -r    same_name  /usr/var, /dev, /tmp, /test;
assign -r    same_t     /;
assign -r -s diff_t     /dte;
```

When the compiler is run on the corrected DTEL specification, it reports no errors and compilation is completed successfully, outputting Java code for a custom security manager.

4.2. File System Permissions

In order to function correctly, the JSM class outputted by D2JC must have the following behaviors:

1. It should be subject to the permission restrictions defined by the DTEL specification for the initial domain.
2. Because the current version of D2JC does not support transitioning to other domains, it should not be subject to permission restrictions for other domains besides the initial domain.
3. It should be able to write to files in those directories defined as writable for the types assigned to the initial domain.

4. It should be able to read from files in those directories defined as writable for the types assigned to the initial domain.
5. It should not be able to read/write from files for which it has not been given permission to do so via type assignment to the initial domain.
6. It should parse both Windows and Unix paths correctly.
7. It should understand that the `../` character sequence in directory paths means to move up in the directory structure.

In order to test the correct functioning of D2JC's file system permissions, the following DTEL specification was created:

Listing 4.3. DTEL specification for testing file system permissions.

```
type generic_t, writable_t, readable_t, both_t, neither_t,
other_t;

domain start_d =  (/sbin/init),
                  (r->readable_t),
                  (w->writable_t),
                  (rw->both_t);

domain unreachable_d = (/fakepath),
                       (rw->other_t);

initial_domain = start_d;

assign -r    generic_t   /;
assign -r    writable_t  /test/writable;
assign -r    readable_t  /test/readable;
assign -r    neither_t   /test/neither;
assign -r    other_t     /test/otherd;
```

```
      assign -r    both_t      /test/both, /test/both2;
```

The DTEL specification in Listing 4.3 defines six types, including a generic type to satisfy DTE requirements. The specification also defines two domains, `start_d` and `unreachable_d`. `start_d` is defined as the initial domain. This domain is given read permission to `readable_t`, write permission to `writable_t`, read and write permission to `both_t`, and no permissions to the other types. `unreachable_d` is given read and write permissions to `other_t`.

To test the JSM class generated when this DTEL file is compiled with D2JC, the following application was created:

Listing 4.4. Security test application to verify the custom JSM's behavior.

```
import java.io.*;

class SecurityTest {

      public static void main(String[] args) {

            // Assign security manager
            try {
                  System.setSecurityManager(new DTESecurityManager());
            } catch (Exception e) {
                  System.out.println("Error: " + e.getMessage());
            }
            System.out.println("Successfully set security manager.");

            // Write to /test/writable
            try {
                  FileWriter fstream = new
FileWriter("M:\\test\\writable\\WriteOut.txt");
                  BufferedWriter out = new BufferedWriter(fstream);
                  out.write("Writing to file.");
```

```
                out.close();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }

            // Read from /test/writable
            try {
                FileReader fstream = new
FileReader("/test/writable/ReadIn.txt");
                BufferedReader in = new BufferedReader(fstream);
                System.out.println(in.readLine());
                in.close();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }

            // Write to /test/writable/deeper
            try {
                FileWriter fstream = new
FileWriter("/test/writable/deeper/../deeper/WriteOut.txt");
                BufferedWriter out = new BufferedWriter(fstream);
                out.write("Writing to file.");
                out.close();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }

            // Read from /test/writable/deeper
            try {
                FileReader fstream = new
FileReader("/test/writable/deeper/ReadIn.txt");
                BufferedReader in = new BufferedReader(fstream);
                System.out.println(in.readLine());
                in.close();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }

            // Similar tests for remaining types
            // (omitted from code listing)
```

```
            System.out.println("Testing complete.");
      }
}
```

The application in Listing 4.4 tries to read from and write to `/test/writable`, and then, to ensure that permissions are being applied recursively (so that paths do not have to be an exact match, but may be prefixes), it tries to read from and write to `/test/writable/deeper`. It performs the same tests with `/test/readable` and `/test/neither`. It then performs the same tests with `/test/otherd, /test/both,` and `/test/both2,` except that the `deeper` checks are omitted for brevity, recursive directories having already been checked by the preceding tests. If any of these operations throws an exception, it catches the exception and prints it to `stdout`. The relevant directories and the `ReadIn.txt` files were created ahead of time for the purposes of the test. The text files contained a single line of text, “I am the first line from the ReadIn file in [path to file].”

This application comprises a thorough test of the required behaviors defined previously. Attempting to read from and write to the various directories defined by the DTEL specification, including directories with only read, only write, and both read and write permissions, verifies requirements (1), (3), (4), and (5). Attempting to read from and write to `/test/otherd`, which the domain `unreachable_d` has permissions to, verifies requirement (2), that other domains' permissions are not being applied to the current domain. The application also includes a Windows-style path, verifying condition (6), and a path that uses the `../` character sequence, verifying condition (7).

Running this test yielded the following output:

```
Successfully set security manager.
Error: Current domain does not have read permission to
/test/writable/ReadIn.txt
Error: Current domain does not have read permission to
/test/writable/deeper/ReadIn.txt
Error: Current domain does not have write permission to
/test/readable/WriteOut.txt
I am the first line from the ReadIn file in /test/readable.
Error: Current domain does not have write permission to
/test/readable/deeper/WriteOut.txt
I am the first line from the ReadIn file in /test/readable/deeper.
Error: Current domain does not have write permission to
/test/neither/WriteOut.txt
Error: Current domain does not have read permission to
/test/neither/ReadIn.txt
Error: Current domain does not have write permission to
/test/neither/deeper/WriteOut.txt
Error: Current domain does not have read permission to
/test/neither/deeper/ReadIn.txt
Error: Current domain does not have write permission to
/test/otherd/WriteOut.txt
Error: Current domain does not have read permission to
/test/otherd/ReadIn.txt
I am the first line from the ReadIn file in /test/both.
I am the first line from the ReadIn file in /test/both2.
Testing complete.
```

Lastly, an examination of the directories revealed that the WriteOut.txt files had been created in `/test/writable`, `/test/writable/deeper`, `/test/both`, and `/test/both2`, and contained the correct text contents, but these files had not been created in the other directories. This is the expected behavior, thus verifying the correct operation of the permission checks.

4.3. Performance Analysis

4.3.1. Compilation Time

In the process of performing semantic error checking, the compiler makes numerous comparisons. A small number of these comparisons occur during the scanning phase and are dependent on the time complexity of the scanner, which was written prior to this project. The remainder of the checks and their time complexities are analyzed below:

A generic type and initial domain are validly specified. These checks both use data gathered during scanning that allows them to be performed in constant time.

Duplicate names in types and domains. To ensure that no type possesses the same name as any domain, the parser compares each domain with each type, resulting in O(dt) complexity where d and t are the numbers of domains and types, respectively.

Duplicate type assignments. To ensure that the same path is not assigned to any two types, the parser compares every path in every type with every path in every other type, resulting in $O(p^2)$ complexity where p is the total number of paths from all types.

Permissions are only applied to types. To ensure that permissions are not applied to a non-type entity, for every set of permissions, the parser checks that permission's target with all types, resulting in O(pt) complexity where p and t are the number of permissions and types, respectively.

Transitions are only applied to domains. To ensure that auto or exec transitions are not applied to a non-domain entity, for every set of transitions, the parser checks that transition's target with all domains, resulting in O(rd) complexity where r and d are the number of transitions and domains, respectively.

All of these checks require either constant or polynomial time, so the semantic error checks added by the D2JC parser add polynomial time complexity to the compilation time of DTEL specifications.

4.3.2. Real-time Permission Checks

Every file system check in D2JC is performed in essentially the same manner. First, the security manager iterates through all of its paths, noting those which match the path of the file being checked and recording their corresponding types. This operation is linear in the number of paths contained in the DTE specification. Then the security manager iterates through all of the types returned in the preceding operation, and for each one, it iterates through all of the permissions defined for the current domain, all of the types assigned to those permissions, and allows the access if the requested operation is allowed for any of the types whose paths correspond to the file being checked. If the security manager completes this entire process without finding any matches, then it denies the access attempt.

The total running time of one check is therefore $O(t_1pt_2) + O(h)$, where t_1 is the number of types to check against, p is the number of permissions in the current domain, t_2 is the number of types in each permission, and h is the number of paths in the DTE specification. Although this is a polynomial-time operation, the values of t_1, p, and t_2 are

likely to be small even in relatively complex DTE specifications, so these checks can be completed quickly in the vast majority of cases.

Chapter 5. Conclusion

5.1. Future Work

The current version of D2JC is limited in the DTEL restrictions it can implement. It could gain even more value as a pedagogical tool if its capabilities were expanded. One approach would be to implement some of the workarounds described in subsection 3.2.3, which were deemed infeasible for this version of the compiler. Different versions of the compiler could be implemented for different operating systems in an effort to preserve cross-platform compatibility.

Alternatively, future iterations of the project could explore alternatives to the Java Security Manager. For example, JSM shares security responsibilities with the access controller and class loader [14]. If D2JC were modified to output not only JSM code, but to utilize additional Java security features, it may be able to achieve a more robust implementation of DTEL specifications.

Another alternative would be to implement a special Java application that acts as a virtual machine specifically for implementing DTE security policies. This virtual machine could implement its own, more powerful version of the security manager, and D2JC could be modified to output code for this customized JSM. This approach would allow for unlimited implementation of DTEL specifications, but implementing the virtual machine might involve a significant amount of work.

5.2. Potential Applications

D2JC's limitations render it inappropriate for industrial use, but it contains many features valuable for pedagogical purposes. It is useful for teaching students how to create well-formed DTEL specifications due to its syntactic and semantic error checking. It also teaches students the basics of incorporating the Java Security Manager into their applications, since the code outputted by D2JC needs to be compiled and installed manually in the application that will make use of it. It also provides an implementation of most of DTE's file permission security checks. However, it must be noted that there are many parts of the DTEL specification that cannot be implemented in the outputted JSM code, so while D2JC has substantial use as a supplement, it is not a complete tool for teaching students how DTE works.

References

[1] Badger L., D.F. Sterne, D.L. Sherman, K.M. Walker, S.A. Haghighat. “A Domain and Type Enforcement UNIX Prototype.” *Proc. Fifth USENIX UNIX Security Symposium* 1995.

[2] Badger L., D.F. Sterne, D.L. Sherman, K.M. Walker, S.A. Haghighat. “Practical Domain and Type Enforcement for UNIX.” *Proc. IEEE Symposium on Security and Privacy* 1995.

[3] Boebert W.E. and R.Y. Kain. “A Practical Alternative to Hierarchical Integrity Policies.” *Proc. 8th National Computer Security Conference* 1985.

[4] Carr S. and J. Mayo. “Teaching Access Control With Domain Type Enforcement.” *Journal of Computing Sciences in Colleges*, 27(1) pp. 74-80, 2011.

[5] “Class SecurityManager.” Oracle Corporation. [Online]. Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/SecurityManager.html

[6] Hallyn S.E. and P. Kearns. “Domain and Type Enforcement for Linux.” *Proc. 4th Annual Linux Showcase & Conference* 2000.

[7] Hallyn S.E. and P. Kearns. “Tools to Administer Domain and Type Enforcement.” *Proc. LISA* 2001, pp. 151-156.

[8] “Java Compiler Compiler (JavaCC) – The Java Parser Generator” [Online]. Available: http://javacc.java.net/

[9] Jenvok, J. “Java Reflection: Private Fields and Methods” [Online]. Available: http://tutorials.jenkov.com/java-reflection/private-fields-and-methods.html

[10] Kiszka J. and B. Wagner. “Domain and Type Enforcement for Real-Time Operating Systems.” *Proc. Emerging Technologies and Factory Automation* 2003.

[11] Komlodi A., P. Rheingans, A. Utkarsha, J.R. Goodall, Joshi A. “A user-centered look at glyph-based security visualization.”*IEEE Workshop on Visualization for Computer Security* 2005.

[12] Li Y., S. Carr, J. Mayo, C.K. Shene, C. Wang. “DTEvisual: A Visualization System for Teaching Access Control Using Domain Type Enforcement.” *Journal of Computing Sciences in Colleges*, 28(1) pp. 125-132, 2012.

[13] Marty R. *Applied Security Visualization.* Addison-Wesley Professional, 2008.

[14] Oaks, S. *Java Security 2nd Edition.* O'Reilly & Associates, 2001.

[15] Tidswell J. and J. Potter. “An Approach to Dynamic Domain and Type Enforcement.” *Information Security and Privacy* 1997.

[16] Yurcik W., X. Meng, N. Kiyanclar. “NVisionCC: A Visualization Framework for High Performance Cluster Security.” *Proc. ACM Workshop on Visualization and Data Mining for Computer Security* 2004, pp. 133-137.