SLICING MIDLETS

JELTE JANSEN

3rd February 2004

Contents

Chapter 1

Introduction

Development in the field of mobile phones has been going very fast lately. Whereas only a few years ago, it was amazing that one could actually call someone else from about anywhere with such a small device, nowadays there are complete software applications that run on these handy devices. Some people have seen the possibility to use these phones for all kinds of (financial) transactions, partly because they are extremely popular and because it seems that just about anyone owns one these days.

It seems that the standard for such applications is going to be the Java Mobile Information Device Profile (MIDP) 2.0 [5], a profile from Sun for the Java J2ME platform, designed for small, mobile and not very powerful computers.

The Java J2ME platform from Sun consists of several configurations and profiles. A configuration is a virtual machine and a minimal set of class libraries, meant for a range of different devices. Profiles are extra libraries on top of configurations that are targeted for specific machines. For instance, a profile for mobile phones can contain functionality to dial a number, because mobile phones can usually dial numbers.

The MIDP profile is based on the notion that manufacturers of mobile phones should have as much freedom as possible in deciding on the technical specifications of their devices. They only have to make a MIDP compatible virtual machine for them and they can run all MIDP applications.

When drafting this specification, Sun has taken security aspects into account. The security mechanisms of MIDP 2.0 are designed to protect the users of mobile phones from malicious programs. But these security aspects are of considerable interest to other parties involved as well:

- Users
  It is not desirable that an application that has just been downloaded starts to make a call to China without warning, or starts sending out personal data to other parties. Taking into account the possibilities of future mobile phones, like paying for coffee at a coffee machine, one can imagine even worse scenarios. There were 2 main security goals:
  - Prevent unwanted use of services that may cost the user money or leak sensitive information to other parties
  - Prevent any form of harm done by software to the mobile phone or the other software on it.

- Manufacturers
  If there are a lot of low quality applications for a certain kind of phone, it can do a lot of damage to the reputation of the manufacturer of the mobile phone these applications run on, even if the manufacturers have nothing to do with the applications themselves. This is even more so for security flaws in applications. Manufacturers want to be sure that the applications they allow on their phones are not too low on quality (or worse: do damage to the phone or other software on it).
- Developers
  The reputation of software developers can also depend on the security of their applications and the environment those applications run in. It has proved to be very difficult to create 'safe' applications, especially when one does not have much to say about the environment they run in.
- Service providers
  The prime concern for Service Providers are worm-like programs, or other malicious software that may flood or possibly even take down the network.
  Also, if mobile phones start making calls spontaneously, and the reason cannot be traced, there is a big chance that the financial consequences of this will be for the service provider. Such cases must therefore be avoided as much as possible.
- Other
  Not applicable at the moment, but in the case of financial transactions mentioned earlier, banks and other agencies can also be affected by possible security holes.

MIDP 2.0 applications (also known as MIDlets) run in a sandbox within a certain protection domain; depending on their 'state' (for instance trusted or untrusted), they have access to certain API methods of the mobile phone. "The Recommended Security Policy for GSM/UMTS Compliant Devices", an addendum to the MIDP 2.0 specification, provides some advised protection domains (manufacturer, operator, trusted and untrusted) but the manufacturer is free in specifying additional or alternative domains, by defining the domain name and giving lists of methods that should be allowed or denied for that domain. Also, they can specify whether the user of the mobile phone should be asked for permission if certain methods are invoked.

In order for the application to run in a certain domain, it must be digitally signed, by a certified party. This can be the phone manufacturer, but it can also be a independent trusted third party.

Of course, signers of MIDlet suites must be aware of the applicable protection domains and their contents. The MIDP 2.0 specification does not specify a procedure for signing MIDlet suites, but it gives a few examples of procedures. In of these examples, the developer creates and compiles a MIDlet suite, which is later signed by another party. It is also stated that suites should not be signed without caution:

> It should be noted that the signer of the MIDlet suite is responsible to its protection domain root certificate owner for protecting the protection domain stake holders assets and capabilities and, as such, must exercise due-diligence in checking the MIDlet suite before signing it.
>
> *MIDP 2.0 Specification [5] page 31*

Somehow, the signing party has to verify that the MIDlet suite does not abuse the protection domain it will receive; The signing party can demand to have access to the source code of the MIDlet suite so that it can be checked as thoroughly as possible. Verifying compiled binaries is a lot harder than verifying source code (and more so if the compiler uses obfuscation techniques). Of course unknown compilers should not be trusted either, and the signing party might demand to compile the MIDlet suite instead of the developing party.

The main problem addressed in this thesis is how it can be made easier for the signing party to inspect the program.

1.1 Outline

In chapter 2 (page 9), I will give a more detailed description of the problem, and some ways the problem might be approached, as well as some advantages and disadvantages of these approaches.

In chapter 3 (page 13), I will describe some recent work that has connections with this subject. I will mention some other projects that I know of and the way they might be related to the research presented here. I will also give some references from where an interested reader might start to learn more about these topics.

In chapter 4 (page 15), I will introduce the notion of program slicing, the original definition and a very small bit of history about it. I will give some references to more elaborate descriptions and papers. I will give the definition of slicing used in the context of this thesis and I will give the mathematical foundation on which the implementation is built.

In chapter 5 (page 19), I will describe the algorithm that is used to do the actual slicing and the proof-of-concept program PDIM (which stands for Program Domain Invocation Matcher). I will step-by-step describe the process that PDIM uses to create a slice from a given program and a method invocation.

In chapter 6 (page 23), I shall show the use of PDIM on an actual MIDlet. I shall show the various functions that PDIM supports at this moment and I shall show the benefits it has when inspecting J2ME applications.

In chapter 7 (page 31), I shall draw some conclusions based upon the work described in this thesis.

In chapter 8 (page 33), I will discuss possible future work on Java slicing and MIDP inspection. This includes improvement of the current program and expansion of the functionality it gives, as well as research topics that could be based on this and other current and past work.

In appendix A (page 37), I will give some examples of the various dependencies, which are introduced in chapter 4. I will show, in increasing complexity, small programs that demonstrate the slicing algorithm and why and how it works. For each form of dependency used, I will give an example, although some dependencies cannot be shown without the inclusion of others. Each example will be followed by a short description of what has happened in the slicing procedure.

In appendix B (page 63), I will describe the MIDlet that I created in order to show the functionality of PDIM, and why this application might be dangerous to the user.

Chapter 2

Problem description

2.1 Applications on mobile phones

The problem in this case is the protection domain/program matching problem; to get an application (in this context also called a MIDlet) to run in a certain domain, it must be signed. This is a way of using cryptography to give a signature to something. With this, one can verify the origin of an application, who the developer was, and whether the software has been tampered with. It can also be used to show that an application is verified and conforms to certain (security-related) demands.

How can one decide whether an application is suitable for a certain security domain?

Of course, the developer or service provider could sign all of it's own applications, but for the 'higher' domains (i.e. the domains in which applications have more rights) this will probably have to be done by the manufacturers of mobile phones or by independent agencies.

Now suppose that you are from an independent agency and you have to approve an application for a certain security domain. This could be done by extended testing, but one cannot be sure that all possible risks have been taken into consideration. A malicious developer could also easily get around such a method, if he knows the techniques that are used and the demands that are used to test against.

It is also possible to interview the developers about the security aspects of their application. They could guarantee that it is correct and safe. But in practice, this does not give a lot of guarantees, as we can see in the desktop market at the moment [12].

Perhaps someone could develop a tool in which all possible flaws are defined. The tool could search the application source code for possible errors. An example of such a tool is the low-level auditing tool RATS [13], which checks the source code in various languages for the occurrence of common security-related mistakes. The problem with such an application is that it is practically impossible to be complete, and it can contain errors itself.

The source code of the application could also be thoroughly studied by experts. The chance that flaws are found is larger, but the process costs a lot of work, the people involved must have a lot of knowledge, and it can take quite some time. And still, there is not a lot of provable certainty.

With the use of formal methods this last option would give us more certainty. Using formal methods, certain properties of software can be proved. Unfortunately, these methods tend to take even more time and require even more knowledge from the experts than ‘simply’ going through the source code of an application.

What I am looking for in this thesis is a combination of these last options: an automated tool that might help using formal methods to show certain security related properties of applications (in this case the protection domain/program matching problem), or at least make code inspection easier. I will make use of the theory of slicing in PDIM, the program that can slice a MIDlet and show the results.

This method will probably not be complete either; there is a big chance that certain parts of applications will still have to be manually verified by experts. But a (big) part can possibly be verified automatically, so the experts can focus on the more interesting parts. Such a tool should never be used blindly; users should be very well aware of what is does and does not do. It does not replace the necessity of knowledge.

Two of the best known formal techniques used in computer science are Theorem proving and Model checking. In theorem proving, one tries to prove certain statements (formulas) from axioms and inference rules. In Model checking one tries to show that a system (or model of that system) has certain properties specified in some logic (or another model). One advantage of model checking over theorem proving is that it can be a ‘push-button’-technique; it could be possible to make a tool that can completely verify a given model, whereas Theorem proving usually requires a lot of interaction with a user. A big problem in model checking however, is the so-called state space explosion problem.

2.2 State space explosion problem

One of the biggest problem in model-checking is the number of states that a non-trivial system can have; The size of the state-space tends to grow exponentially with the number of variables and processes, which makes (automated) checking of models extremely hard. There are several techniques to address this problem, which fall mostly in the following categories:

- **Abstraction**
  In a lot of cases, the behavior of the system does not rely on the actual value of a certain variable, but on a more general value. For instance, if a program has a variable i, and a control structure like $while(i > 0)$, the behaviour of the while-loop will only depend on the value of the variable being greater than zero or not. In this case, it is possible to reduce the possible values of i in the model to the two abstract values *positive* and *notpositive*.
- **Decomposition**
  If one is only interested in certain properties of the system, it is often possible to reduce the system to a smaller system for which the given property holds. All parts of the system that do not influence the given property, be it directly or indirectly, could be removed, so that a much smaller system is used to actually use in the model-checking process.
- **Symmetry reduction**
  A system that contains a lot of identical components may exhibit symmetries which can be used to reduce the state-space.

In this thesis, I will only use the technique known as Slicing, which falls under decomposition.

The goal is to find out whether slicing is a suitable preparation technique for formal methods or code inspection, by drastically reducing the size of the problem. In this case I will only limit myself to verifying if MIDP applications suit certain protection domains, but I will also see if slicing is applicable to other fields and techniques.

Most slicing techniques have the program source code as their input, because some of the derivations need annotations that a normal compiler does not make, for instance pre-divergence points, see section 4.3.4 on page 17, which are a lot more difficult to separate from 'normal' branches in unannotated bytecode.

The difference between source and bytecode level is twofold:

- The bytecode is the actual program itself, independent on the compiler used to generate it (although not independent on the virtual machine it runs on, but that one is made by the manufacturer of the phone in this case).
  If a third party uses source code for verification, it also has to compile that code itself. It cannot rely on the developer to deliver a compiled version, because different compilers might produce different bytecode representations. This would make the process of checking whether the same source code was used very difficult.
- Developers might be reluctant to deliver source code, as it might contain trade secrets. Although usually the involved parties sign non-disclosure agreements, this could still be a problem.

I will present a technique and implementation that does not require the source code, but works directly on (unannotated) bytecode. This is the main contribution of this thesis, and as far as I know, this has, on the time of writing, not been done before.

Chapter 3

Current work in program analysis and verification

3.1 ESC/Java

ESC/Java [1] is a static checker, that verifies certain properties of Java programs. It is supposed to 'feel' like a type-checker, but uses program verification techniques to detect errors. It is developed by Compaq and contains a annotation language (a subset of JML) that can be used to specify design decisions.

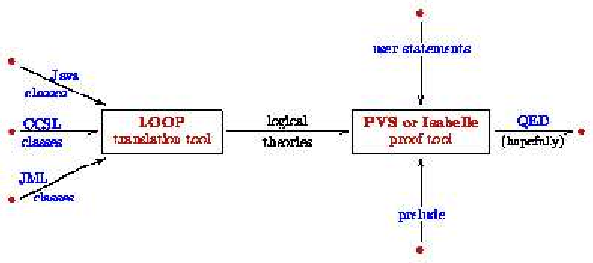
The Security of Systems group has created a second version of this tool, called ESC/Java 2. ESC/Java 2 understands the full JML annotation language. For more information, see [14].

JML is a very detailed language. The specifications written in JML are often as long as the program itself. Usually, the user writes a specification and runs it through ESC/Java, and after analyzing the output, rewrites some parts of the specification or implementation. So, while ESC/Java is a static checker, actual usage is more interactive.

3.2 Loop

Loop [7] (Logic of Object-Oriented Programming) is a tool for theorem-proving. It uses PVS for the verification of JML-specified Properties.

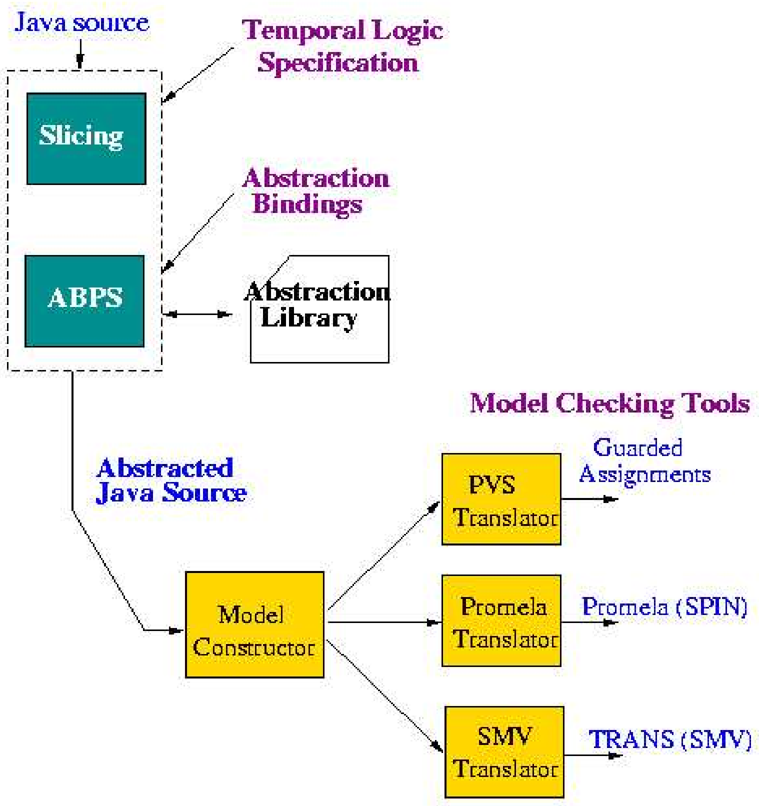
As input it takes Java source code, annotated with JML specifications. It translates this input to several files that describe the meaning of the program in PVS syntax. After that, the user can use PVS to prove that the program conforms to the specification.



3.3 Bandera

Bandera [11] is a complete toolset to model-check Java applications. It uses a number of techniques to overcome the state explosion problem, among which Slicing and Abstraction. It also uses the soot framework, but it contains it's own Compiler (JJJC, the Java to Jimple to Java compiler) and needs source code as input. It can create models for PathFinder, SMV and Spin which can then do the actual model checking.

As input Bandera takes the source code of a Java program and a Linear Temporal Logic specification. After abstracting the source code it constructs a model of the program and then translates it to input for a model checking tool.



Bandera initially seemed like a very good tool to work with, but unfortunately it is not nearly complete, and at the moment is not able to work with programs that use libraries (which almost every Java program does).

For all of these formal approaches it is important that the input (program) is very small. Slicing techniques can contribute to reducing the input, while keeping the relevant elements for the verification at hand. Only Bandera can handle somewhat larger input programs, because it also uses a slicer, as well as some other reduction techniques.

Chapter 4

Program slicing

4.1 Introduction

Program slicing is not a new concept, it was introduced by Mark Weiser [18] as an aid to program debugging. He found that the mental process that programmers made while debugging programs was slicing, and tried to formally define this process [19]. Since then, numerous papers have been published that present different forms of program slicing. A survey can be found in [15].

The original definition is as follows:

> A slice is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point [19].

The slicing used in Bandera is based on the research at SAnToS Laboratory, Kansas state university. The main discussion on this subject is published in [3]. A study of slicing for Java programs (which can be multi-threaded) is presented in [2]. For slicing multi-threaded programs, some extra dependencies have been added. Bandera's slicing algorithm uses annotations made by Bandera's own compiler, and at this moment, Bandera only works on closed programs (i.e. without library calls). The slicing criterion is made by Linear Temporal Logic formulas on certain annotations in the source code of the program.

My slicing algorithm is a form of static slicing (independent on user input), which works directly on Java byte-code, and uses Program Dependency Graphs [9] for the creation of slices. I have included the extra dependencies from [2] for slicing multi-threaded Java programs. I have used a slightly different definition for slicing, which results in a somewhat simpler algorithm and is more appropriate for the goal I had in mind:

> A slice is an executable subset of program statements that preserves the original behavior of the program with respect to the invocations of a user-specified method, including the data used in those invocations.

The slicing algorithm itself does not work directly on the bytecode of a Java program, but uses the Jimple representation of the program. Jimple is a 3-address intermediate representation that is not stack-based, and makes analyses and transformations easier. See [17] for a description of Jimple and the translations from and to Java bytecode.

In this chapter, I will provide a theoretical and mathematical layer on which the algorithm is based. A more detailed description can be found in [2]. The specific slicing technique I use is called Static slicing (i.e. independent on user input) and uses Program Dependency Graphs (PDGs) [9].

4.2 Slicing

With the second definition in 4.1, slicing can also be seen as eliminating those parts of a program that do not have any influence on the reachability of certain 'interesting' statements.

The notion of slicing relies on the assumption that a program contains *dependencies* (see 4.3), i.e. some nodes are dependent on other nodes. For a certain node n, the **slice set** is the set of nodes that n is directly or indirectly dependent on.

Given a **slicing-criterion** C (a set of nodes), compute a slice S that contains the slicing criterion and all nodes on which the nodes in C depend (either directly or indirectly).

The slice set S is constructed using a **Control Flow Graph** (CFG) and a **Program Dependency Graph** (PDG). A CFG is a graph where the nodes represent basic code blocks and the edges represent the flow of the program, i.e. the order in which they are executed. A PDG is a graph where the nodes represent basic code blocks and the edges represent dependencies between these nodes.

Now all the nodes and edges that are not in the PDG are removed from the CFG, resulting in a representation of the sliced program.

4.3 Program dependencies

4.3.1 definitions

In the definitions of program dependencies I will use a number of other definitions, which will be explained here:

For readability, I have chosen to separate the definitions from their examples. For examples of the dependency definitions described here, see Appendix A on page 37.

For more information on the mathematical basis of slicing, see [2].

- A *control flow graph* $G = (N, E, s, e)$ consists of a set of statement nodes N, a set of directed control-flow edges E a unique start-node s and a unique end node e, such that all nodes in N are reachable from s and e is reachable from all nodes in N.
- Node n *dominates* node m in G (notation: $\mathit{dom}(n, m,))$ if every path from the start node to m passes through n.
- Node n *post-dominates* node m in G (notation: $\mathit{postdom}(n, m,))$ if every path from node m to the end node passes through n.

- The function $def(n)$ maps each node to the set of variables defined or modified at node n (the set always contains either zero or one elements). Defined in this case means assigned to, not declared.
- The function $ref(n)$ maps n to the set of variables it references.
- The thread map $\theta(n)$ maps a node n to a thread identifier to which n belongs.
- A synchronization pair $CF(n)$ is the set of inner-most enclosing enter-monitor and exit-monitor statements of statement n (i.e. the statements that make up the synchronization block that contains n).

4.3.2 Data dependence

A node n is data-dependent on a node m if m assigns a value to a variable v that is used by n, and there is a path between both nodes that does not contain any other assignments to v.

A node n is data-dependent on a node m if there is a variable v such that:

1. $v \in def(m) \cap ref(n)$
2. there exists a non-trivial path π from m to n such that for every node $m' \in \pi - \{m, n\}, v \notin def(m')$

4.3.3 Control dependence

A node n is control-dependent on a node m if m branches, and there is one path that passes through n, and one path that does not pass through n.

A node n is control-dependent on a node m if:

1. there exists a non-trivial path π from m to n such that for every node $m' \in \pi - \{m, n\}, m'$ is post-dominated by n
2. m is not post-dominated by n

4.3.4 Divergence dependence

A pre-divergence point is the 'decision-point' of a loop, where the condition is checked to stay in the loop or leave it. A node n is divergence-dependent on a node m if m is a pre-divergence point and m dominates n.

A node n is divergence-dependent on a node m if:

1. m is a pre-divergence point
2. there exists a non-trivial path π such that no node $m' \in \pi - \{m, n\}$ is a pre-divergence point

4.3.5 Interference dependence

A node n is interference-dependent on a node m if both nodes are in different threads and m assigns a value to a variable v that is used by n.

A node n is interference-dependent on a node m if:

1. $\theta(n) \neq \theta(m)$
2. there is a variable v, such that $v \in def(m)\ andv \in ref(n)$

4.3.6 Synchronization dependence

A node n is synchronization-dependent on a node m if m is one of the inner-most enclosing synchronization pairs, i.e. if m is one of the statements that define the synchronization block that contains n.

A node n is synchronization-dependent on a node m if $m \in CR(n)$.

4.3.7 Ready dependence

A node n is ready-dependent on a node m if m is one of the statements that can release a lock that is needed to reach n.

A node n is ready-dependent on a node m if:

1. $\theta(n) = \theta(m)$ and n is reachable from m in the CFG of $\theta(m)$ and $\mathit{code}(m) =$ **enter-monitor** k, or
2. $\theta(n) \neq \theta(m)$ and $\mathit{code}(n) =$ **enter-monitor** k and $\mathit{code}(m) =$ **exit-monitor** k, or
3. $\theta(n) = \theta(m)$ and n is reachable from m and $\mathit{code}(m) =$ **wait** k, or
4. $\theta(n) \neq \theta(m)$ and $\mathit{code}(n) =$ **wait** k and $\mathit{code}(m) \in \{\textbf{notify } k, \textbf{notify-all } k\}$

4.4 Residual program construction

The sliced CFG can be mapped back to byte- or source-code, which finishes the slicing. Some extra measures must be taken to ensure that the resulting CFG represents a valid Java program before mapping back; for instance, if one branch of a conditional is sliced away, while the other is retained, there might be a goto-statement in the result code that points to a non-existent code block. That jump must then be redirected to the correct block. Also, some of the techniques used require a unique end node in the control flow graphs used, which is not necessarily present (for instance, there might be multiple return statements). If a unique end node has been added, it has to be removed again.

Chapter 5

Algorithm and implementation

The general algorithm of PDIM is as follows:

1. **Make a slicing-criterion**
2. **Construct CFG**
3. **Construct PDG**
4. **Construct sliced CFG**
5. **Create residual program**

But first, let me explain a bit about Soot and Jimple.

5.1 On Soot and Jimple

Soot is a Java optimization framework, it uses several intermediate representations of Java bytecode that each have their own advantages for certain optimization of Java bytecode [16]. I will only be using Jimple, a typed 3-address representation, which is more suitable for slicing than the original stack-based bytecode. For more information on how Jimple is constructed and advantages it has over bytecode see [17].

Here is some code that shows the difference between the 3 representations (source, bytecode and jimple representation)

Let's take the very simple source code:

```
public class mini {
        public static void main (String argv[]) {
                int i = 1;
                i = i * 2;
                System.out.println(i);
        }
}
```

When this code is compiled, it produces bytecode, which cannot be read directly, but it can be disassembled:

```
public class mini extends java.lang.Object{
public mini();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

public static void main(java.lang.String[]);
  Code:
   0: iconst_1
   1: istore_1
   2: iload_1
   3: iconst_2
   4: imul
   5: istore_1
   6: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   9: iload_1
   10: invokevirtual #3; //Method java/io/PrintStream.println:(I)V
   13: return

}
```

Here you see how it actually works on the virtual machine level:

first the program puts the the constant 1 on top of the stack and stores the top of the stack on the first free memory position it has (1 in this case). Then it reloads it onto the stack. It also puts the constant 2 on the stack, after which it multiplies the top 2 values on the stack. The result is then stored in memory again. The rest is the invocation of the `println()` method.

This is not very readable, and tends to get extremely complicated when the program is any less trivial. We would like to see the variables reintroduced, whilst staying close to the bytecode level. That's where jimple comes in. The jimple representation of the above bytecode is as follows:

```
public class mini extends java.lang.Object
{

    public void <init>()
    {
        mini r0;

        r0 := @this: mini;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public static void main(java.lang.String[])
    {
        java.lang.String[] r0;
        int i0, i1;
        java.io.PrintStream $r1;

        r0 := @parameter0: java.lang.String[];
        i0 = 1;
        i1 = i0 * 2;
        $r1 = <java.lang.System: java.io.PrintStream out>;
        virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i1);
        return;
    }
}
```

This looks pretty much like the original source code, with some lines that the compiler added automatically. There are some differences though, the most important being:

- Where the original code 'reused' the variable name i, in the jimple representation there are separate variables for the different assignments to i ($i0$ and $i1$).
- When invoking a method, the type of invocation is specified. This is not important in our context, and therefore I shall not discuss it further. Interested minds can always refer to the Java Language Specification [4] and the Java Virtual Machine Specification [6].

5.2 Make a slicing criterion

In the case of MIDlets, the slicing criterion can be any of the calls to one of the methods described in the policy file. For a given method (specified by the user), PDIM searches through the program and marks any statement that calls the given function.

These statements occur in their own methods, which are probably invoked somewhere in the application. PDIM searches for the invocations the way it searched for the invocation of the user-supplied method and adds these to the collection of statements it already has.

This is reiterated until PDIM reaches the entry points of the application, which have no internal invocation. The collection of the statements that have been collected make up the slicing criterion.

5.3 Construct CFG

For each method that is not abstract, the CFG is computed. The control flow grammar creation is handled by the Soot framework. The program just walks through all units in the method and connects them to their successors (i.e. the targets of the goto-statement or the next unit if the unit does not contain a goto-statement).

For more information on the Soot framework, see [16] and [10].

5.4 Construct PDG

Given a statement s, find all statements $\{s_1, \ldots, s_n\}$ so that for all $s_i \in \{s_1, \ldots, s_n\}$, s depends on s_i.

For each statement $s \in \{s_1, \ldots, s_n\}$, repeat the first step and collect all the statements that are found. If statements have already been found, skip those statements.

This algorithm always halts (in the worst case it handles every node once) if the algorithm to find the dependencies halts.

5.5 Construct sliced CFG

From the CFG and the PDG, a sliced CFG is constructed, by walking through all the nodes:

For each node n:

- If n is a goto-statement or a return-statement, leave it in the slice.
- If n is a conditional statement, there are 3 options:
  - if n is not present in the PDG, it can be removed.
  - if n is present in the PDG, but one of its branches is not, replace the jump to that branch with a jump to the convergence node of the branch (the node where the two branches reconnect). If that node does not exists, replace the jump with a jump to the return statement of the program.
  - if n is present in the PDG, and both branches are also present in the PDG, leave n as it is.
- Otherwise, check if n is present in the PDG, if not, remove it.

5.6 Creating residual program

To recreate a valid program from the set of statements gathered above, a few statements have to be added or modified:

- Goto-statements must be retained, unless they point to a statement that is removed, in which case the target of the jump must be modified to the best statement, or the return statement of the method.
- All method declarations are kept in the slice, even if they do not contain any statements (except return), so that the resulting class files keep the global structure of the original program.

5.7 Exporting residual program

If all statements and the relevant structures surrounding them have been identified, the residual program can be created. From this, several actions could be possible, for instance translating the residual program again to a model checking language, or decompiling it to source code. I have only implemented translation back to byte and source code, so actual model-checking should be done after that. PDIM does not contain any interfaces to existing model checkers.

Chapter 6

Example of PDIM using an actual MIDlet

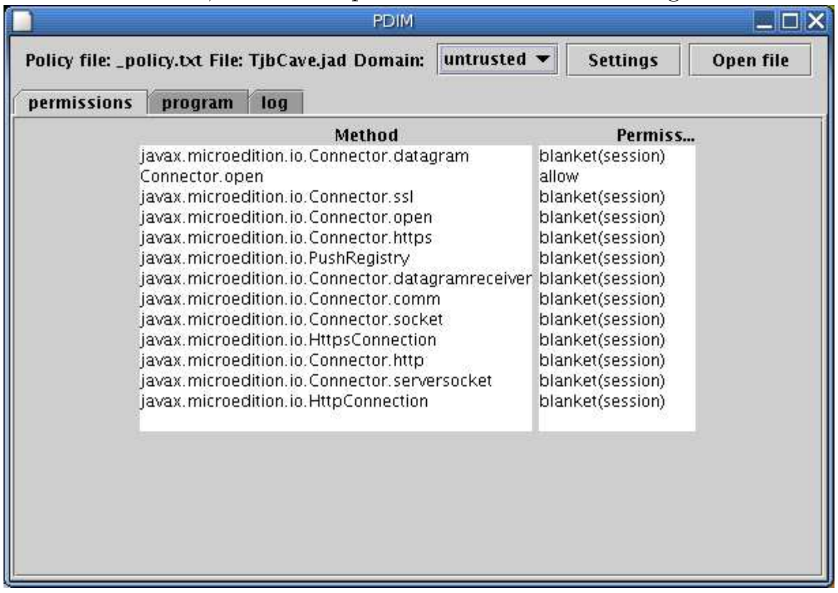
To demonstrate PDIM, I have created a real-world MIDP application, which is in essence a little game, but has some hidden features that could make it unfit for domains as the manufacturers domain. In this chapter, I will show how to detect such potential problems using PDIM. I shall give a quick overview over how the PDIM may be used, after which I will provide a more detailed description of the different windows.

For more information on the MIDlet used, see appendix B on page 63.

6.1 Overview

6.1.1 Permissions screen

When PDIM is started, the user is presented with the following window:



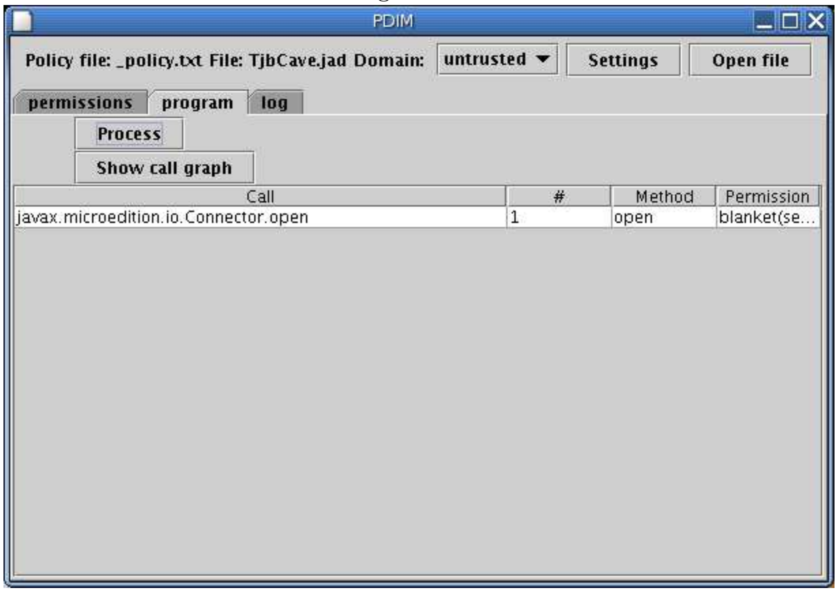
Policy file: _policy.txt File: TjbCave.jad Domain: untrusted

Method	Permiss...
javax.microedition.io.Connector.datagram	blanket(session)
Connector.open	allow
javax.microedition.io.Connector.ssl	blanket(session)
javax.microedition.io.Connector.open	blanket(session)
javax.microedition.io.Connector.https	blanket(session)
javax.microedition.io.PushRegistry	blanket(session)
javax.microedition.io.Connector.datagramreceiver	blanket(session)
javax.microedition.io.Connector.comm	blanket(session)
javax.microedition.io.Connector.socket	blanket(session)
javax.microedition.io.HttpsConnection	blanket(session)
javax.microedition.io.Connector.http	blanket(session)
javax.microedition.io.Connector.serversocket	blanket(session)
javax.microedition.io.HttpConnection	blanket(session)

This screen shows the permissions in the loaded policy file. With the **Domain:** pulldown button the user can select a protection domain to see the permissions of that particular domain, for example untrusted, minimum or maximum.

6.1.2 Program screen

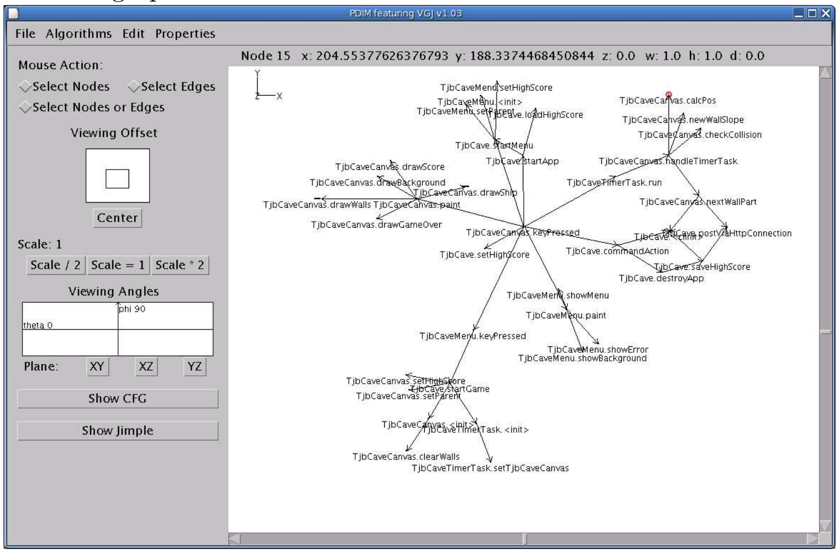
In this screen, the user can view which of the methods from the policy files are called by the application. To see this, the user has to press the button 'Process', after which the screen looks something like this:



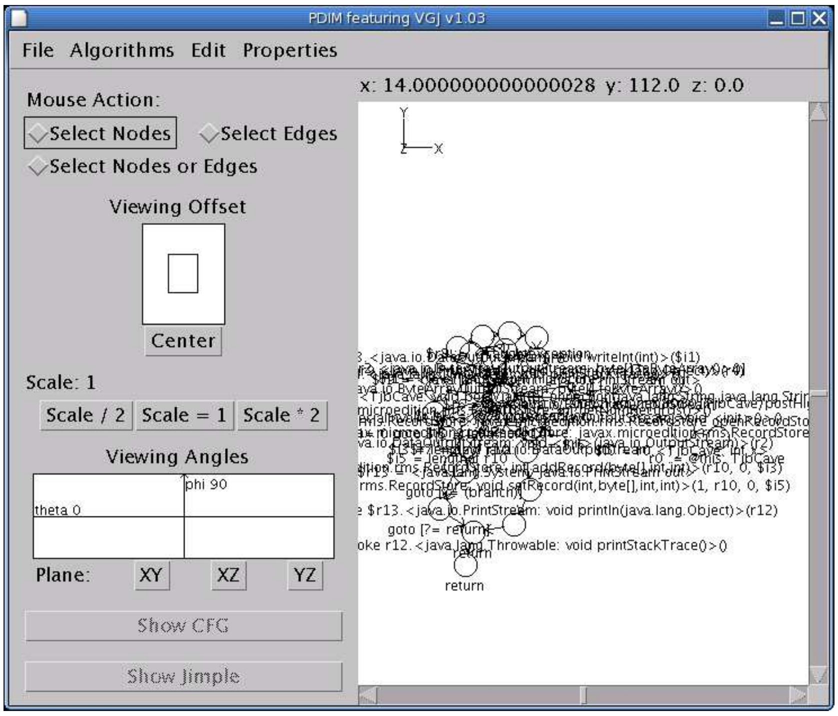
Policy file: _policy.txt File: TjbCave.jad Domain: untrusted

Call	#	Method	Permission
javax.microedition.io.Connector.open	1	open	blanket(se...

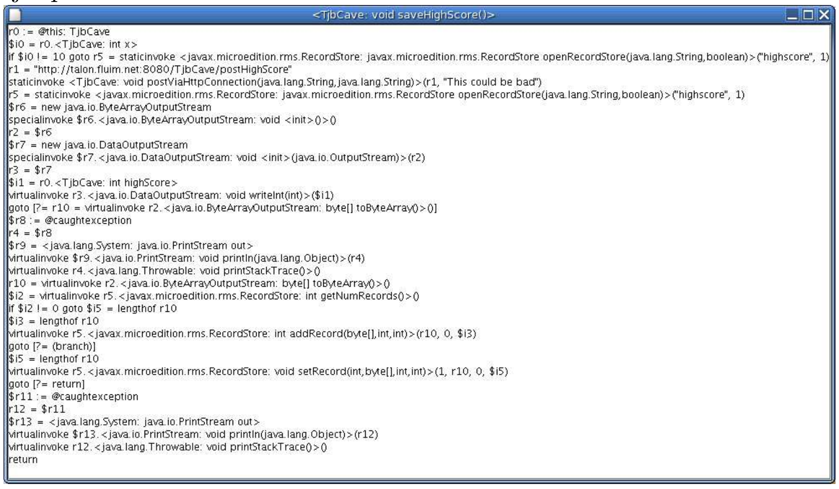
The user can view the method call graph of the complete application by pressing the 'Show call graph' button:



where the user can select a node and press 'show CFG' to get:

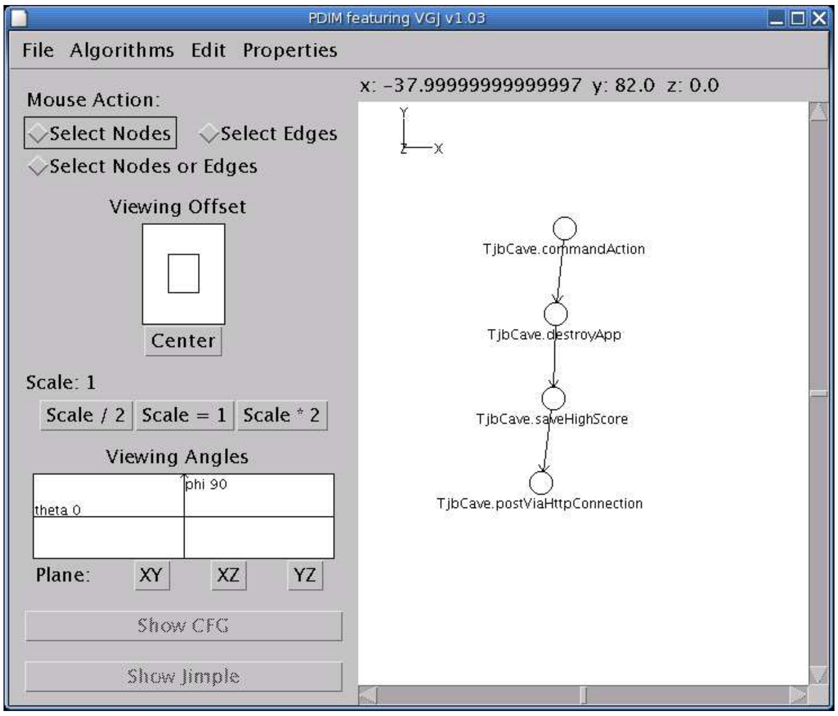


The user can also request the jimple representation of a method by pressing the 'show jimple' button:

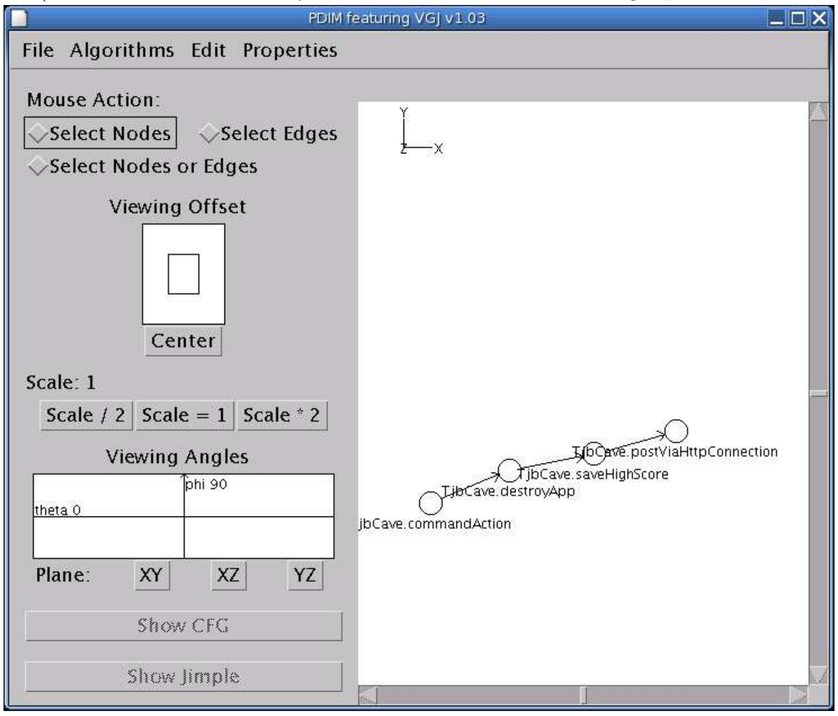


```
r0 := @this: TjbCave
$i0 = r0.<TjbCave: int x>
if $i0 != 10 goto r5 = staticinvoke <javax.microedition.rms.RecordStore: javax.microedition.rms.RecordStore openRecordStore(java.lang.String,boolean)>("highscore", 1)
r1 = "http://talon.fluim.net:8080/TjbCave/postHighScore"
staticinvoke <TjbCave: void postViaHttpConnection(java.lang.String,java.lang.String)>(r1, "This could be bad")
r5 = staticinvoke <javax.microedition.rms.RecordStore: javax.microedition.rms.RecordStore openRecordStore(java.lang.String,boolean)>("highscore", 1)
$r6 = new java.io.ByteArrayOutputStream
specialinvoke $r6.<java.io.ByteArrayOutputStream: void <init>()>()
r2 = $r6
$r7 = new java.io.DataOutputStream
specialinvoke $r7.<java.io.DataOutputStream: void <init>(java.io.OutputStream)>(r2)
r3 = $r7
$i1 = r0.<TjbCave: int highScore>
virtualinvoke r3.<java.io.DataOutputStream: void writeInt(int)>($i1)
goto [?= r10 = virtualinvoke r2..<java.io.ByteArrayOutputStream: byte[] toByteArray()>()]
$r8 := @caughtexception
r4 = $r8
$r9 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r9.<java.io.PrintStream: void println(java.lang.Object)>(r4)
virtualinvoke r4.<java.lang.Throwable: void printStackTrace()>()
r10 = virtualinvoke r2.<java.io.ByteArrayOutputStream: byte[] toByteArray()>()
$i2 = virtualinvoke r5.<javax.microedition.rms.RecordStore: int getNumRecords()>()
if $i2 != 0 goto $i5 = lengthof r10
$i3 = lengthof r10
virtualinvoke r5.<javax.microedition.rms.RecordStore: int addRecord(byte[],int,int)>(r10, 0, $i3)
goto [?= (branch)]
$i5 = lengthof r10
virtualinvoke r5.<javax.microedition.rms.RecordStore: void setRecord(int,byte[],int,int)>(1, r10, 0, $i5)
goto [?= return]
$r11 := @caughtexception
r12 = $r11
$r13 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r13.<java.io.PrintStream: void println(java.lang.Object)>(r12)
virtualinvoke r12.<java.lang.Throwable: void printStackTrace()>()
return
```

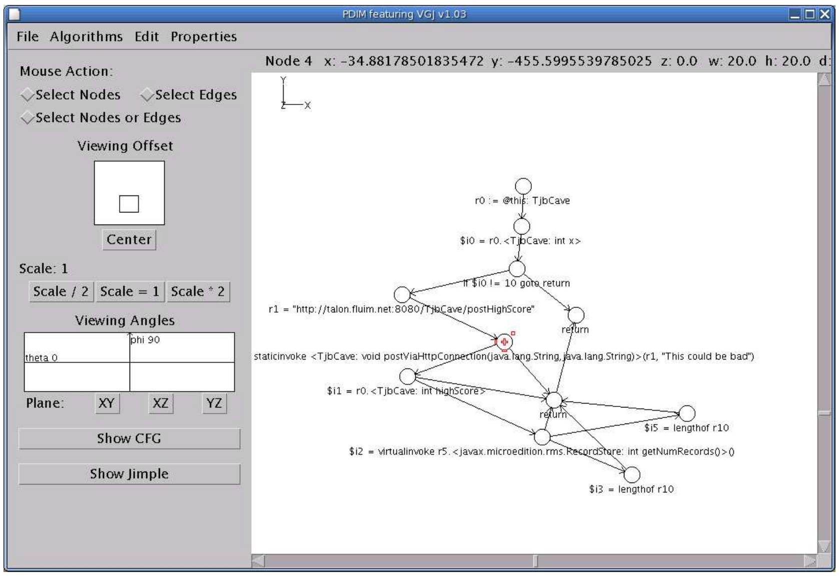
The complete callgraph will usually be too complex to be viewed in a single representation, and therefore it might be more useful to view a call graph to a single selected method (the methods that invoke the selected method, and the methods that invoke those methods etc.):



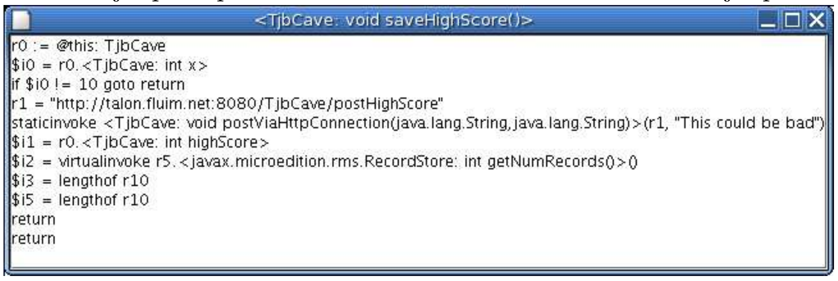
In the program screen, the user can also select a method and choose 'show sliced callgraph', which slices the code to the given method and shows the callgraph to this method (which is of course very much like the unsliced callgraph to this method):



From here, the user can again inspect the control flow of the methods, but this time they are sliced:


PDIM featuring VGJ v1.03
File Algorithms Edit Properties
Mouse Action:
Select Nodes
Select Edges
Select Nodes or Edges
Viewing Offset
Center
Scale: 1
Scale / 2
Scale = 1
Scale * 2
Viewing Angles
phi 90
theta 0
Plane:
XY
XZ
YZ
Show CFG
Show Jimple
Node 4 x: -34.88178501835472 y: -455.5995539785025 z: 0.0 w: 20.0 h: 20.0 d:
r0 := @this: TjbCave
$i0 = r0.<TjbCave: int x>
if $i0 != 10 goto return
r1 = "http://talon.fluim.net:8080/TjbCave/postHighScore"
return
staticinvoke <TjbCave: void postViaHttpConnection(java.lang.String,java.lang.String)>(r1, "This could be bad")
$i1 = r0.<TjbCave: int highScore>
return
$i5 = lengthof r10
$i2 = virtualinvoke r5.<javax.microedition.rms.RecordStore: int getNumRecords()>()
$i3 = lengthof r10

And the sliced jimple representation can be reached with the 'show jimple' button:



```
r0 := @this: TjbCave
$i0 = r0.<TjbCave: int x>
if $i0 != 10 goto return
r1 = "http://talon.fluim.net:8080/TjbCave/postHighScore"
staticinvoke <TjbCave: void postViaHttpConnection(java.lang.String,java.lang.String)>(r1, "This could be bad")
$i1 = r0.<TjbCave: int highScore>
$i2 = virtualinvoke r5.<javax.microedition.rms.RecordStore: int getNumRecords()>()
$i3 = lengthof r10
$i5 = lengthof r10
return
return
```

With some inspection, one can see in the jimple code and in the sliced CFG of the method `saveHighscore()` that the method `postViaHttpConnection` is called if the highscore is higher than a certain value. This is a feature that casual testing might not stumble across, for instance because the value might be so high that only experts (in this game) can achieve it.

6.2 Main window

The main window of PDIM has 3 buttons and 3 tabbed panels:

- Domain button
  This button lets the user select the protection domain that is used to search the program for dangerous method invocations.
- Settings button
  This button calls the Settings window.
- Open file button
  With this button, the user can select another .jad file, which lets the user inspect another application.
- Permission panel

This panel shows the permissions that are defined in the currently selected protection domain of the policy file that has been read.

- Program panel
  This panel contains 2 buttons and a table.
  The Process button processes the current application and searches for invocations of the methods defined in the currently selected protection domain, which it shows in the table below.
  The Show call graph button shows a complete call graph of the unsliced application.
  The table shows methods from the currently selected protection domain that are called somewhere in the application.
  If the user clicks one one of these methods the following menu is shown:
  - Show call graph to this method
    This shows the call graph of the unsliced application to the selected method, i.e. the methods that call the selected method, and the methods that call them and so on.
  - Show complete call graph, sliced to this method
    This shows the complete call graph of the program after it is sliced to the selected method.
  - Show sliced call graph to this method
    This slices the program to the selected method and shows the call graph to that method.
  - Export slice to this method
    This slices the program to the selected method and saves the result in one of the following formats:
    - to source
      Exports Java source code, as generated by Dava, the decompiler of soot.
    - to jimple
      Exports the jimple representation of the Java code.
    - to class file
      This just exports the bytecode of the sliced class files.
- Log panel
  This panel shows a log of the activities of PDIM.

6.3 Call graph Window

The call graph window is a modified version of the VGJ library that visualizes graphs. To the left are various control options:

- Mouse action
  There are 3 selection modes:
  - Select nodes (Default)
    If this is selected, the user can select nodes in the graph.
  - Select edges
    If this is selected, the user can select edges in the graph.
  - Select nodes or edges
    If this is selected, the user can select both nodes and edges in the graph.

If the user has selected one or more nodes, he can move them all by dragging one node. The user can also change the size of nodes. If a single node is selected, the buttons 'Show CFG' and 'Show jimple' are activated.

- Viewing offset
  Here the user can change the part of the image that is viewed in the right pane.
- Scale
  With these buttons the user can zoom in and out of the graph.
- Viewing angles
  Here the user can change the angle in which the graph is viewed in the right panel.
- Show CFG
  This button is only activated if a single node has been selected in the graph. This button opens a window with the control flow graph of the selected method.
- Show Jimple
  This button is only activated if a single node has been selected in the graph. This button opens a window with the jimple representation of the selected method.

6.4 Control Flow Graph window

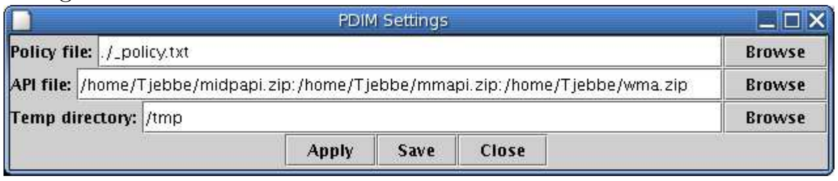
This window is mostly the same as the call graph window, but the 'Show CFG' and 'Show Jimple' buttons do not do anything.

6.5 Jimple Representation window

This window shows the jimple representation of a method, for quick inspection of how the actual Java code might look.

6.6 Settings window

The settings window looks like this:



PDIM Settings

Policy file: ./_policy.txt

API file: /home/Tjebbe/midpapi.zip:/home/Tjebbe/mmapi.zip:/home/Tjebbe/wma.zip

Temp directory: /tmp

The various options are as follows:

- **Policy file**
  Here the user can specify which MIDP policy file to use to find interesting method invocations.

- **API file**
  Here the user can specify one or more (separated by colons, or selected in the browse screen using the ctrl-key) API files, for the MIDP API, which comes with Sun's J2ME Wireless Toolkit [8]. Also, if vendor-specific APIs are used in the application they have to be added here. If the APIs are in the users classpath when starting PDIM these values need not be filled in.
- **Temp directory**
  This is the directory where PDIM creates it's temporary directory for the storage of output files. PDIM tries to create a directory pdim_temp0 in the directory given here. If this directory already exists, it tries to create pdim_temp1, etcetera.

Chapter 7

Conclusion

When inspecting code, the first difficulty is usually to overcome the complex inner structures of non-trivial programs, and separate the important parts from the unimportant ones. Slicing can do a lot of this automatically, thereby reducing the amount of work one has to do to inspect a certain property as well as reducing the chance of mistakes.

The PDIM application I described in chapter 6 is a proof-of-concept application which can actually help to make the visualization of a program more comprehensible, and slice the program to the interesting statements.

I have also shown that slicing of Java programs is possible without the original source code of the application.

The example mentioned is, after slicing, reduced from 697 lines of code to 290, and those lines include all method declarations (which are all contained), while the complete bytecode is reduced from a jar-file of 6.2 kilobyte to 2.6 kilobyte, so slicing a non-trivial Java program significantly reduces the size of the program. Of course, the amount of reduction can greatly vary for different applications.

Although PDIM is not yet the complete push-button tool to verify whether an application fits in a certain MIDP protection domain, because it still requires code inspection, I believe that I did make certain advancements in the field of Java slicing and visualization, and PDIM shows that slicing can make the program domain matching problem a lot easier.

Chapter 8

Future work

During the making of this thesis and the accompanying tool PDIM, I came across several points that could be addressed in the future:

- Improvements of the algorithm
  The slicing algorithm is pretty lenient; in order to be certain that the result is still a valid program, it keeps some statements that could be sliced away. On the other hand, some parts of the program are sliced away when they could be of interest, for instance, with the current algorithm, if one is interested in the invocation of the method `javax.microedition.io.Connector.open()` it also slices away the statements that actually do something with the opened connection. An algorithm that uses some form of forward dependencies would allow a user to see what happens with this connection after is has been opened.
- Improvements of PDIM
  The implementation of the slicing algorithm can also be improved; as a proof of concept, PDIM is not created with efficiency in mind, and especially some of the graph searching algorithms used could use some improvements.
  Also, PDIM currently leaves all the methods of the original program, even if they are empty and could be sliced away. An intelligent algorithm to check if these methods cannot be removed would greatly reduce the size of the sliced source code.
  Finally, the graphical user interface of PDIM could use some work, it contains some small bugs, and could probably be implemented more user-friendly.
- Improvements of the underlying libraries
  The soot framework, although very powerful, is still not able to successfully decompile every program. For instance, decompilation of nested try-catch statements sometimes causes errors. It also contains some other bugs, but it is very actively maintained. Check out the soot web site for updates.

Appendices

Appendix A

Examples

In this appendix I will use the slicing program to show how slicing works.

The first examples will only be for very trivial programs, mainly to show how dependencies are generated and what the connection to the actual code is.

For these simple examples I shall first give the Java program code, then the CFG for each method in the code, then the program dependency graph(s), and finally the sliced code, decompiled with Dava.

I will slice all the simple ones to the slicing 'criterion' java.io.PrintStream.println.

In the dependency graphs, the dependencies are colored as follows:

- Control flow arrows are colored in black
- Data-dependencies are colored in blue
- Control-dependencies are colored in red
- Divergence-dependencies are colored in yellow
- Interference-dependencies are colored in purple
- Synchronization-dependencies are colored in lightblue
- Ready-dependencies are colored in green
- Extra dependencies are colored in black

A.1 Data dependency

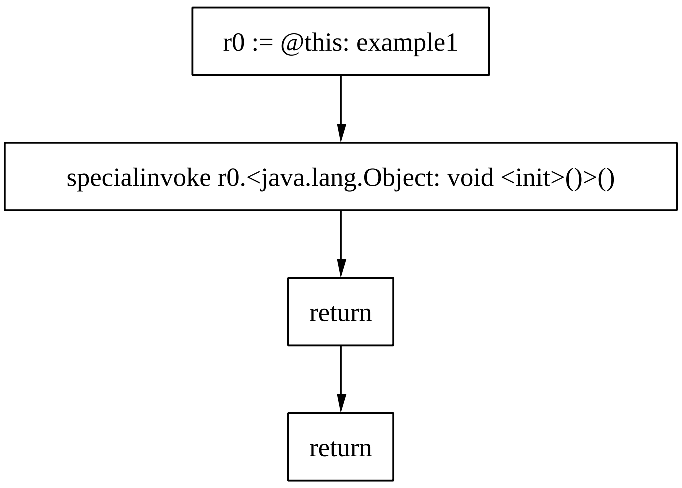
Here I will show a simple example of data dependency. The program declares 2 variables and assigns some values to them. Then it will print one of the variables.

A.1.1 Code

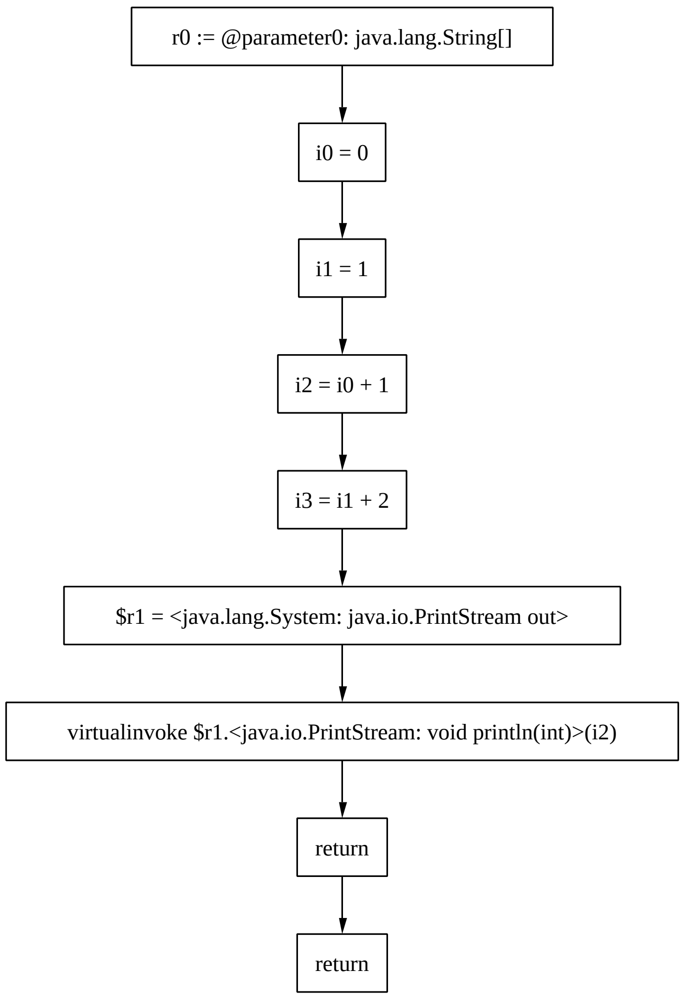
example1.java

```
import java.io.*;

public class example1
{

    public static void main(java.lang.String[] r0)
    {
                int a = 0;
                int b = 1;
                a = a + 1;
                b = b + 2;

                System.out.println(a);
    }
}
```

A.1.2 Control flow graph



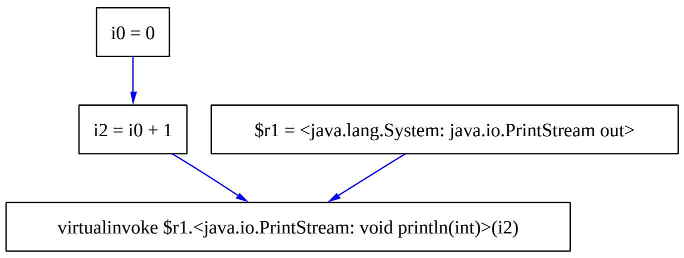
Control flow graph for method <init>


r0 := @parameter0: java.lang.String[]
i0 = 0
i1 = 1
i2 = i0 + 1
i3 = i1 + 2
$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i2)
return
return

Control flow graph for method main

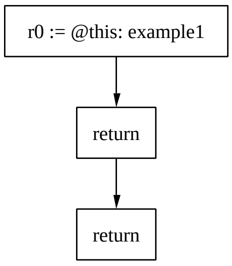
A.1.3 Program Dependency graph

dependency graph for method <init>

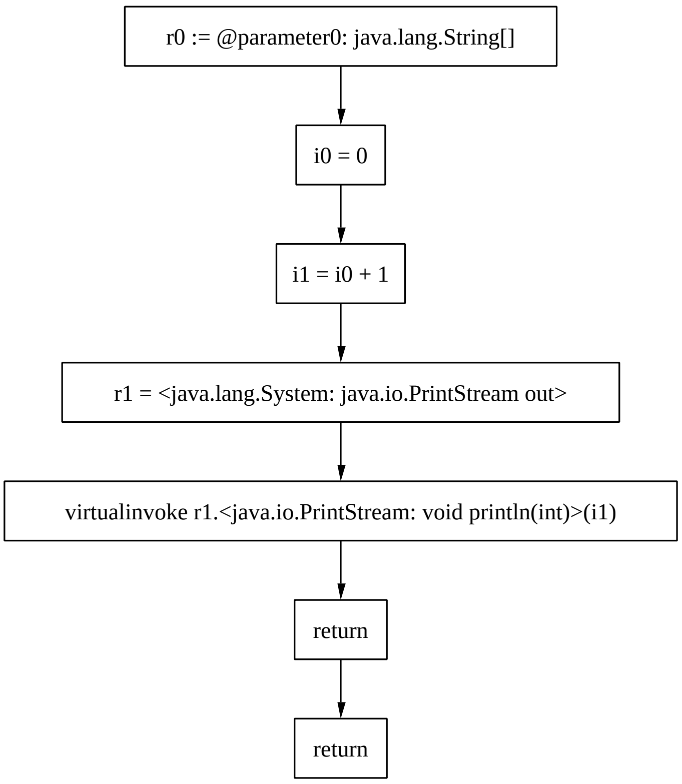

i0 = 0
i2 = i0 + 1
$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i2)

dependency graph for method main

A.1.4 Sliced control flow graph



Sliced control flow graph for method <init>


r0 := @parameter0: java.lang.String[]
i0 = 0
i1 = i0 + 1
r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke r1.<java.io.PrintStream: void println(int)>(i1)
return
return

Sliced control flow graph for method main

A.1.5 Sliced code

example1.java

```
import java.io.*;

public class example1
{

    public example1()
    {
        return;
    }

    public static void main(java.lang.String[] r0)
    {
        int i0, i1;
        java.io.PrintStream r1;

        i0 = 0;
        i1 = i0 + 1;
        r1 = System.out;
        r1.println(i1);
        return;
    }
}
```

A.1.6 Notes

In the PDG (A.1.3) we see that the statement
`virtualinvoke r1.<java.io.PrintStream: void println(int)> (i3)`
which stands for the `println` method in the original Java code, is data-dependent on `r1`, the stream to print to, and `i2`, the variable to print. `i2` is dependent on the assignment to `i0`.

As you can see, the variables `i1` and `i3` (which stand for the variable `b` in the source code), do not appear in the dependency graph, because they do not have any influence on the execution of the `println()` method. Since that is the method that we are interested in here the assignments to `i1` and `i3` (and, accordingly, `b`) can be removed, which leaves us with the code in A.1.5.

A.2 Control dependency

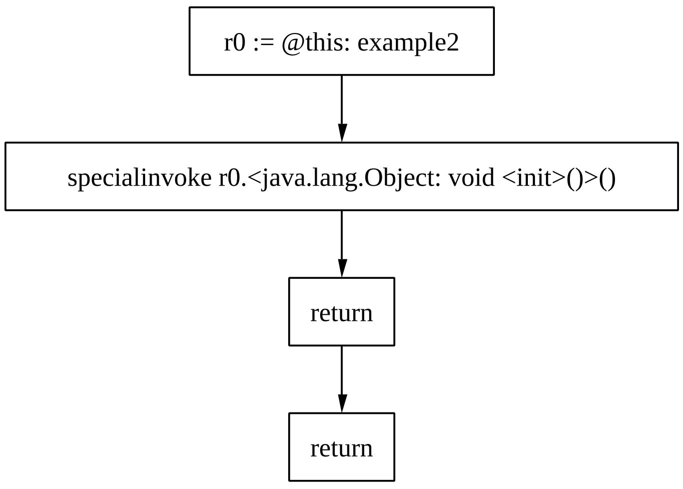
In this example, the program again declares 2 variables and assigns values to them. Depending on the value of variable b, which is the number of arguments the program is called with, it either prints that number or assigns an other value to a.

A.2.1 Code

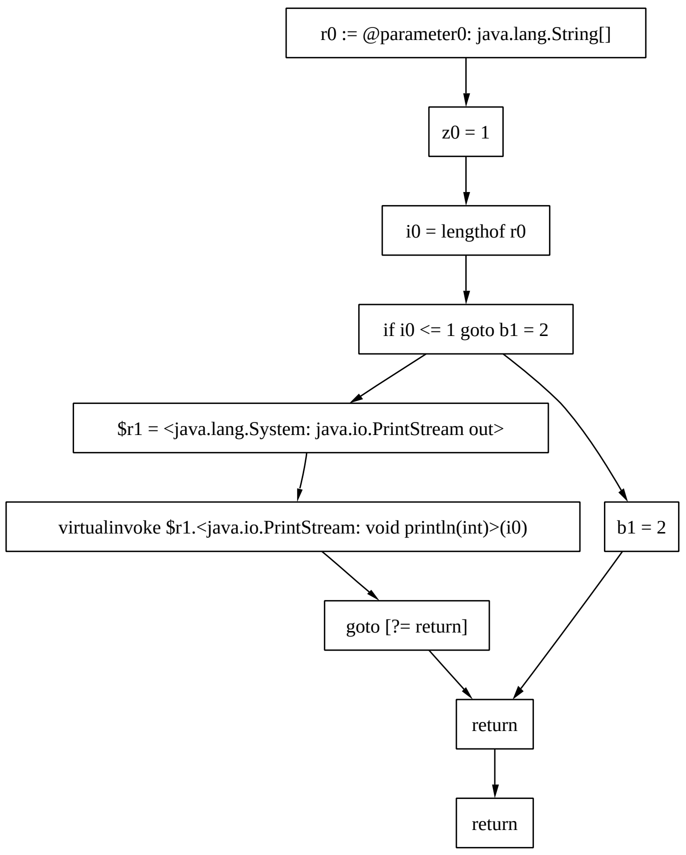
example2.java

```
import java.io.*;

public class example2
{

    public static void Main(String argv[]) {

    int a = 1;
    int b = argv.length;

        if (b > 1)
        {
            System.out.println(b);
        } else {
            a = 2;
        }
    }
}
```

A.2.2 Control flow graph

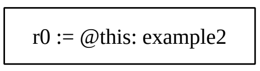


Control flow graph for method <init>

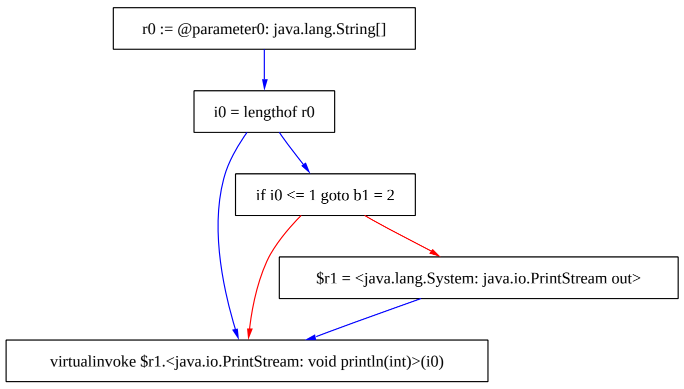

r0 := @parameter0: java.lang.String[]
z0 = 1
i0 = lengthof r0
if i0 <= 1 goto b1 = 2
$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i0)
b1 = 2
goto [?= return]
return
return

Control flow graph for method Main

A.2.3 Program Dependency graph

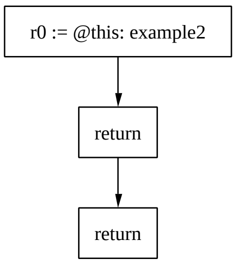


dependency graph for method <init>

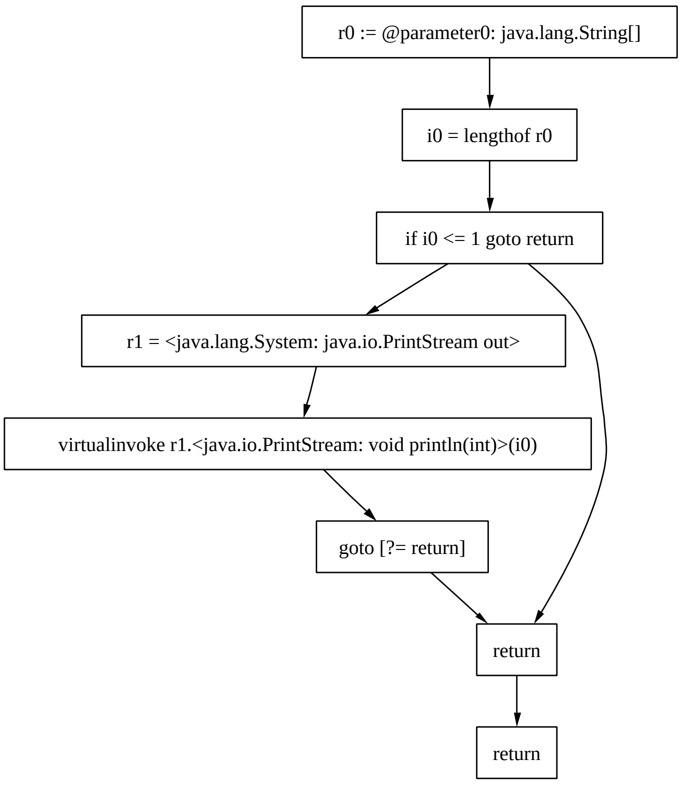

r0 := @parameter0: java.lang.String[]
i0 = lengthof r0
if i0 <= 1 goto b1 = 2
$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i0)

dependency graph for method Main

A.2.4 Sliced control flow graph



Sliced control flow graph for method <init>


r0 := @parameter0: java.lang.String[]
i0 = lengthof r0
if i0 <= 1 goto return
r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke r1.<java.io.PrintStream: void println(int)>(i0)
goto [?= return]
return
return

Sliced control flow graph for method Main

A.2.5 Sliced code

example2.java

```
import java.io.*;

public class example2
{

    public example2()
    {
        return;
    }

    public static void Main(java.lang.String[] r0)
    {
        java.io.PrintStream r1;
        int i0;

        i0 = r0.length;

        if (i0 > 1)
        {
            r1 = System.out;
            r1.println(i0);
        }

        return;
    }
}
```

A.2.6 Notes

In the PDG (A.2.3), we see that the print statement is now dependent on `if i0 <= 1 goto return`, as well as the assignment to the variables it uses (like in A.1). Since both the other branch of the if-statement and the statements that use and define a are not in the PDG, they can be sliced away. Note that a more advanced program might change the if-statement to an assert statement or something similar, see also chapter 8 on page 33.

A.3 Divergence dependency

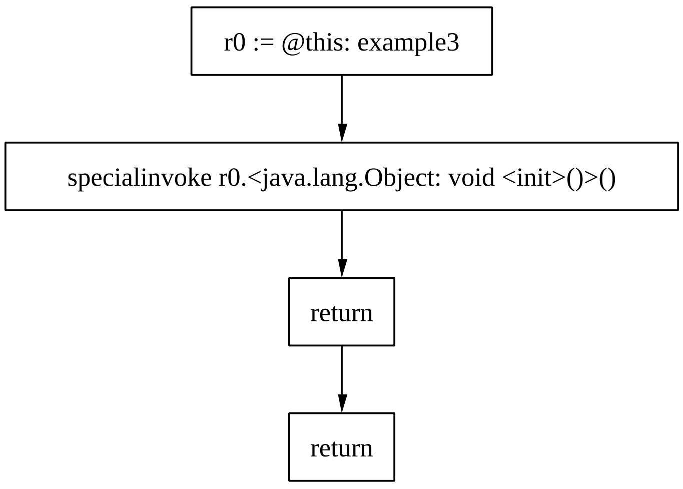
In this example the program first performs a small while loop, and then prints ”done”.

A.3.1 Code

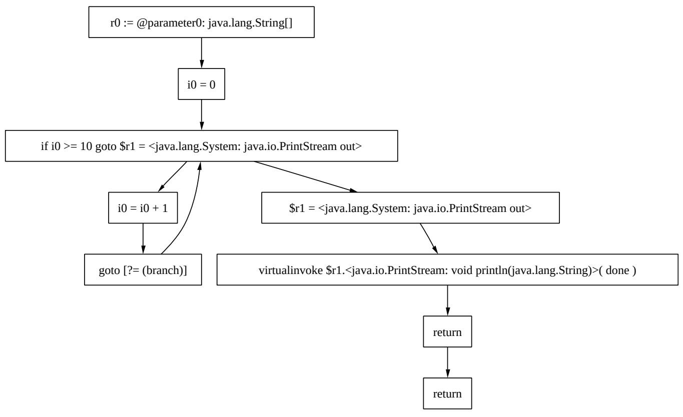
example3.java

```
import java.io.*;

public class example3
{

    public static void main(java.lang.String[] r0)
    {
        int a = 0;
        while (a < 10) {
            a = a + 1;
        }

        System.out.println("done");

    }
}
```

A.3.2 Control flow graph

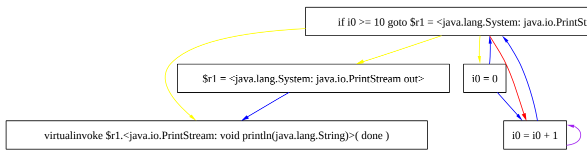


Control flow graph for method <init>

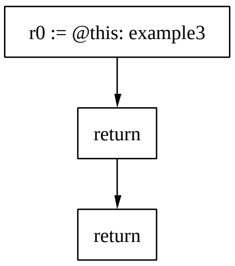

r0 := @parameter0: java.lang.String[]
i0 = 0
if i0 >= 10 goto $r1 = <java.lang.System: java.io.PrintStream out>
i0 = i0 + 1
$r1 = <java.lang.System: java.io.PrintStream out>
goto [?= (branch)]
virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>(done)
return
return
Control flow graph for method main

A.3.3 Program Dependency graph

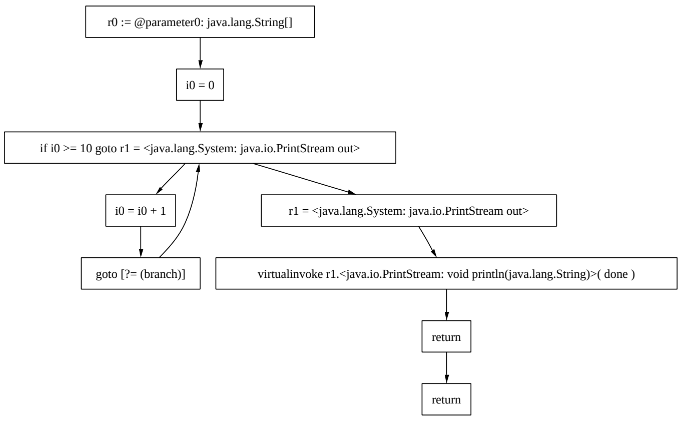
dependency graph for method <init>


if i0 >= 10 goto $r1 = <java.lang.System: java.io.PrintStream out>
$r1 = <java.lang.System: java.io.PrintStream out>
i0 = 0
virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>(done)
i0 = i0 + 1
dependency graph for method main

A.3.4 Sliced control flow graph



Sliced control flow graph for method <init>


r0 := @parameter0: java.lang.String[]
i0 = 0
if i0 >= 10 goto r1 = <java.lang.System: java.io.PrintStream out>
i0 = i0 + 1
r1 = <java.lang.System: java.io.PrintStream out>
goto [?= (branch)]
virtualinvoke r1.<java.io.PrintStream: void println(java.lang.String)>(done)
return
return

Sliced control flow graph for method main

A.3.5 Sliced code

example3.java

```
import java.io.*;

public class example3
{

    public example3()
    {
        return;
    }

    public static void main(java.lang.String[] r0)
    {
        int i0;
        java.io.PrintStream r1;

        i0 = 0;

        while (i0 < 10)
        {
            i0 = i0 + 1;
        }

        r1 = System.out;
        r1.println("done");
        return;
    }
}
```

A.3.6 Notes

While the print statement only prints the string ”done”, and thus has no data or control dependencies in the rest of the program, it is dependent on the while loop; in order for the print statement to execute, the while loop must be finished. Therefore, the while loop cannot be sliced away.

A.4 Interference dependency

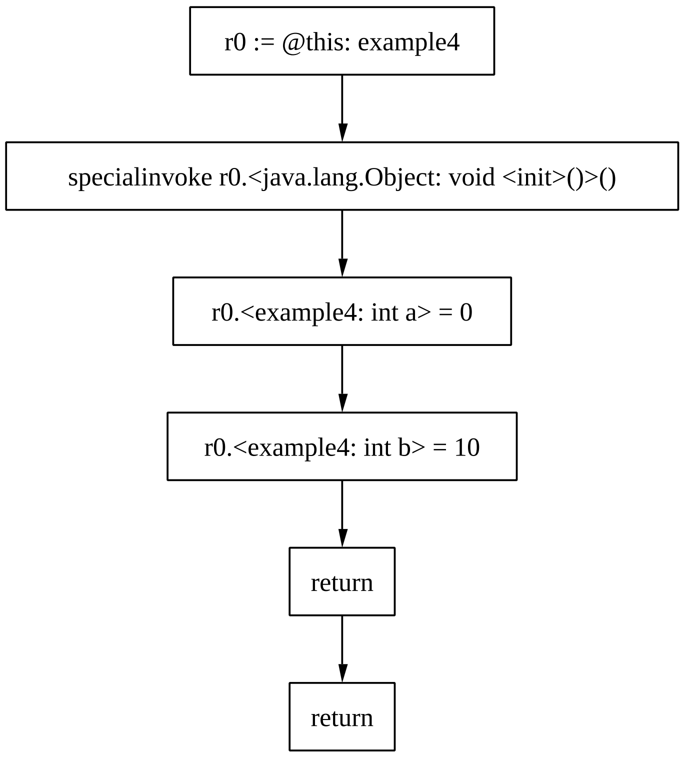
This example will show dependencies between different methods. Because of the current way dependency graphs are represented, A.4.3 will show all the nodes that have dependencies, but not the actual connections.

A.4.1 Code

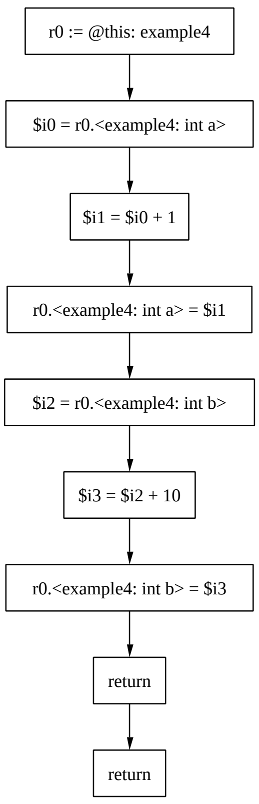
example4.java

```
public class example4
{

    int a = 0;
    int b = 10;

    public void doSomething() {
        a = a + 1;
        b = b + 10;
    }

    public void doSomethingElse() {
        System.out.println(a);
    }

}
```

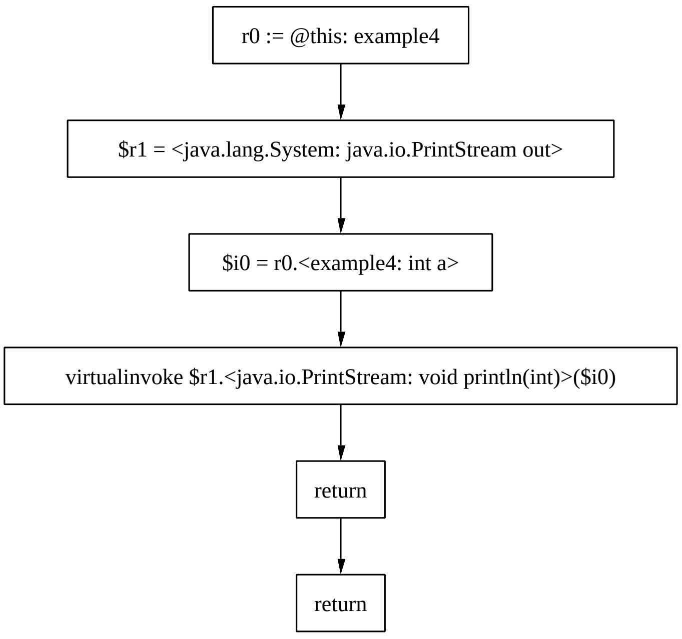
A.4.2 Control flow graph


r0 := @this: example4
specialinvoke r0.<java.lang.Object: void <init>()>()
r0.<example4: int a> = 0
r0.<example4: int b> = 10
return
return

Control flow graph for method <init>


r0 := @this: example4
$i0 = r0.<example4: int a>
$i1 = $i0 + 1
r0.<example4: int a> = $i1
$i2 = r0.<example4: int b>
$i3 = $i2 + 10
r0.<example4: int b> = $i3
return
return

Control flow graph for method doSomething

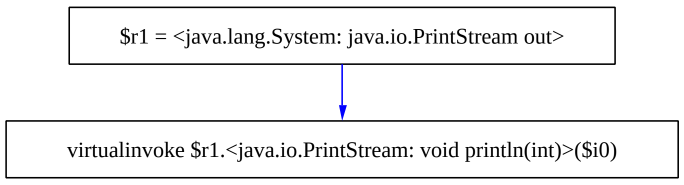

r0 := @this: example4
$r1 = <java.lang.System: java.io.PrintStream out>
$i0 = r0.<example4: int a>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>($i0)
return
return

Control flow graph for method doSomethingElse

A.4.3 Program Dependency graph

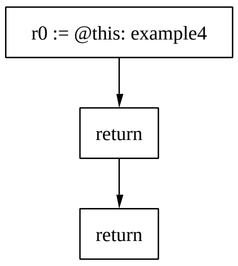
dependency graph for method <init>

dependency graph for method doSomething

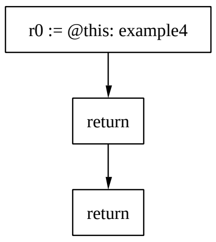

$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(int)>($i0)

dependency graph for method doSomethingElse

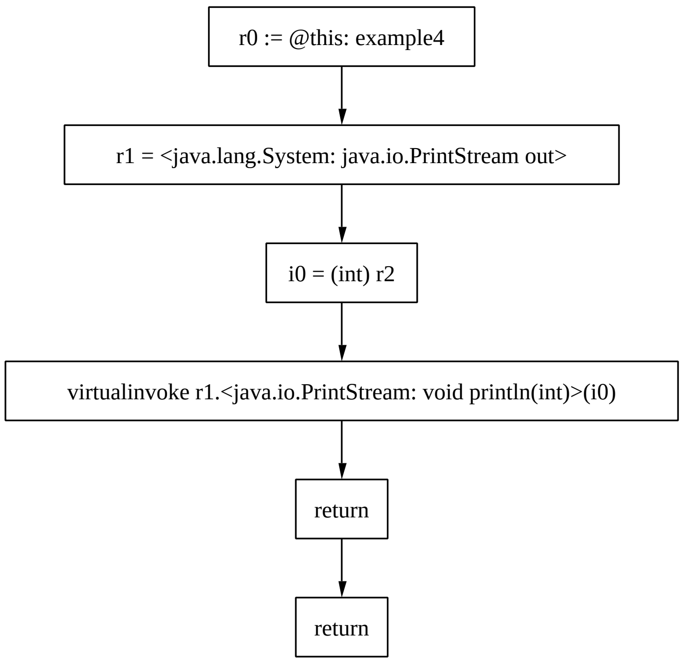
A.4.4 Sliced control flow graph



Sliced control flow graph for method <init>



Sliced control flow graph for method doSomething


r0 := @this: example4
r1 = <java.lang.System: java.io.PrintStream out>
i0 = (int) r2
virtualinvoke r1.<java.io.PrintStream: void println(int)>(i0)
return
return

Sliced control flow graph for method doSomethingElse

A.4.5 Sliced code

example4.java

```
import java.io.*;

public class example4
{

    public example4()
    {
        return;
    }

    public void doSomething()
    {
        return;
    }

    public void doSomethingElse()
    {
        java.io.PrintStream r1;
        java.lang.Object r2;
        int i0;

        r1 = System.out;
        i0 = (int) r2;
        r1.println(i0);
        return;
    }
}
```

A.4.6 Notes

As you can see, because the variable a is printed in the method `doSomethingElse()`, the statements in method `doSomething()` that influence a cannot be sliced away. b is not used and thus can be safely removed from the program.

A.5 Synchronization and ready dependency

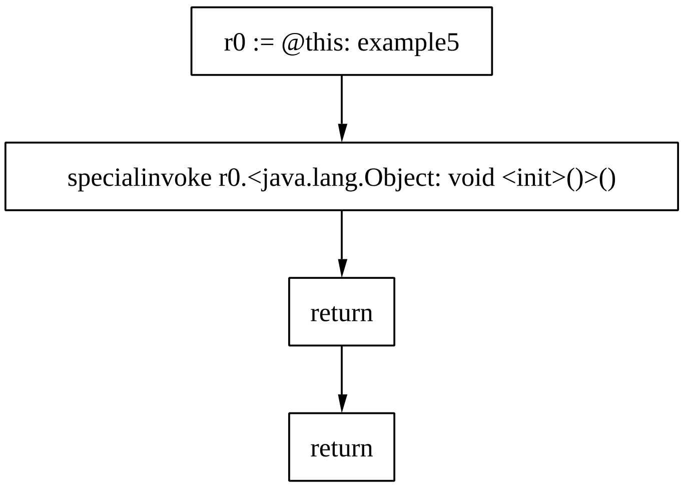
For this example, I created a small method that contains a synchronized block, in which some class variables are changed, this example both shows synchronization and ready dependencies.

A.5.1 Code

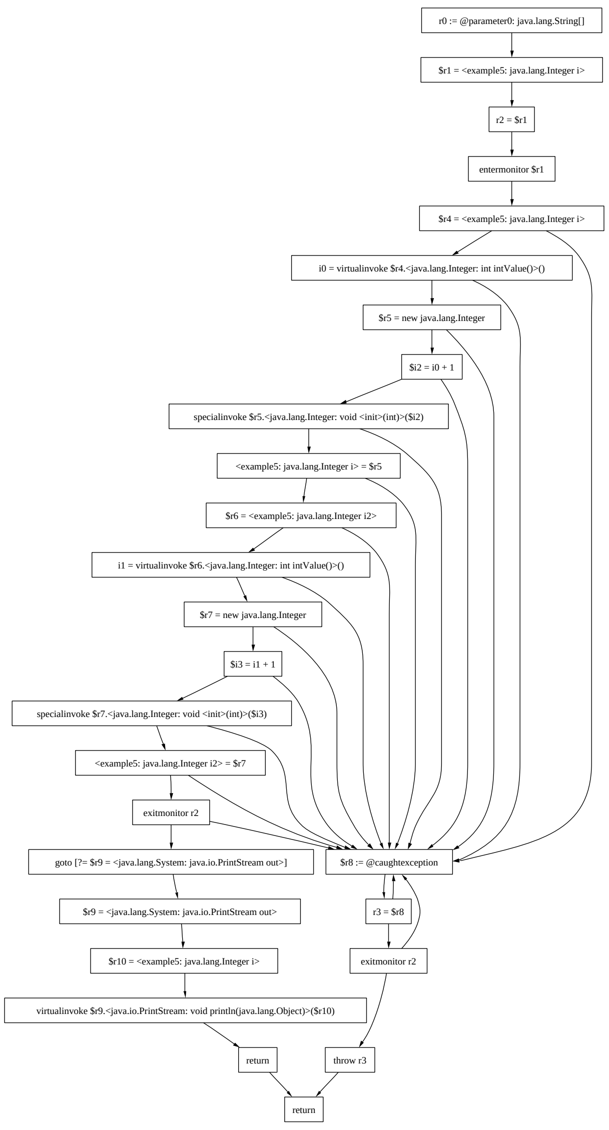
example5.java

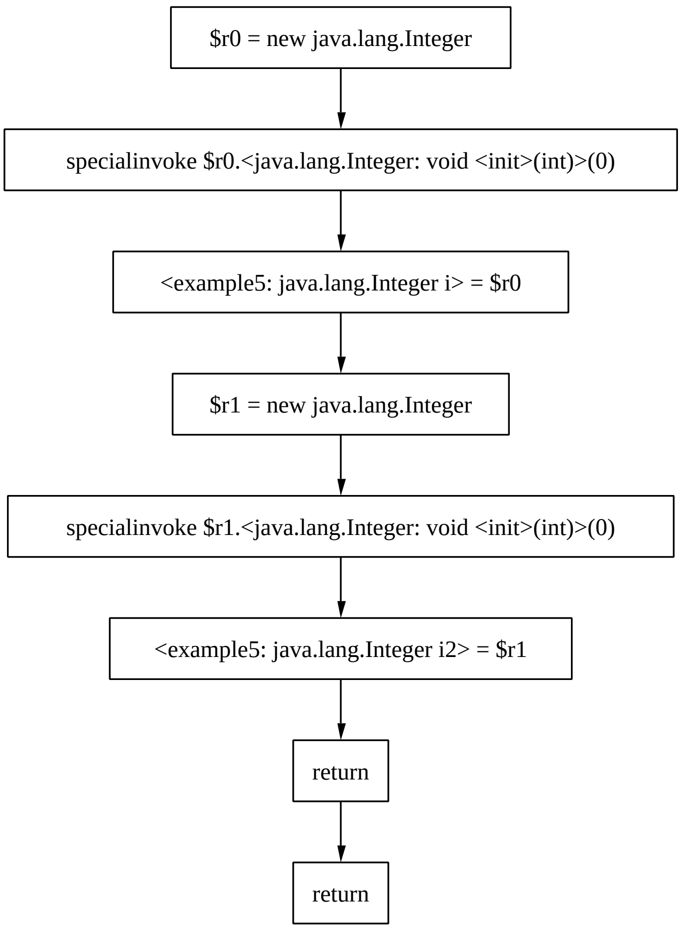
```
import java.io.*;

public class example5
{
    static java.lang.Integer i = new Integer(0);
    static java.lang.Integer i2 = new Integer(0);


    public static void main(String argv[]) {
        synchronized(i) {
            int a = i.intValue();
            i = new Integer(a+1);
            int b = i2.intValue();
            i2 = new Integer(b+1);
        }
        System.out.println(i);
    }
}
```

A.5.2 Control flow graph



Control flow graph for method <init>

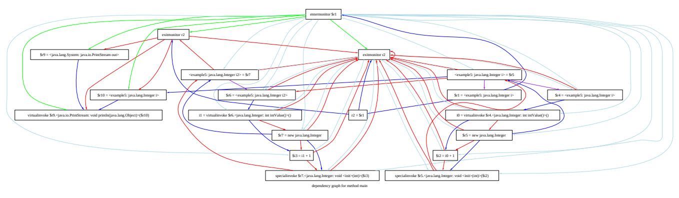

r0 := @parameter0: java.lang.String[]
$r1 = <example5: java.lang.Integer i>
r2 = $r1
entermonitor $r1
$r4 = <example5: java.lang.Integer i>
i0 = virtualinvoke $r4.<java.lang.Integer: int intValue()>()
$r5 = new java.lang.Integer
$i2 = i0 + 1
specialinvoke $r5.<java.lang.Integer: void <init>(int)>($i2)
<example5: java.lang.Integer i> = $r5
$r6 = <example5: java.lang.Integer i2>
i1 = virtualinvoke $r6.<java.lang.Integer: int intValue()>()
$r7 = new java.lang.Integer
$i3 = i1 + 1
specialinvoke $r7.<java.lang.Integer: void <init>(int)>($i3)
<example5: java.lang.Integer i2> = $r7
exitmonitor r2
goto [?= $r9 = <java.lang.System: java.io.PrintStream out>]
$r8 := @caughtexception
$r9 = <java.lang.System: java.io.PrintStream out>
r3 = $r8
$r10 = <example5: java.lang.Integer i>
exitmonitor r2
virtualinvoke $r9.<java.io.PrintStream: void println(java.lang.Object)>($r10)
return
throw r3
return
Control flow graph for method main

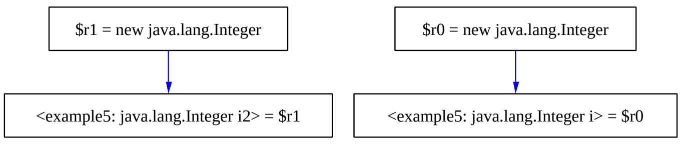

$r0 = new java.lang.Integer
specialinvoke $r0.<java.lang.Integer: void <init>(int)>(0)
<example5: java.lang.Integer i> = $r0
$r1 = new java.lang.Integer
specialinvoke $r1.<java.lang.Integer: void <init>(int)>(0)
<example5: java.lang.Integer i2> = $r1
return
return

Control flow graph for method <clinit>

A.5.3 Program Dependency graph

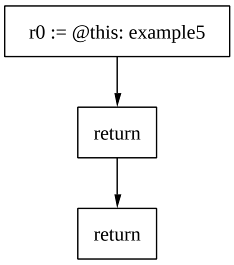
dependency graph for method <init>



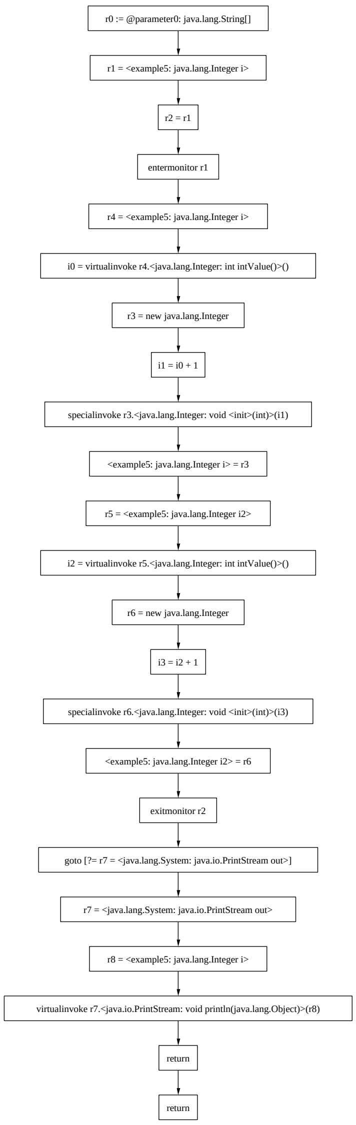

$r1 = new java.lang.Integer
$r0 = new java.lang.Integer
<example5: java.lang.Integer i2> = $r1
<example5: java.lang.Integer i> = $r0

dependency graph for method <clinit>

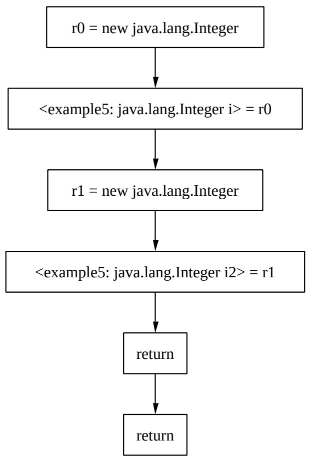
A.5.4 Sliced control flow graph



Sliced control flow graph for method <init>


r0 := @parameter0: java.lang.String[]
r1 = <example5: java.lang.Integer i>
r2 = r1
entermonitor r1
r4 = <example5: java.lang.Integer i>
i0 = virtualinvoke r4.<java.lang.Integer: int intValue()>()
r3 = new java.lang.Integer
i1 = i0 + 1
specialinvoke r3.<java.lang.Integer: void <init>(int)>(i1)
<example5: java.lang.Integer i> = r3
r5 = <example5: java.lang.Integer i2>
i2 = virtualinvoke r5.<java.lang.Integer: int intValue()>()
r6 = new java.lang.Integer
i3 = i2 + 1
specialinvoke r6.<java.lang.Integer: void <init>(int)>(i3)
<example5: java.lang.Integer i2> = r6
exitmonitor r2
goto [?= r7 = <java.lang.System: java.io.PrintStream out>]
r7 = <java.lang.System: java.io.PrintStream out>
r8 = <example5: java.lang.Integer i>
virtualinvoke r7.<java.io.PrintStream: void println(java.lang.Object)>(r8)
return
return

Sliced control flow graph for method main


r0 = new java.lang.Integer
<example5: java.lang.Integer i> = r0
r1 = new java.lang.Integer
<example5: java.lang.Integer i2> = r1
return
return

Sliced control flow graph for method <clinit>

A.5.5 Sliced code

example5.java

```
import java.io.*;
import soot.dava.toolkits.base.DavaMonitor.*;

public class example5
{
    static java.lang.Integer i;
    static java.lang.Integer i2;

    public example5()
    {
        return;
    }

    public static void main(java.lang.String[] r0)
    {
        java.lang.Integer r1, r2, r3, r4, r5, r6, r8;
        int i0, i1, i2, i3;
        java.io.PrintStream r7;

        r1 = i;
        r2 = r1;
        DavaMonitor.v().enter(r1);
        r4 = i;
        i0 = r4.intValue();
        i1 = i0 + 1;
        r3 = new Integer(i1);
        i = r3;
        r5 = i2;
        i2 = r5.intValue();
        i3 = i2 + 1;
        r6 = new Integer(i3);
        i2 = r6;
        DavaMonitor.v().exit(r2);
        r7 = System.out;
        r8 = i;
        r7.println(r8);
        return;
    }

    static
    {
        java.lang.Integer r0, r1;

        r0 = new Integer;
        i = r0;
        r1 = new Integer;
        i2 = r1;
    }
}
```

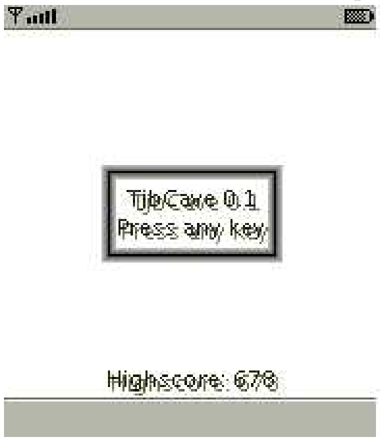
A.5.6 Notes

As you can see in the control flow graph, almost all functions can lead to the exception handler, which is a result from the synchronization. This makes the dependency graph quite messy, to say the least. In the sliced source code, the synchronization statements have been replaced by Dava statements, which are equivalent to original Java synchronization statements. Only statements that modify the variable that is printed are kept in the sliced code.

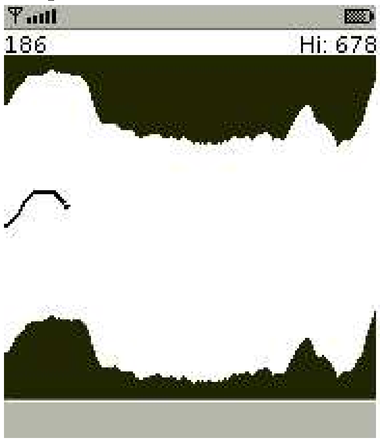
Appendix B

TjbCave, a MIDlet

In this appendix I will describe the MIDlet used in chapter 6 on page 23.



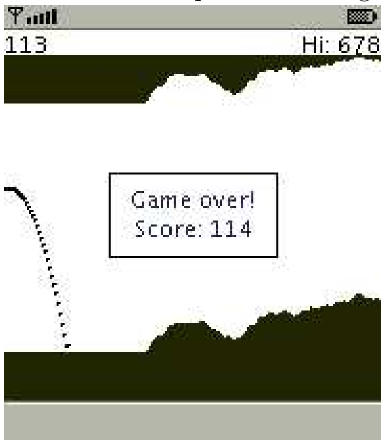
TjbCave is a simple ‘flying’ game where the player has to navigate through a cave that keeps getting smaller. Gravity continually pulls the player down, and pressing a button pushes the player up.



Though mostly harmless, the game does have an invocation that might be suspect, when the player has achieved a high-score, the score is posted to a (in this case fictional) website. Of course the mobile phone asks the user if it may connect to the

Internet for this, because it runs in an untrusted domain. But if TjbCave was to be signed for a higher domain, it should be verified that it does nothing more than post the highscore to the web page. Or maybe it's not even supposed to connect to the Internet and the verifier just wants to make sure of that.

As seen in chapter 6 it does connect and post something.



Bibliography

[1] Compaq Systems Research Center. Extended Static Checker for Java, 2001. version 1.2.4, `http://research.compaq.com/SRC/esc/`.

[2] John Hatcliff, James Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. March 1999.

[3] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. December 1999.

[4] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java(tm) Language Specification, Second Edition*. Addison-Wesley Pub Co, 2000. ISBN 0201310082.

[5] JSR 118 Expert Group. Mobile information device profile, November 2002. Version 2.0 `http://jcp.org/aboutJava/communityprocess/final/jsr118/`.

[6] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification, Second Edition*. Addison-Wesley Pub Co, 1999. ISBN 0201432943.

[7] Loop Project. Logic of Object-Oriented Programs. University of Nijmegen `http://www.cs.kun.nl/ita/research/projects/loop/`.

[8] Sun Microsystems. J2me wireless toolkit. `http://java.sun.com/products/j2mewtoolkit/index.html`.

[9] K.J. Ottenstain and L.M. Ottenstain. The program dependency graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

[10] McGill University Sable Research Group. Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[11] SAnToS Laboratory. Bandera toolset, 2003. `http://bandera.projects.cis.ksu.edu/`.

[12] Bruce Schneier. *Secrets & Lies, Digital Security in a Networked World*. John Wiley & Sons, 2000. ISBN 0-471-25311-1.

[13] SecureSoftware(tm). Rats - rough auditing tool for security. `http://www.securesoftware.com/download_form_rats.htm`.

[14] Security of Systems Group. Extended Static Checker for Java 2, 2003. University of Nijmegen `http://www.cs.kun.nl/ita/research/projects/sos/projects/escjava.html`.

[15] Frank Tip. A survey of program slicing techniques. 1995.

[16] Raja Vale-Rai, Laurie Hendren, Phong Co, Patrick Lam, Etienne Gagnon, and Vijay Sundaresan. Soot - a java bytecode optimalization framework. 1999.

[17] Raja Vale-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. July 1998.

[18] Mark Weiser. Program slices, formal, psychological and practical investigations of an automatic program abstraction method. 1979. PhD thesis, University of Michigan, Ann Arbor, MI.

[19] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.