

Integration with existing security model

Matthew Prowse and Stephen Gilmore
LFCS, Edinburgh

Melrose, Scotland, 13th October 2004

Outline

- 1 Integration with existing security model
 - Execution platform and execution policy
 - Producer and consumer relationship
- 2 Overview of packages and classes
 - mrg.javaagent
 - mrg.security
 - mrg.proofchecker
- 3 Supporting software
 - Consumer side: MRGjava command
 - Producer side: makeMRGjar command
- 4 Conclusions and observations

Outline

- 1 Integration with existing security model
 - Execution platform and execution policy
 - Producer and consumer relationship
- 2 Overview of packages and classes
 - mrg.javaagent
 - mrg.security
 - mrg.proofchecker
- 3 Supporting software
 - Consumer side: MRGjava command
 - Producer side: makeMRGjar command
- 4 Conclusions and observations

Workpackage 8b: Integration with existing security model

From the project description:

A prototype implementation of a certificate-led resource manager as outlined in Section 5.1 will be produced. This will provide a platform for further speculative research on developments in static security assessment. Recent work on Java-based agent models which work with the Java 2 security model would provide the basis for further development here.

Execution platform and execution policy

When a `.class` file is downloaded for execution, we wish to verify that its resource usage conforms to the consumer resource policy. In the event that we cannot verify this property, execution should be prevented. Here we present a prototype implementation of the consumer-side component of a Proof-Carrying Code infrastructure, specifically using features of Sun's J2SDK 5.0.

Execution platform and execution policy

When a `.class` file is downloaded for execution, we wish to verify that its resource usage conforms to the consumer resource policy. In the event that we cannot verify this property, execution should be prevented. Here we present a prototype implementation of the consumer-side component of a Proof-Carrying Code infrastructure, specifically using features of Sun's J2SDK 5.0.

- Producer creates a `.jar` containing class files and proof script

Execution platform and execution policy

When a `.class` file is downloaded for execution, we wish to verify that its resource usage conforms to the consumer resource policy. In the event that we cannot verify this property, execution should be prevented. Here we present a prototype implementation of the consumer-side component of a Proof-Carrying Code infrastructure, specifically using features of Sun's J2SDK 5.0.

- Producer creates a `.jar` containing class files and proof script
- Consumer-side codebase packaged in `mrg.jar` (around 16K)

Execution platform and execution policy

When a `.class` file is downloaded for execution, we wish to verify that its resource usage conforms to the consumer resource policy. In the event that we cannot verify this property, execution should be prevented. Here we present a prototype implementation of the consumer-side component of a Proof-Carrying Code infrastructure, specifically using features of Sun's J2SDK 5.0.

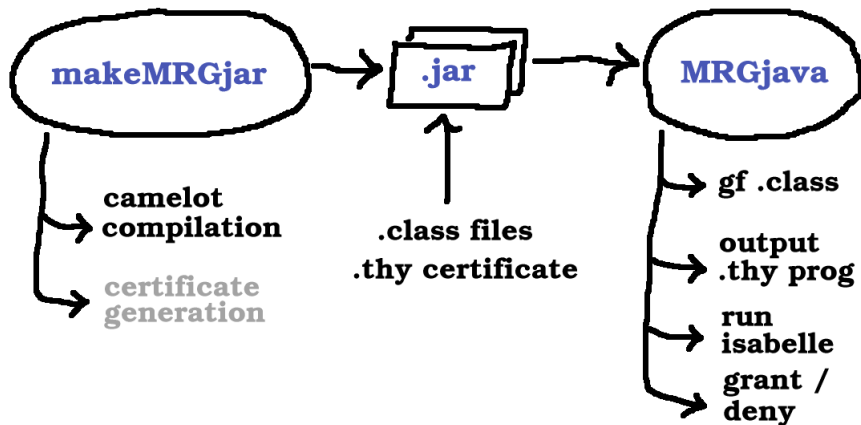
- Producer creates a `.jar` containing class files and proof script
- Consumer-side codebase packaged in `mrg.jar` (around 16K)
- Decompilation, and proof verification performed within a shell script

Execution platform and execution policy

When a `.class` file is downloaded for execution, we wish to verify that its resource usage conforms to the consumer resource policy. In the event that we cannot verify this property, execution should be prevented. Here we present a prototype implementation of the consumer-side component of a Proof-Carrying Code infrastructure, specifically using features of Sun's J2SDK 5.0.

- Producer creates a `.jar` containing class files and proof script
- Consumer-side codebase packaged in `mrg.jar` (around 16K)
- Decompilation, and proof verification performed within a shell script
- Discretionary execution on proof failure

Producer and consumer relationship



Outline

- 1 Integration with existing security model
 - Execution platform and execution policy
 - Producer and consumer relationship
- 2 Overview of packages and classes
 - mrg.javaagent
 - mrg.security
 - mrg.proofchecker
- 3 Supporting software
 - Consumer side: MRGjava command
 - Producer side: makeMRGjar command
- 4 Conclusions and observations

Overview of packages and classes

- `mrg.javaagent`
 - `MRGAgent` — a Java *agent* implementing a `premain` method
 - `MRGTransformer` — an implementation of `ClassFileTransformer`
- `mrg.security`
 - `MRGPermission` — a named subclass of `BasicPermission`
 - `MRGSecurityManager` — handles `MRGPermission` requests
- `mrg.proofchecker`
 - `MRGProofChecker` — invokes a shell script to perform proof checking

MRGAgent

To use our Java *agent*, we specify the following command line switch: `-javaagent:mrg.jar[=options]`.

MRGAgent

To use our Java *agent*, we specify the following command line switch: `-javaagent:mrg.jar[=options]`.

The *manifest* file of `mrg.jar` contains the entry:
`Premain-Class: mrg.javaagent.MRGAgent`

MRGAgent

To use our Java *agent*, we specify the following command line switch: `-javaagent:mrg.jar[=options]`.

The *manifest* file of `mrg.jar` contains the entry:

```
Premain-Class:  mrg.javaagent.MRGAgent
```

On JVM startup, the `premain` method of `MRGAgent` will be called:

```
public static void premain(String arg,  
                           Instrumentation i) { ... }
```

MRGTransformer

During the execution of premain, We make the following call:

```
i.addTransformer(new MRGTransformer());
```

MRGTransformer has a method transform which uses the Byte Code Engineering Library (BCEL) to prepend a security check to the main method of the specified class, before it is constructed in memory and initialised.

MRGPermission and MRGSecurityManager

An MRGPermission is a named subclass of BasicPermission. If code is granted this permission, it has *permission to execute*.

MRGPermission and MRGSecurityManager

An MRGPermission is a named subclass of BasicPermission. If code is granted this permission, it has *permission to execute*.

The MRGSecurityManager class is a subclass of SecurityManager. It overrides the checkPermission method to handle MRGPermission objects. Permission is granted or denied according to the return value of: `MRGProofChecker.check()`;

MRGPermission and MRGSecurityManager

An MRGPermission is a named subclass of BasicPermission. If code is granted this permission, it has *permission to execute*.

The MRGSecurityManager class is a subclass of SecurityManager. It overrides the checkPermission method to handle MRGPermission objects. Permission is granted or denied according to the return value of: MRGProofChecker.check();

To use our Security Manager, we specify the following command line switch:

```
-Djava.security.manager=mrg.security.MRGSecurityManager
```

Inserting a Security Check

When the transform method of MRGTransformer is called with the specified class, the classfile buffer is modified and returned. The security check inserted into the main method is equivalent to:

```
SecurityManager s = System.getSecurityManager();  
if (s == null) throw new SecurityException("...");  
s.checkPermission(new MRGPermission());
```

Inserting a Security Check

When the transform method of MRGTransformer is called with the specified class, the classfile buffer is modified and returned. The security check inserted into the main method is equivalent to:

```
SecurityManager s = System.getSecurityManager();  
if (s == null) throw new SecurityException("...");  
s.checkPermission(new MRGPermission());
```

If the check is successful then execution will continue, otherwise an instance of SecurityException will be thrown.

Inserting a Security Check

When the transform method of MRGTransformer is called with the specified class, the classfile buffer is modified and returned. The security check inserted into the main method is equivalent to:

```
SecurityManager s = System.getSecurityManager();  
if (s == null) throw new SecurityException("...");  
s.checkPermission(new MRGPermission());
```

If the check is successful then execution will continue, otherwise an instance of SecurityException will be thrown.

We insert a number of constant pool entries and bytecode instructions using the BCEL, with little effort.

MRGProofChecker

MRGProofChecker has a `check()` method that is used to initiate the proof verification process by executing a shell script. The boolean return value of the `check` method depends on the exit status of the script.

MRGProofChecker

MRGProofChecker has a `check()` method that is used to initiate the proof verification process by executing a shell script. The boolean return value of the `check` method depends on the exit status of the script.

The shell script is executed using the `Runtime.exec` method and in turn invokes `gf`, `gdf` and `isabelle`. In order to allow execution to complete, we use the `waitFor` method on the process running the script.

MRGProofChecker

MRGProofChecker has a `check()` method that is used to initiate the proof verification process by executing a shell script. The boolean return value of the `check` method depends on the exit status of the script.

The shell script is executed using the `Runtime.exec` method and in turn invokes `gf`, `gdf` and `isabelle`. In order to allow execution to complete, we use the `waitFor` method on the process running the script.

It must be the case that the class file(s) and certificate be output as temporary files before `check()` is invoked, providing input to the external programs launched by the shell script.

Outline

- 1 Integration with existing security model
 - Execution platform and execution policy
 - Producer and consumer relationship
- 2 Overview of packages and classes
 - mrg.javaagent
 - mrg.security
 - mrg.proofchecker
- 3 Supporting software
 - Consumer side: MRGjava command
 - Producer side: makeMRGjar command
- 4 Conclusions and observations

MRGjava script

The shell script MRGjava is an easy way to configure a JVM and check and execute an *MRG* program. It is used as follows:

```
MRGjava [options] <file> [args]
```

where <file> is the path to a .jar file. For example:

```
MRGjava -d ./examples/InsSortWrap/InsSortM.jar 3 2 1
```

The -d option turns on debug messages.

The -p enables an execution profiling report.

makeMRGjar script

The shell script `makeMRGjar` hides the details of creating a `.jar` file compatible with this architecture. It is used as follows:

```
makeMRGjar <name>
```

where `<name>` is the name of a program whose `<name>.cmlt` and `<name>Proof.thy` files are in the current directory.

makeMRGjar script

The shell script `makeMRGjar` hides the details of creating a `.jar` file compatible with this architecture. It is used as follows:

```
makeMRGjar <name>
```

where `<name>` is the name of a program whose `<name>.cmlt` and `<name>Proof.thy` files are in the current directory.

Ongoing work is to modify the MRG tools to produce the `.thy` file syntax expected by this script.

Outline

- 1 Integration with existing security model
 - Execution platform and execution policy
 - Producer and consumer relationship
- 2 Overview of packages and classes
 - mrg.javaagent
 - mrg.security
 - mrg.proofchecker
- 3 Supporting software
 - Consumer side: MRGjava command
 - Producer side: makeMRGjar command
- 4 Conclusions and observations

Conclusions and observations

- Now quite far towards having a JVM back end which is compatible with the Camelot/Grail/Isabelle front end.

Conclusions and observations

- Now quite far towards having a JVM back end which is compatible with the Camelot/Grail/Isabelle front end.
- Instrumenting a piece of code with the additional security check proposed here does incur an overhead, but this is a constant. The time taken to perform the proof is very large in comparison.

Conclusions and observations

- Now quite far towards having a JVM back end which is compatible with the Camelot/Grail/Isabelle front end.
- Instrumenting a piece of code with the additional security check proposed here does incur an overhead, but this is a constant. The time taken to perform the proof is very large in comparison.
- Implementation method uses features of J2SDK 5.0 (“Java 1.5 JVM”) which were not available in earlier JVMs.

Conclusions and observations

- Now quite far towards having a JVM back end which is compatible with the Camelot/Grail/Isabelle front end.
- Instrumenting a piece of code with the additional security check proposed here does incur an overhead, but this is a constant. The time taken to perform the proof is very large in comparison.
- Implementation method uses features of J2SDK 5.0 (“Java 1.5 JVM”) which were not available in earlier JVMs.
- SecurityManager is not available in some smaller JVM implementations.