Aalborg University

Master Thesis

Java Smart Card Security

Automated Implementation of Fault Attack Countermeasures

Group members:

Dennis Jakobsen

Erik Sidelmann Jensen

Supervisors:

René Rydhof Hansen

Mads Christian Olesen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science

Software 10th semester

Address: Selma Lagerlöfs Vej 300

9220 Aalborg Øst

Phone no.: 99 40 99 40

Fax no.: 99 40 97 98

Homepage: http://www.cs.aau.dk

Project title:
Automated Implementation of Fault Attack Countermeasures

Subject:
Java Smart Card Security

Project period:
1. Feb 2016 - 31. May 2016

Group name:
des108f16

Supervisors:
René Rydhof Hansen

Mads Christian Olesen

Group members:
Dennis Jakobsen

Erik Sidelmann Jensen

Copies: 0
Pages: 108
Appendices: 1 & 1 Attachment
Finished: 31. May 2016

Abstract:

With fault attacks on smart cards being a method of performing unauthorized transactions with credit cards, various research papers have proposed software implemented countermeasures against these attacks. The aim of this report is to investigate the details required to implement a tool for automatically inserting countermeasures into smart card applets, specifically Java Card applets. The tool developed implements branch duplication and call graph integrity based on various analyses and a call graph provided with the used bytecode optimization framework called Soot. The aim of the tool is to easen the development process of a Java Card applet developer that needs to secure the applet. The report outlines the considerations that was made during the development of the tool as well as identifies situations where the tool has its shortcomings along with a proposal to a solution of the shortcomming.

Preface

This report is the result of a master thesis in software engineering at Aalborg University.

The aim of the report is to provide an automated tool for implementing fault attack countermeasures at bytecode level for Java Card Applets. References are listed with numbers and not by the author's name(s), e.g. [23]. Furthermore, whenever writing "we", it refers to the project group and its members.

We would like to thank our supervisors for their guidance during the writing of this report.

Summary

In the light of fault attacks on smart cards and the proposed countermeasures mentioned in various research papers, the aim of this project is to automate the implementation of these countermeasures in bytecode. Specifically, a tool is implemented that, given a Java Card applet class file, can insert call graph integrity and implement branch duplication in the applet.

For branch duplication the tool can handle `if` statements, `lookupswitch`, as well as `tableswitch` at bytecode level. Each branch is duplicated by repeating the bytecodes in relation to the branching instruction, whether that being recalculating the value of a variable again or performing a mehtod call again. The tool handles situations where a method call is impure, i.e. it cannot safely be repeated because it may change the state of the program because of side-effects. Furthermore, the tool handles the sitation where it cannot statically be determined which instructions are required to recalculate the variable needed in the condition of the branching instruction. The situation occur when a variable can obtain two or more different values depending on which execution flow is followed at run-time.

For call graph integrity the tool uses a call graph to determine which methods are called. The tool handles polymorphism by grouping all methods in the same hierarchy chain that are overrides and assigning all methods in the group the same unique identifiers that are used to check against when control changes from one method to another.

The tool is implemented using the Soot framework. The framework provides intermediate representations suitable for analysis and rewriting, as the framework originally was designed as an optimization framework for Java. Because of the difficulties implied by the operand stack used in bytecode, the bytecode is transformed into an intermediate representation. The transformation of the Java Card applet is performed on the stackless three address intermediate representation called Jimple, after which the framework transforms the Jimple code back into bytecode. A definition-use and use-definition chain is created in order to determine which Jimple statements to repeat for the duplicated branch. This analysis along with a purity analysis is provided with Soot and used in the transformation for the branch duplication.

The report gives implementation details for creating a tool to automatically insert branch duplication and call graph integrity into a Java Card applet. Furthermore, the report outlines the considerations that need to be taken when developing such tool. Experiments on a few sample applets have been conducted to better understand the impact of the inserted countermeasures in terms of program size, memory usage, and running time.

Contents

CHAPTER 1

Introduction

Today payment transactions are performed using credit cards, also known as smart cards, provided by the customers bank. The smart cards typically follow a well known protocol called EMV with over 730 million cards in circulation[18]. A transaction is completed with a method called "Chip and PIN" where the customer enters a PIN to authorize the payment. The security of such protocol is important to ensure that criminals cannot perform illegal transactions. However, an attack on credit cards has been revealed; a so called man-in-the-middle attack allowing the attacker to trick the credit card into thinking that no PIN was entered while the credit card terminal thinks that a correct PIN was entered, although any PIN would do[18]. What happens is that the credit card falls back to another method called "Chip and signature" when no PIN is entered thus assuming that a signature is given instead. Whereas the terminal thinks that the credit card has authorized the entered PIN. Normally the customer is protected against credit card fraud, but the bank in this case recognizes the transaction as "Verified by PIN" and thus accuse the customer of revealing the PIN, although the attacker never knew the right PIN[18]. The equipment for performing such an attack can be hidden in a backpack leaving the cashier with no suspicion.

The attack shows that security is important whether it being in the protocol itself as in this case or in the authentication directly on the card. Once an attacker gains authorization he or she is able to withdraw money from the victims account.

There are several possible ways an attacker might gain insight in the smart card implementation and explore vulnerabilities. One such way is to reverse engineer the architecture and tamper with memory addresses to alter the program execution. Such an attack is called a fault attack because you introduce a fault in the data causing the program to execute differently. If an attacker knows exactly where to alter the memory to skip execution or otherwise gain authorization he or she is able to authorize payments.

It is difficult to obtain public information about vulnerabilities on smart cards, such as the man-in-the-middle attack described above, because it can be hard to detect the attack when the system acts normally. Another reason is that it is in the manufacturers interest to keep vulnerabilities a secret.

This was the case a few years back when Volkswagen prevented researchers from publishing an article on how insecure a lot of their cars with keyless ignition were[23]. The researchers were able to retrieve the transponder secret key from the car keys, allowing them to copy or emulate the keys and thereby start the cars without the original key. The flaw was revealed to the manufacturer of the keys in February 2012 where they were given nine month to fix the issue. In May 2013 the flaw was revealed to Volkswagen before attempting to publish it at the USENIX conference. This was, however, prevented by Volkswagen as they filed a lawsuit. First after two years and alterations to the publication they were allowed to publish it.

In order to hamper the attacker of learning where to tamper with the chip, countermeasures can be inserted into the code to detect if an attacker is tampering with the execution. If this is the case, the countermeasure will react by executing error handling code that might lock the smart card for further execution and thus further reverse engineering of the code. These countermeasures are called fault attack countermeasures.

Implementing such countermeasures at source code level may render them useless because of compiler optimizations, and implementing them at a lower level may be troublesome for the developer. Fault attack countermeasures that can be implemented with software can be automatically implemented as such countermeasures are designed to generally fit all applications. Automatically implementing countermeasures into a Java Card applet ease the development process by replacing the hard labour of implementing the countermeasures by hand. The subject of this master thesis is to develop such automated tool to implement fault attack countermeasures.

1.1 Problem Statement

Given the above-mentioned problems and goals the problem statement for this project is as follows:

- What is required to automatically insert countermeasures in a Java Card applet?
- What has to be considered when implementing branch duplication and call graph integrity?
- How much of the process can be automated?

CHAPTER 2

Preliminaries

We briefly mention a number of technologies and terms that we use throughout the report.

Java Card

A Java Card is a small embedded system often used for credit cards or access cards. The cards are powered by external equipment, e.g., by inserting the card into a card reader or by wireless power transfer, and communication happens through the Application Protocol Data Unit (APDU) protocol. The cards typically have very limited resources in terms of both memory and computation power. As for memory a card typically has 16 kB of EEPROM (non-volatile mutable memory), 32-48 kB ROM (non-volatile immutable memory), and 1.2 kB RAM (volatile mutable memory)[1, JCVM Section 2.1]. The system is running a variation of the Java Virtual Machine (JVM), namely the Java Card Virtual Machine (JCVM). This virtual machine is mostly a subset of the Java Virtual Machine. It typically does not include types such as integers, floating points, or strings. Also certain concepts like multithreading[1, JCVM Section 3.3], just in time compilation[30], and garbage collection is not included in the JCVM[1, JCVM Section 3.3]. The JCVM is, like the JVM, responsible for running the applets containing Java bytecode. Objects created in the applets are stored in the EEPROM which means that, without garbage collection, the memory is never freed again. Because of this, it is considered good practice to allocate all objects during the install phase of the applet. In this way they are allocated only once.

APDU

Application Protocol Data Unit is the protocol used to communicate between card readers and the Java Card applet on the card[1, API Page 46]. There are two different types of APDU's, command APDU and response APDU. A command is sent to the card and a response is sent back to the card reader[1, API Page 43].

Java Bytecode

Java bytecode, or just bytecode hereafter, is a low level stack based language which is executed by the JVM and JCVM. It is generated by e.g. the Java compiler. It consists of a number of instructions which is executed one by one.

Definition-use/use-definition chain

A definition-use chain is a data structure used to find uses of a specific definition while a use-definition chain does the opposite[16]. The chains can for example be used to find the possible values of a variable at a given point.

Fault attack

Fault attack in general is an attack on some electronic device causing a wrong result[5]. A fault attack is an external attack where e.g., the voltage of the card is tampered with or a strong laser is pointed at the device. These attacks may lead to errors in the execution of the software on the device, or it may break the device in which case no further exploitation is possible. If the attack introduces a fault in the system without breaking the device, this fault may be a transient or a permanent fault. When attacking a smart card the attack can target different parts of the card memory, e.g., the stack, heap, or program code. The next step for the attacker is to be able to take advantage of the faults, e.g. to execute parts of the program that should not be executed under normal circumstances.

CAP File

A CAP file is a container file containing information about a package. The CAP file is a standalone file that contains all information needed to install the applet on Java Card, including information about the constant pool, classes, and methods[19].

CHAPTER 3

Fault Attack Countermeasures

In the light of the attack mentioned in Chapter 1 and the identified possibility of fault attacks on smart cards, this chapter introduces two fault attack countermeasures that is the focus for the automated tool. The countermeasures are branch duplication which can be found in Section 3.1 and call graph integrity in Section 3.2. Our assumption is that an attacker is able to set a specific value in memory, but only one fault for each execution of the program. Furthermore, we assume that fault attacks only affect the operand stack and the program counter.

3.1 Branch Duplication

Branch duplication is a type of countermeasure which attempts to counter attacks on the operand stack where a stack value has been changed by an attacker or attacks on the program counter where instructions are skipped. Branch duplication works by duplicating instructions used to produce values on the stack used by the branching instruction. This type of countermeasure can be used when you have a branch, e.g., `if`, `while`, or `switch`. The considerations you have to make when applying this countermeasure is described in Section 3.1.1.

An example of branch duplication can be seen in Listing 3.2.

```
// If statement without duplication
int cond = authorize();
if (cond == OK) {
  // Some sensitive code
}
```

Listing 3.1: An example of Java source code before rewriting.

```
// If statement with duplication
int cond = authorize();
if (cond == OK) {
  cond = authorize();
  if (cond != OK) {
    // Error handling, condition has changed
  } else {
    // Some sensitive code
  }
}
```

Listing 3.2: An example of Java source code after rewriting.

In Listing 3.2, which is the rewritten version of Listing 3.1, it can be seen that the variable that is part of the condition in the `if` statement is rewritten inside the first `if` statement, and then the condition is checked again in the nested `if` statement. If an attacker manage to change the values on the stack so the true branch is executed instead of the false branch, this type of countermeasure can prevent the attacker from gaining any benefit from his attack. The extra check makes sure that this attack is detected and that the appropriate actions are taken.

As can be seen in Listing 3.2, the nested `if` statement is inverted compared to the original statement. This is a deliberate choice, since this helps protect against attacks that makes the card skip an instruction or change an existing instruction to `NOP` (no operation)[20, Slide 66 and 74-75]. If this happens for the duplicated `if` instruction, the error handling code is executed.

The example in Listing 3.1 and Listing 3.2 is written in a high-level programming language. This has the potential drawback that the compiler might optimize away the extra code, if it finds that the code does not do anything meaningful. A better option is to make the changes at a lower level, e.g. bytecode where the potential optimizations already have taken place. Listing 3.3 and Listing 3.4 illustrates duplication at bytecode level.

```
0:  aload_0
1:  invokevirtual #1 // authorize()
4:  sipush 1234      // OK value
6:  if_icmpne 20
...                  // Sensitive code
20: ...              // Outside sensitive code
```

Listing 3.3: An example of bytecode before rewriting. Roughly equivalent to Listing 3.1.

```
0:  aload_0
1:  invokevirtual #1 // authorize()
4:  sipush 1234      // OK value
6:  if_icmpne 31
9:  aload_0
10: invokevirtual #1 // authorize()
13: sipush 1234      // OK value
15: if_icmpeq 20
...                  // Error handling
20: ...              // Sensitive code
31: ...              // Outside sensitive code
```

Listing 3.4: An example of bytecode after rewriting. Roughly equivalent to Listing 3.2.

As mentioned above some considerations have to be made when implementing branch duplication which will be listed in the following section.

3.1.1 Considerations

Automating the implementation of branch duplication is not always as straight forward as duplicating the instructions required to calculate the branching condition again. There are situations where duplication either cause a false result, undesirable memory usage, or cause the whole program to end up in a wrong state.
Consider the following situations:

Method invocation

When an `if` statement depends on a method invocation, the branch duplication needs to consider whether it is safe to perform the calculation again i.e. it is safe to perform the method invocation again. If the method is not side-effect free invoking it again may cause the program to end up in an undesired state as variables outside the method may be changed, possibly in an uncontrollable way. Another consideration to make before duplicating a method invocation is whether new objects or arrays are instantiated and stored on the heap. This causes problems as there is no garbage collector available on Java Card and thus invoking the method again leaves twice as many objects and/or arrays on the heap. Although this can happen, it is considered bad practice to allocate objects other than in the install method which is only called once. If the method does not return the same value for every invocation the method is not safe either.

Writes to EEPROM

Every persistent data used on Java Card is stored in EEPROM. EEPROM flash memory writes is about 10,000 times slower than writing to RAM[29]. When rewriting a program this fact is worth considering as a rewritten program potentially can have twice as many writes to EEPROM as the original program thus impacting the running time of the program. An example of data types stored in EEPROM are objects and persistent arrays.

Loop constructs

All loop constructs are created at bytecode-level using `if` statements. If the condition of the loop is dependent on a variable that changes for every loop cycle we cannot simply perform the calculation of this variable again as this would overwrite the value and cause an endless loop. An example of such loop could be `while (i < array.length)` where i is incremented by one for every loop cycle. In this case the branch duplication cannot perform the calculation of i again as it would effectively set the variable to its original value, typically `0`.

Overwrites of variables

Variable x is calculated based on variable y and z. Right before an `if` statement with the condition `x >= y` the variable y is overwritten. In this situation we cannot just perform the calculation of variable x as one of the original values used to calculate x is no longer available. The importance in this scenario is to keep the original values used to calculate x such that the duplicated `if` statement uses the right values.

Branch dependent condition

Variable x can obtain two different values dependent on a condition, later in the program execution variable x is used in another branch condition. Which instructions should be duplicated to recreate the value of x for the duplicated branch condition? To elaborate, consider the example where an `if` statement has the condition `x >= y` and that in a prior `switch` statement the variable x is set differently based on some other switch condition `var1`, see example in Listing 3.5. In this case, the required bytecode for checking the condition `x >= y` again is dependent on which branch was taken in the `switch` statement. This may or may not be determinable at compile time and thus extra considerations are required to rewrite the program.

```
int var1 = getValue();
int x, y = 4;
switch (var1) {
   case 20:
      x = 3;
      break;
   case 50:
      x = 4;
      break;
   default:
      x = 1;
}
if (x >= y) {
   // duplicate code to calculate y and x again
   if (x < y) {
      // Error handling
   } else {
      // Some sensitive code
   }
}
```

Listing 3.5: Example of branch dependent condition.

3.2 Call Graph Integrity

Call graph integrity is a countermeasure against attacks on the control flow of the program execution w.r.t. method invocation. An attacker might attack the program counter to jump over an invoke instruction to change the control flow. The attacker may also attack the return address and thereby alter the program flow.

Call graph integrity ensures that the control flow follows the at-compile-time determinable call graph by inserting checks that ensures that a method has been called from one of the allowed methods and by checking that control returns from the right method. This can be done by assigning each method a unique identifier where for every call to a method the unique identifier is assigned to a global variable before the invoke and checked by the callee method that the control indeed came from one of the allowed caller methods. The same check can be done after the callee return, that control indeed came from the callee.

```
public void caller() {
   callee();
}

public void callee() {
   // code
}
```

Listing 3.6: Simple caller-callee example.

```
1  // id: 1
2  public void caller() {
3     identifier = 1;
4     callee();
5     if (identifier != 2) {
6        // error handling
7     }
8  }

10 // id: 2
11 public void callee() {
12    if (identifier != 1) {
13       // error handling
14    }
15    // code
16    identifier = 2;
17 }
```

Listing 3.7: Implementation of CGI for Listing 3.6.

A simple example without the countermeasure can be seen in Listing 3.6 and one with the countermeasure in Listing 3.7. A global variable `identifier` is assigned the ID of the caller right before calling `callee()`, which then checks that this `identifier` has been assigned the correct value. Before `callee()` returns the `identifier` is assigned to the callee ID and checked again at line 5.

As was the case for branch duplication, there are also considerations to take when implementing call graph integrity.

3.2.1 Considerations

One important aspect of implementing call graph integrity is to consider polymorphism where multiple different implementations of the same method may return to the caller. In such situation you can assign a unique ID to each of the methods and check for every possible identifier value, or you can assign a single identifier for all overrides of a method and only check for one value. Being on Java Card with limited memory a solution that takes up as little space as possible is desirable. Because it takes three bytecodes to check a possible identifier value, a `getstatic` to fetch the global identifier variable, a `push` or `const` to compare with a constant value, and lastly an `ifcmp` to do the comparison, keeping the number of comparisons at a minimum is desired.

Taking the memory footprint into account another solution exists that assigns two identifiers to each method; the first is checked against at method entry and the second is assigned at method exit. This approach can be seen in Listing 3.8. Comparing the two approaches in Listing 3.8 and Listing 3.9 w.r.t. memory footprint the approach in Listing 3.8 has a smaller footprint because the number of comparisons at method entry cannot exceed one, whereas in Listing 3.9 the number of comparisons is equal to the number of different methods calling the method as can be seen in line 3.

```
1  // ID1 1234 and ID2 4321
2  public void callee() {
3     // Check ID == 1234
4     // Code
5     // Set ID = 4321
6  }

8  public void caller1() {
9     // Set ID = 1234
10    callee();
11    // Check ID == 4321
12 }

14 public void caller2() {
15    // Set ID = 1234
16    callee();
17    // Check ID == 4321
18 }
```

Listing 3.8: Call graph integrity with two identifiers per method.

```
1  // ID 3
2  public void callee() {
3     // Check ID == 1 || ID == 2
4     // Code
5     // Set ID = 3
6  }
7  // ID 1
8  public void caller1() {
9     // Set ID = 1
10    callee();
11    // Check ID == 3
12 }
13 // ID 2
14 public void caller2() {
15    // Set ID = 2
16    callee();
17    // Check ID == 3
18 }
```

Listing 3.9: Call graph integrity as described in Section 3.2.

CHAPTER 4

Tools

In this chapter we describe the candidate tools for automatically implementing branch duplication and call graph integrity. The chosen tool for this project, Soot, is described in detail, whereas other tools is only described briefly. At last the rationale for choosing Soot is given.

4.1 Soot

Soot is a bytecode optimization framework implemented in Java consisting of three main intermediate representations; Baf, Jimple, and Grimp [25]. The framework provides conversions between each intermediate representation as well as an API for manipulation. An overview of which conversions exists in Soot can be seen in Figure F4-1. The figure illustrates that Soot accepts a compiled bytecode file (`.class`), after which optimizations are available at every intermediate representation. From Jimple, there are two different options of obtaining bytecode again, either via Grimp or via Baf. Each of these options have their advantages and disadvantages which is further described in Section 4.1.2.5 A description of the intermediate representations is given in Section 4.1.2, and a description of the available optimizations at the different representations is given in Section 4.1.3.

Later Soot has been extended with another intermediate representation Shimple, which is the Single Static Assignment (SSA) variation of Jimple [11]. SSA means that each variable is assigned exactly once enforcing different versions of the same variable indicated with a # e.g. `variable#2`.

4.1.1 Packs & Phases

Soot is divdided into packs and phases, where each pack consists of phases. Every `.class` file is passed through the `jb` pack, which does the conversion from bytecode via Baf to Jimple, see Section 4.1.2.4 for further description of this conversion. The `jb` pack (Jimple Body) is applied to every method body of the `.class` file. If Soot's *whole-program* mode is enabled, this Jimple Body is passed through `cg`, `wjtp`, `wjop`, and `wjap`. Figure F4-2 is an example of the flow through packs with whole-program

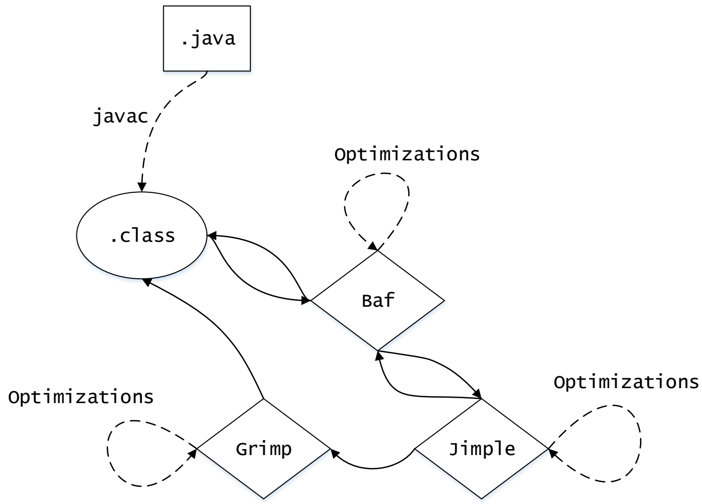

Figure F4-1: Illustration of conversions between intermediate representations. Redrawn and updated from [25].

mode enabled. The first of these packs is the call graph pack which generates a call graph for the whole program. For the rest of the packs the naming convention is as follows: `w` for whole-program, `j` for Jimple, `t`, `o`, `a` for transformation, optimization, and annotation, respectively, and `p` for pack. The last two packs in the flow (`bb` and `tag`) are responsible for converting the Jimple code back into Baf (which is later converted into bytecode) and for aggregating tags to gain uniqueness among them.

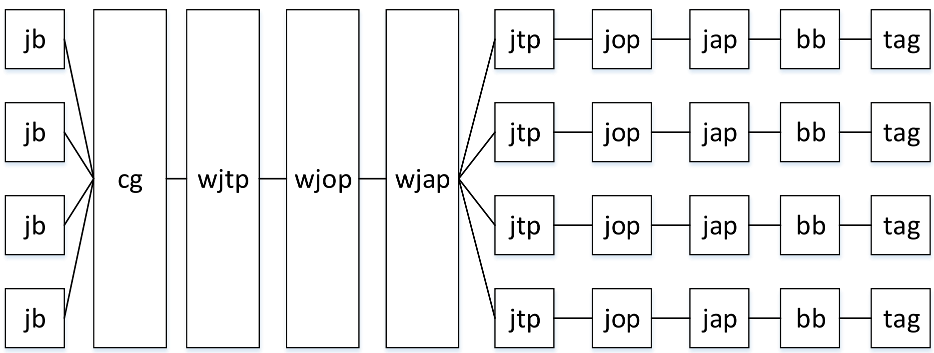

Figure F4-2: A Jimple Body's flow through different packs. Redrawn from [12].

An example of a flow through the Jimple packs can be seen in Figure F4-2. Similar packs for the other intermediate representations exists named after the same naming convention, e.g. `stp` for Shimple Tranformation Pack. An example of the different phases available in a pack can be seen in Listing 4.1.

```
jop           Jimple optimization pack (intraprocedural)
    jop.cse      Common subexpression eliminator
    jop.bcm      Busy code motion: unaggressive partial
                 redundancy elimination
    jop.lcm      Lazy code motion: aggressive partial
                 redundancy elimination
    jop.cp       Copy propagator
    jop.cpf      Constant propagator and folder
    jop.cbf      Conditional branch folder
    jop.dae      Dead assignment eliminator
    jop.nce      Null Check Eliminator
    jop.uce1     Unreachable code eliminator, pass 1
    jop.ubf1     Unconditional branch folder, pass 1
    jop.uce2     Unreachable code eliminator, pass 2
    jop.ubf2     Unconditional branch folder, pass 2
    jop.ule      Unused local eliminator
```

Listing 4.1: List of available phases in the `jop` pack.

Each of the transformation, optimization, and annotation packs can be enabled and disabled, and each of the phases in these packs can also be enabled and disabled. This allows the end-user to gain control of which phases are run on the code. This is useful when you do not want the optimization framework to optimize away intentionally redundant code.

In order to know which intermediate representation is best suited for the transformation needed to implement the countermeasures described in Chapter 3, a closer look at the intermediate representation is needed.

4.1.2 Intermediate Representations

As mentioned in Section 4.1 Soot operates on three main intermediate languages, Baf, Jimple, and Grimp. Different languages is used as they each have different capabilities in terms of optimization and analysis. Optimizing directly on bytecode gives rise to a number of issues. Some of the issues are related to the stack based nature of bytecode, as well as the large number of bytecodes.

The bytecode instructions can be split into two groups, expressions and actions[27]. Expressions is instructions that add, remove, and manipulate values on the operand stack, and actions being store, put and invoke instructions; instructions that produce a side effect.

Knowing which expressions that influences the outcome of an action and how they do it, is not straight forward when analyzing bytecode. Since different order of instructions may result in the same outcome, and because all expressions does not necessarily have to be right before the action, you potentially have to analyze all instructions preceding the action. An example of two different orders of bytecode that produce the same result can be seen in Listing 4.2 and Listing 4.3.

```
iconst_1
bipush 14
iadd
bipush 20
imul
istore_1
```

Listing 4.2: One possible order of instructions in bytecode.

```
bipush 20
iconst_1
bipush 14
iadd
imul
istore_1
```

Listing 4.3: Another order producing same result as Listing 4.2.

Even though the example in Listing 4.2 and Listing 4.3 are different, they both store the result of $(1 + 14) \times 20$ in local variable 1. It is even possible, because of the stack based system, to have intermingled instructions in between the expressions used by the `istore_1` action. In order to know which expressions that influence the outcome of an action, you have to construct an expression tree.

If more control of the generated bytecode is needed, Baf is a possibility. Baf is used when creating Jimple from bytecode and is also one of the alternatives when translating back to bytecode, with Grimp being the other alternative. Both Baf and Grimp is described below, together with Jimple which is the main intermediate representation in Soot.

When creating your own analysis or rewriting, the intermediate language you should use depends not only on what is to be analyzed or what to rewrite, but also at what stage you want to do it. Often Jimple will be the langauge to use, but if you for example need to rewrite something just before bytecode is created, you might want to consider using Baf or Grimp, depending on which language is used to create the bytecode.

4.1.2.1 Baf

Baf is a stack based intermediate representation which is simpler and more readable than regular bytecode. It is used by analyses and optimizations which have to be performed on stack code, e.g., in the production of Jimple code and peephole optimizations. The number of instructions is heavily reduced compared to bytecode. There are only about 60 Baf instructions while there are roughly 200 bytecodes. Even with this heavily reduced number of instructions, Baf is still able to represent all the bytecodes. This is possible because Baf has eliminated all the type specific instructions found in bytecode, e.g., `iload` and `fload`, as well as the shorthand single byte instructions, e.g., `iload_0` and `iload_1`. All these have been replace by a `load.t` *`local`* instruction, where `t` specifies the type, e.g., `load.i` and `load.l`, and *local* specifies which local variable to load.

An example of Baf code can be seen in Listing 4.5 which is the Baf equivalent of the bytecode example in Listing 4.4.

```
public short secureMethod1(short);
    Code:
       0: aload_0
       1: iload_1
       2: invokevirtual #4  // Method secureMethod2:(S)S
       5: istore_1
       6: iload_1
       7: sipush        12456
      10: if_icmpge     20
      13: iload_1
      14: sipush        4000
      17: iadd
      18: i2s
      19: ireturn
      20: iload_1
      21: sipush        4000
      24: isub
      25: i2s
      26: ireturn
```

Listing 4.4: An example of a method in bytecode.

```
1  public short secureMethod1(short)
2  {
3      word r0, s0;
4      r0 := @this: dk.aau.cs.test.TestMethods;
5      s0 := @parameter0: short;
6      load.r r0;
7      load.s s0;
8      virtualinvoke <dk.aau.cs.test.TestMethods: short secureMethod2(short)>;
9      store.s s0;
10     load.s s0;
11     push 12456;
12     ifcmpge.s label0;
13     load.s s0;
14     push 4000;
15     add.s;
16     i2s;
17     return.s;
18  label0:
19     load.s s0;
20     push 4000;
21     sub.s;
22     i2s;
23     return.s;
24 }
```

Listing 4.5: An example of Baf intermediate representation of the method in Listing 4.4.

Comparing the two code samples in Listing 4.4 and Listing 4.5 we see a couple of differences. First at line 3 in Listing 4.5 there are explicit declarations of local variables. `word` denotes that both variables are allocated 32 bits, where `r0` is a reference variable and `s0` is a `short` variable. Their value is explicitly assigned in the next two lines. The second notable difference is that there is no constant pool in Baf, which means that content from the constant pool is instead explicitly written out, which can be seen in line 8 in Listing 4.5. The last notable difference is that Baf does not reference jumps by index but by labels, e.g. `label0:`.

4.1.2.2 Jimple

Jimple is the primary intermediate language in Soot. It is a typed 3-address code representation of the Baf intermediate language. 3-address code is a way of writing code such that each expression has at most 3 operands. The 3 operands are often combined with an assignment and a binary operator, e.g. $a1 = a2 + a3$. An example of a how the calculation in Listing 4.2 is written in Jimple:

$$i1 = 1 + 14$$
$$i2 = i1 \times 20$$

Where Baf has about 60 different instructions, Jimple only has about 20. The simplicity of this representation makes it ideal for writing analyses and optimizations. Another feature of Jimple, is that the stack is replaced by additional local variables and references to stack locations is instead replaces by references to local variables. This makes Jimple the intermediate representation in Soot where most of the analyses and optimizations takes place.

An example of Jimple code can be seen in Listing 4.6 which is the equivalent of the Baf code in Listing 4.5.

```
1  public short secureMethod1(short)
2  {
3      dk.aau.cs.test.TestMethods r0;
4      short s0, s1, $s3, $s5;
5      int $i2, $i4;

7      r0 := @this: dk.aau.cs.test.TestMethods;
8      s0 := @parameter0: short;
9      s1 = virtualinvoke r0.<dk.aau.cs.test.TestMethods: short secureMethod2(short)>(s0
           );
10     if s1 >= 12456 goto label0;

12     $i2 = s1 + 4000;
13     $s3 = (short) $i2;
14     return $s3;

16  label0:
17     $i4 = s1 - 4000;
18     $s5 = (short) $i4;
19     return $s5;
20 }
```

Listing 4.6: An example of Jimple intermediate representation of the method in Listing 4.5.

On line 3-5 in Listing 4.6 you can see that all variables are typed. These declarations as well as those at line 7-8 have to be declared at the beginning of each method. As in Baf, class and method names are written explicitly when called. Since Jimple is stackless, stack values are instead represented as local variables starting with $, while variables without are representing otherwise local variables.

4.1.2.3 Grimp

Grimp is an easier to read version of Jimple. It is not on 3-address form, which allows for more compact and closer to Java source representation. It does, however, still hold the property of a 3-address representation that a statement only allows one side-effect. This form allows for tree constructions which help in code generation. Grimp is therefore one of the intermediate representations, along with Baf, that can be the intermediate representation used to generate bytecode.

```
public short secureMethod1(short)
{
    dk.aau.cs.test.TestMethods r0;
    short s0, s1;

    r0 := @this;
    s0 := @parameter0;
    s1 = r0.secureMethod2(s0);
    if s1 >= 12456 goto label0;

    return (short) (s1 + 4000);

 label0:
    return (short) (s1 - 4000);
}
```

Listing 4.7: An example of Grimp intermediate representation of the method in Listing 4.4.

As can be seen in Listing 4.7 it is shorter than the Jimple example in Listing 4.6. Primarily because Grimp is not on 3-address code representation. Method calls are also shortened in Grimp, where only the name of the method, and not the class is written.

4.1.2.4 Transforming Bytecode to Jimple

Converting bytecode into Jimple is a 5 step process, as illustrated in Figure F4-3. These steps are necessary because bytecode is untyped stack code while Jimple is typed 3-address code.

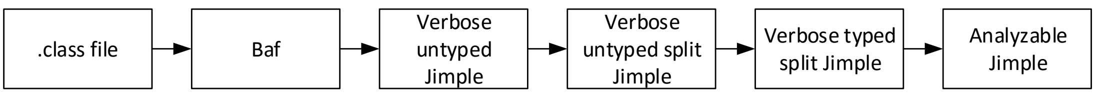

Figure F4-3: Steps for turning bytecode into Jimple.

The first step in this process is to turn the bytecode into Baf. This is mostly a straightforward process since most bytecode instructions have an equivalent Baf instruction. Only two instructions require special care, namely `dup` and `dup2`. This is because the stack in Baf is typed whereas in bytecode it is not. When pushing a `long` or `double` value to the stack in bytecode, the value uses 64 bit, which means that it is split into two 32 bit values, whereas on the stack in Baf it is just one value. `dup` in Baf duplicates one value, and `dup2` two values. This means that `dup2` may potentially duplicate 128 bit, where it only duplicates 64 bit in bytecode. To convert these two instructions is it necessary to compute an abstract stack interpretation where the content on the stack after each instruction is determined in order to know what type of value is to be duplicated.

The next step in the conversion is to convert each Baf instruction into a Jimple instruction. First the stack height is computed after each Baf instruction. This is used to determine the number of local variables needed in Jimple to store all stack values. This can be computed by a simple traversal of the program. When the height is known a variable can be created for each local variable in Baf, as well as one for each stack position. These variables are named `l`x where $0 \leq x < numberOfLocals$ for locals, and `$stack`y where $0 \leq y < stackHeight$ for stack variables. Lastly the Baf instructions are converted to Jimple instructions and local and stack values are mapped to the aforementioned Jimple locals.

Next Jimple locals are split according to webs, computed by traversing use-definition and definition-use chains, such that each web has its own local variable. A web is a subset of the uses and definitions of a local variable. These webs are self-contained which means that the local for each web can safely be renamed without breaking other parts of the code. Generally this means that for each overwrite of a variable in another web, a new variable is assigned instead of overwriting the previous one, and uses of this value uses the new variable. All Jimple locals are split, both local variables and stack variables. This split will therefore often increase the number of Jimple locals, some of which may later be optimized away. An example of how locals are split can be seen in Listing 4.9. The following listings is a direct copy from [27].

```
public int runningExample()
{
  unknown l0, $stack0, l2,
  l3, $stack1, l1, $stack2;
  l0 := @this: Type;
  $stack0 = 0;
  l2 = $stack0;
  $stack0 = l0;
  $stack0 = $stack0.condition;
  if $stack0 == 0 goto label0;
  $stack0 = 5;
  l3 = $stack0;
  $stack0 = new B;
  $stack1 = $stack0;
  specialinvoke
  $stack1.<init>();
  l1 = $stack0;
  $stack0 = l2;
  $stack1 = l3;
  l3 = l3 + 1;
  $stack0 = $stack0 + $stack1;
  l2 = $stack0;
  goto label1;
  ...
}
```

Listing 4.8: A Jimple code sample before splitting local and stack variables. Taken from [27].

```
public int runningExample()
{
  unknown l0, $stack0, l2, l3, $stack1
      , l1, $stack2, $stack0#2,
  $stack0#3, $stack0#4, $stack0#5,
      $stack0#6, $stack1#2,
  l3#2, $stack0#7, $stack0#8, l3#3,
      $stack0#9, $stack1#3,
  $stack0#10, $stack0#11, $stack1#4,
      $stack0#12;
  l0 := @this: Test;
  $stack0 = 0;
  l2 = $stack0;
  $stack0#2 = l0;
  $stack0#3 = $stack0#2.condition;
  if $stack0#3 == 0 goto label0;
  $stack0#4 = 5;
  l3 = $stack0#4;
  $stack0#5 = new B;
  $stack1 = $stack0#5;
  specialinvoke $stack1.<init>();
  l1 = $stack0#5;
  $stack0#6 = l2;
  $stack1#2 = l3;
  l3#2 = l3 + 1;
  $stack0#7 = $stack0#6 + $stack1#2;
  l2 = $stack0#7;
  goto label1;
  ...
}
```

Listing 4.9: Example after splitting Listing 4.8. Taken from [27].

In Listing 4.8 there are 4 local variables and 3 stack variables. After splitting the variables, in Listing 4.9, we end up with 6 local variables and 17 stack variables. We see that `$stack0` is split into 12 different variables. It is expected that the first stack position, `$stack0`, is to be split the most since this position is generally used the most.

This splitting makes typing easier, which is the next step in the process. The typing performed by Soot is done using an efficient multi-stage static typing algorithm. According to experiments this polynomial time multi-stage algorithm can type 99.8% of the methods using only stage 1 out of 3[10]. A typed example can be seen in Listing 4.10 which is the typed equivalent of Listing 4.9.

The last phase is a clean up phase where some redundant code may be removed. By using copy propagation, back copy propagation, and constant propagation it may be possible to remove some of the added locals from the splitting phase. The final code after propagation can be seen in Listing 4.11.

```
public int runningExample()
{
  Test l0, $stack0#2;
  int $stack0, l2, l3, $stack0#3, $stack0#4, $stack0#6,
  $stack1#2, l3#2, $stack0#7, $stack0#11, $stack1#4,
  $stack0#12;
  B $stack1, $stack0#5;
  A l1, $stack0#10;
  java.lang.String $stack2, $stack0#8, l3#3;
  C $stack0#9, $stack1#3;
  l0 := @this;
  $stack0 = 0;
  l2 = $stack0;
  $stack0#2 = l0;
  $stack0#3 = $stack0#2.condition;
  if $stack0#3 == 0 goto label0;
  $stack0#4 = 5;
  l3 = $stack0#4;
  $stack0#5 = new B;
  $stack1 = $stack0#5;
  specialinvoke $stack1.<init>();
  l1 = $stack0#5;
  $stack0#6 = l2;
  $stack1#2 = l3;
  l3#2 = l3 + 1;
  $stack0#7 = $stack0#6 + $stack1#2;
  l2 = $stack0#7;
  goto label1;
  ...
}
```

Listing 4.10: Example Jimple code after typing and before propagating constants and copies. Taken from [27].

```
public int runningExample()
{
  Test l0;
  int l2, l3, $stack0#3, l3#2,
  $stack0#11, $stack0#12;
  A l1;
  B $stack0#5;
  java.lang.String l3#3;
  C $stack0#9;
  l0 := @this;
  l2 = 0;
  $stack0#3 = l0.condition;
  if $stack0#3 == 0 goto label0;
  l3 = 5;
  $stack0#5 = new B;
  specialinvoke $stack0#5.<init>();
  l1 = $stack0#5;
  l3#2 = l3 + 1;
  l2 = l2 + l3;
  goto label1;
  ...
}
```

Listing 4.11: Example Jimple code after propagating constants and copies. Taken from [27].

4.1.2.5 Transforming Jimple to Bytecode

Soot provides two solutions to transform the Jimple code back to bytecode. One is by turning Jimple into Grimp, and then into bytecode. The other is by going through Baf instead of Grimp.

In the first approach, through Grimp, Soot attempts to generate Grimp code that resembles the original tree representation of the code, and then produce the bytecode by traversing the tree. In order to do so, two algorithms have to be applied. The first one is expression aggregation. Since Grimp is not on 3-address form like Jimple, it allows for more operands. The expression aggregation attempts to move all relevant operands to the right hand side of the assignment, e.g. `l0 = 2 * 5; l1 = l0 + 4;` in Jimple will be `l0 = 2 * 5 + 4` in Grimp. This is, however, not always possible since Grimp only allows for one side effect per expression. When multiple side effects are encountered for the same assignment Soot splits the assignment into more assignments but this causes inefficient bytecode compared to the Java compiler. This is attempted to be mitigated by performing peephole optimizations on the code. However, this does not always resolve the problem. The second algorithm is constructor folding. Since Grimp features a `newinvoke` expression that combines the Jimple `new` and `specialinvoke` expressions, these expressions are collapsed into the `newinvoke` expression which then allows for further expression aggregation. The motivation for aggregating expressions is because larger expressions is better when generating bytecode[27]. Finally the tree is traversed and bytecode is produced.

The other approach, which goes via Baf, creates naive bytecode and then attempts to optimize upon it, whereas the Grimp approach attempts to make efficient bytecode directly. This transformation is split into four steps. The first step is to convert the Jimple code directly into Baf code. This produces inefficient Baf code, since Jimple is not a stack based language, so all temporary values in Jimple are stored in locals in Baf instead of on the stack. This means that more local variables are used, and redundant `store` and `load` instructions are used. These are to be optimized away in the second step. A few cases cover most of the redundancy. The first case is where a `load` is followed directly after a `store` on the same local, and that the value is not used afterwards. In this case both instructions can be removed. Another case is when a `store` is followed directly by two `load` on the same local. This case can be replaced by a `dup` instruction. A third case requires a little more care. This case is when a `load` does not directly follow the `store` instruction, i.e. a sequence of interleaving instructions exists. To determine if these `store` and `load` instructions can be removed the net stack height variation (nshv) and minimum stack height variation (mshv)[26] is calculated for the interleaving sequence. Nshv is the stack height difference after executing the sequence while the mshv is calculated while executing the sequence of instructions. If a sequence of instructions both have a nshv and a mshv of 0, then the `load` and `store` instructions can safely be removed. This happens for example if the interleaving instructions have nothing to do with the following instructions, i.e. everything pushed to the stack is also popped from the stack in the interleaving part. If this is not the case, then reordering of the instructions is attempted, which may permit the removal of the instructions. The third step in converting to bytecode is packing local variables. This has the purpose of reusing local variables when they are no longer in use, instead of naively introduce a new local variable for each value. The last step is to convert the Baf code into bytecode. First the maximum stack height is calculated for each method, since this is required by the Java Virtual Machine. This is done by a simple traversal of the Baf code. Then every Baf instruction is converted to the equivalent bytecode instruction and Baf local variables is mapped to local variables in bytecode.

4.1.3 Optimizations

As mentioned previously in Section 4.1.1 Soot is divided into packs and phases, where one pack for each intermediate representation contains optimization phases on that representation. To get a better understanding of which optimizations are applied to a program through the optimization framework the same flow through packs as in

Figure F4-2 is used.

Assuming that Soot is running in *whole-program* mode and that the optimization packs are enabled, the optimizing packs applied are `wjop` and `jop`. The available phases can be seen in Listing 4.1 for `jop` and Listing 4.12 for `wjop`. By default, Soot does not run in *whole-program* mode and neither of the optimizing packs are enabled, thus using Soot as is does not cause any optimization.

```
wjop         Whole-jimple optimization pack
    wjop.smb      Static method binder: Devirtualizes
                  monomorphic calls
    wjop.si       Static inliner: inlines monomorphic calls
```

Listing 4.12: List of available phases in `wjop` pack.

Although the optimizing packs are not enabled by default, enabling them does not necessarily enable all the optimizations available. Each phase in a pack can also be enabled or disabled and does also have a default value. Enabling both `wjop` and `jop` applies these optimizations by default: **jop.cp** Copy Propagator, **jop.cpf** Jimple Constant Propagator and Folder, **jop.cbf** Conditional Branch Folder, **jop.dae** Dead Assignment Eliminator, **jop.uce1** Unreachable Code Eliminator 1, **jop.ubf1** Unconditional Branch Folder 1, **jop.uce2** Unreachable Code Eliminator 2, **jop.ubf2** Unconditional Branch Folder 2, **jop.ule** Unused Local Eliminator, and **wjop.si** Static Inliner[13].

There is no difference in the `jop.uce1` and `jop.uce2` phases. It simply means that the code is passes through this phase again. It is possible to enable and disable phases and thus gives the end-user full control of which optimizations are performed on the program. For example one might want to disable some of the eliminator phases if the purpose is to create such program with redundancy.

4.1.4 Available Analyses

Soot supports a number of analyses out of the box. For our project we need a call graph as well as the definition-use/use-definition chains. We also use the purity analysis to gain knowledge about the purity of methods, see Section 5.5.

- Call-graph construction.
- Points-to analysis.
- Definition-use/use-definition chains.
- Template-driven Intra-procedural data-flow analysis.
- Template-driven Inter-procedural data-flow analysis, in combination with heros.
- Taint analysis in combination with FlowDroid.
- Purity analysis.

4.2 Other Tools

In this section we describe the other candidates for implementing a tool to automatically insert branch duplication and call graph integrity. In order to do so we need to find definitions and uses of a variable to determine which instructions to duplicate to calculate the condition of an `if` statement again in the branch duplication countermeasure. Furthermore we need a call graph to implement the call graph integrity countermeasure. Operating directly on bytecode introduces some challenges that is not present in an intermediate representation on, for instance, three-address form. A challenge would be to find out which instructions are related as these may be located far from each other in the bytecode[21]. In other words, once a value is pushed onto the stack it does not necessarily have to be popped as soon as possible. Therefore we primarily focus on tools where a more convenient stackless representation is available and even better where a definition-use analysis is provided. In the following we describe Sawja which has a suitable intermediate representation for static analysis. Another alternative is Wala[9]. WALA is a tool used for static and dynamic analysis of bytecode. According to their tutorial WALA only supports limited code transformation [8]. It uses an intermediate representation but this representation is immutable and does not provide any code generation which is a problem for our project. Furthermore, according to our supervisor the learning curve is very steep[14]. Therefore, this tool is not considered for this project. Other tools discovered does not provide an intermediate representation and operates directly on bytecode. Among these are ASM[7], BCEL[6], and SERP[28]. These tools are not described or considered further as the workload required for performing the necessary analysis is too high.

4.2.1 Sawja - Static Analysis Workshop for Java

Sawja consists of two parts, one for providing a high level representation of bytecode, and one for operating on this high level representation. The first part is called *Javalib* and can be used as a stand-alone library, whereas *Sawja* itself is dependent on the high level representation provided in *Javalib*. The reason for making *Javalib* available as an independent library is that the process of parsing bytecode into a high level representation is a common task for all analysis and thus is available for other analysis tools to use[15].

Sawja provides two stackless intermediate representations called *JBir* and *A3Bir* which is the 3-address representation of *JBir*[15]. There are also SSA forms of these representations called *JBirSSA* and *A3BirSSA*[3].

A focus point in the development of *Sawja* was performance w.r.t. running time and memory footprint.

The target of *Sawja* is static analysis tools and thus a reverse transformation from intermediate representation back to bytecode is not available. This means that there is no current possibility of rewriting the *.class* file. In [15] the authors state that they would like to facilitate transfer of annotation from source to intermediate representation and back, with the results of the analysis. But in its current state the output format of the analysis is HTML or through an Eclipse plugin.

According to [3] Sawja supports the following analyses:

- Class Reachability Analysis.
- Rapid Type Analysis.
- Live variable analysis.
- Reachable definitions analysis.
- Available expressions analysis.
- Reachable Methods analysis.

4.3 Choosing Soot

We have chosen to implement the automated tool using Soot as framework for several reasons:

- Soot is a mature framework that started out as an optimizing framework in 1999, but has later been used by researchers and practitioners to analyse, instrument, optimize, and visualize Java applications.
- Soot provides the analyses needed to implement the countermeasures described in Chapter 3.
- Soot operates on suitable intermediate representations that enables easier analysis and rewriting.
- Soot is originally designed for optimization and thus suits the process of rewriting applications well.

Sawja does also provide suitable intermediate representations, but does not provide functionality to output to a bytecode class file again. Sawja is originally designed to perform analysis on bytecode and visualize the result in e.g. HTML. Therefore Sawja is not as good a candidate as Soot.

CHAPTER 5

Implementation

Soot works as a stand-alone tool divided into packs and phases as described in Section 4.1.1. To extend Soot with custom analyses you have to add your own phase to a pack. This is done by adding a `Transformer`, either a `BodyTransformer` or a `SceneTransformer`, where the former only applies for intra-procedural analysis and the latter applies for inter-procedural analysis. The phases visited during execution of the tool can be seen in Figure F5-1.

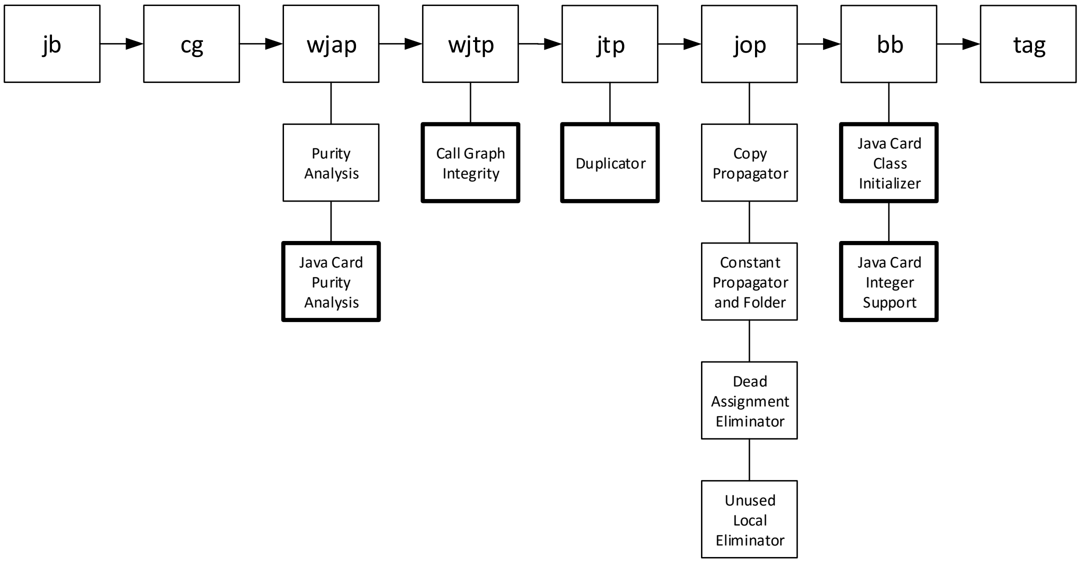

Figure F5-1: The applied phases in the rewriting tool. The bold marked phases are the added phases.

The first phase, `jb`, is necessary to transform bytecode into Jimple as described in Section 4.1.2.4. In order to perform a purity analysis to determine whether a method is pure or impure, we enable Soots *whole-program* mode which enables the `cg` that creates a call graph required to perform whole program analyses. As can be seen the purity analysis is located in the `wjap` pack determining the purity of each method in the call graph. The provided implementation of a purity analysis is based on [22] in which the definition of a pure method is given as:

> *A method is pure if it does not mutate any location that exists in the program state right before method invocation[22].*

In Figure F5-1 the phases implemented in this project are highlighted with a thicker line. As can be seen we implemented a phase to the `wjap` pack called *Java Card Purity Analysis* that corrects the result from the *Purity Analysis* to take into account issues arisen because of a potentially lacking garbage collector on Java Card. This is further described in Section 5.3. Furthermore, we implemented a *whole-program* transformation phase called *Call Graph Integrity* which is responsible for implementing the call graph integrity countermeasure using the call graph generated by the `cg` pack. This phase is further described in Section 5.4. Then we have the *Duplicator* which is responsible for transforming the body of a method to implement branch duplication. This phase is further described in Section 5.5. Lastly we have added a couple of phases to the Baf Body Creation (bb) pack which fixes some issues related to the differences between Java and Java Card. We add them to this pack since it is the last intermediate representation phase, which means that a later translation will not disturb our changes. These are described in Section 5.6 and Section 5.7.

Notice that the order of applying the `wjtp`, `wjop`, `wjap` has been altered such that the annotation pack is applied before the transformation pack. This reorder is done because the call graph integrity phase otherwise would render the purity analysis invalid as writes to a static variable is inserted into every method in the call graph, resulting in these methods being impure.

Besides implementing branch duplication on Jimple code, we enable the Jimple optimization pack to eliminate unnecessary `load` and `store` bytecode instructions generated by the transformation from unoptimized Jimple code to bytecode. The applied optimizations are further described in Section 5.8.

The last two packs, `bb` and `tag`, are enabled by default and applies the default phases necessary to transform Jimple code to bytecode as described in Section 4.1.2.5.

Java Card exists in different version with different capabilities. In order to develop the tool against one specification we make this choice in the following section.

5.1 Java Card Specification

For this project we focus on Java Card 2, since this version is the most limited version, which is still widely used. There exists a new version, version 3, which have some new capabilities, e.g. a volatile heap, while still being compatible with version 2[2]. The instruction set for Java Card is a subset of the full Java instruction set, which means that we do not focus on instructions that does not exists in Java Card, e.g. `monitorenter`. This decision also influence how we handle object creation, since Java Card does not necessarily use garbage collection[1, JCVM Section 3.3]. We therefore do not duplicate objects, even if the object is part of a condition. If our focus was on Java Card 3, the extra volatile heap would allow us to duplicate some objects, since they would be removed when the power source is removed. By targeting the older Java Card 2 it allows us to rewrite applets for both version.

5.2 Running Our Tool

Our tool is provided in a `.jar` file taking as input `.class` files to automatically implement the countermeasures in. A folder called `sootOutput` is created containing the rewritten files. Running the program from the class path root could look like this,

```
java -jar RewritingTool.jar dk.aau.cs.ClassA
```

where the class file to rewrite is `ClassA` and is located in subdirectory `dk/aau/cs/`. The secured class file is located in `sootOutput/dk/aau/cs/`. Which countermeasure to apply should be specified through annotating the methods in source code. One important annotation is to mark at least one entry-point in the program, typically the `process` method. Without this entry-point call graph integrity will not be applied. Examples of annotating methods can be seen in Listing 5.1 and Listing 5.2, where entry points are annotated with the `@EntryPoint` annotation and methods to implement the branch duplication countermeasure in is annotated with `@DuplicateBranches`.

```
@DuplicateBranches
public short methodWithBranches() {
  ...
}
```

Listing 5.1: Annotating method for applying branch duplication.

```
@EntryPoint
public void process(APDU command) {
  ...
}
```

Listing 5.2: Annotating the entry-point for the call graph.

The next sections will cover our added phases as seen in Figure F5-1.

5.3 Java Card Purity Analysis

This analysis is added to the `wjap` pack, which is short for Whole Jimple Annotation Pack. The purpose of this pack is not to change the body of a Jimple body, but rather annotate methods and statements that may be used in later analyses. This analysis is necessary because the default purity analysis does not mark methods that create objects as impure. For the purpose of Java Card we consider this to be wrong, because allocated objects are not freed by the virtual machine.

Listing 5.3 shows the pseudo-code for our Java Card purity analysis. Our analysis takes advantage of the purity analysis that is run before our analysis. This means that we can skip all methods that is already marked as impure, since our analysis does not change that result. If method calls are chained and one of the calls is changed to impure, all previous methods in the chain are marked as impure as well.

```
For each entry point ep
  Skip to next ep if current ep is marked as impure
  Else call checkPurity with ep

checkPurity:
Check if ep contains new, newarray, or multinewarray instructions, and if so, mark ep
     as impure
For each method call out of ep, t, call checkPurity recursively on t
If t is marked as impure, mark caller as impure as well
```

Listing 5.3: Pseudo-code for Java Card purity analysis.

For this analysis we have to be aware of the `<clinit>` initialization method. The call graph shows a possible call to `<clinit>` for all methods using static variables. This method is marked as impure, since it writes to static variables, but this should not mark all methods reading from static variables as impure.

5.4 Call Graph Integrity

As seen in Figure F5-1 we have added a call graph integrity countermeasure to the `wjtp` (whole Jimple transformation pack) pack. This countermeasure attempts to ensure that the right method is called by an invoke instructions, as well as ensuring that the program returns from the right method. This phase can be disabled by leaving out the `@EntryPoint` annotation or by passing the command line option:

```
-p wjtp.CallGraphIntegrity enabled:false
```

We only apply the countermeasure for methods that we can actually rewrite. We do this by getting a list of methods that is within the classes supplied as input to the tool. The tool works by assigning two IDs to each rewritable method group. A method group of a given method is the method itself as well as all methods that it overwrites or is overwritten by. This is necessary in order to handle polymorphism. If a method is not overwriting a method and it is not overwritten by any method, it is the only method in its group. The first that happens in this phase is the creation of a new class, called `CGII`, which has one static field, `identifier`. This field is used to store the ID of the method being called or returned from. At the beginning of each method, if it is not an entry point, we check that `identifier` is set to the methods first ID. We do not create any checks at the beginning of entry methods, since `identifier` may not be set at this point. Next, for each invoke statement in the body of a method, we check if the target is a rewritable method. If this is the case, we assign `identifier` to the target method's first ID, and after the invoke, we check if `identifier` is now the target method's second ID. If it is not the target a jump to the endless loop at the end of the method is performed. This works because we set `identifier` to the methods second ID before each return instruction. This is illustrated in Listing 5.4 which roughly correspond to the Java code in Listing 5.5. Finally we add a goto loop at the very end of the method that is used to perform an endless loop if a wrong path has been taken. The whole implementation is shown as pseudo code in Listing 5.6.

For this countermeasure we only use a single static variable; the one found in the `CGII` class. This is enough since Java Card does not support multithreading, so we do not risk race conditions. It has the benefit of being an effective solution in terms of memory usage.

The call graph for our call graph integrity is generated using class hierarchy analysis. Another solution available in Soot is one they call Spark. We decided to use Soot's built-in class hierarchy analysis solution since we was unable to get a call graph from Spark when working on only a single class. For most Java Card applets there will probably be at least two classes, but since the call graph generated from both solutions seems to fit our needs we decided to go for the solution that worked on a single class as well.

```
// Method ID1 = 1, ID2 = 2
public int entryMethod() {
  0:  ...
  5:  iconst_3
  // #1: CGII.identifier
  6:  putstatic #1
  9:  aload_0
  // #2: methodInClass()
  10: invokevirtual #2
  13: iconst_4
  14: getstatic #1
  17: if_icmpne 36
  ...
  30: iconst_2
  31: putstatic #1
  34: iconst_0
  35: ireturn
  36: goto 36
}

// Method ID1 = 3, ID2 = 4
public void methodInClass() {
  0:  getstatic #1
  3:  iconst_3
  4:  if_icmpne 25
  ...
  20:  iconst_4
  21:  putstatic #1
  24:  return
  25:  goto 25
}
```

Listing 5.4: The assignment and check flow for call graph integrity. Roughly correspond to Listing 5.5.

```
// Method ID1 = 1, ID2 = 2
@EntryPoint
public int entryMethod() {
  ...
  CGII.identifier = 3;
  methodInClass();
  if (CGII.identifier != 4) {
    // Some error handling
  }
  ...
  CGI.identifier = 2;
  return 0;
}

// Method ID1 = 3, ID2 = 4
public void methodInClass() {
  if (CGII.identifier != 3) {
    // Some error handling
  }
  ...
  CGII.identifier = 4;
  return 0;
}
```

Listing 5.5: Assignment and check in Java.

```
Generate CGII class and add public static identifier to it
Get compilable methods cm
Assign unique IDs, mid1 and mid2, to each method in cm, where overwritten methods get
     the same IDs as the overwriting method
For each method m in cm
  Find edges out of m, m.out
  If m is not entry point
    Create check for mid1 at the beginning of the method
  For each method out in m.out
    If out is in cm
      For each invoke in method out
        If target of invoke is in cm
          Create assignment of identifier to mid1 of invoke before invoke
          Create check of identifier equals mid2 of invoke after invoke
  Create assignment of identifier to mid2 of m before each return
  Add kill statement to the end of m
```

Listing 5.6: Pseudo-code for call graph integrity.

5.5 Duplicator

The *Duplicator* phase implements the abstract method `internalTransform(Body body, String s, Map map)` which is invoked for every method in the `.class` file(s). The parameters given are a `Body` which is the method body containing Jimple statements, a string containing the phase name (“jtp.Duplicator”), and a `Map` from string to string defining the phase options such as for example “enabled:true”. This phase can be disabled by passing the command line option:

```
-p jtp.Duplicator enabled:false
```

The *Duplicator* does not duplicate branches in every method despite the `internal-Transform` method being invoked for every method. A special annotation has to exist for that method, which should be inserted by the programmer at source code level, see Listing 5.1. Furthermore, the extra check is only inserted into the `if` branch, not the `else` branch.

In Listing 5.7 the steps required to implement branch duplication for a single method’s body is given.

```
Given a Body b:
  For each if statement i in b:
    Recursively find the set of statements dup involved in is condition
    Duplicate dup right after i
    Duplicate i with inverted condition right after dup with branch to sensitive code
    Insert goto after duplicated i with a target to kill
  Insert a goto statement kill to itself at the end of b
```

Listing 5.7: Pseudo-code for branch duplication of `if` statements.

In Listing 5.7 we search through the method body for `if` statements. Because the intermediate language that we are transforming is Jimple which is in three address form, the `if` statement contains exactly two operands and is given on the form:

```
if op1 BINOP op2 branch
```

Where `branch` is the instruction to branch to if the condition is true. In order to find the list of instructions to duplicate, ***dup***, we conduct a definition-use analysis and find recursively the definitions of the two operands, meaning that if a variable `b` is used as the first operand and the definition of `b` is `b = x + y` we also include the definitions of `x` and `y`. The combined list of instructions to duplicate for both operands are inserted right after the `if` statement. Another `if` statement is inserted after these duplicated statements with an inverted condition compared to the original `if` statement but in this case if the condition is true we branch to the sensitive code. Otherwise, if false we branch to the endless loop.

An example of an implementation of branch duplication at bytecode level for a simple method body can be seen in Listing 5.8 and Listing 5.9. On the left the original bytecode for the method body is shown, and on the right the branch duplication has been inserted. Note the `goto` statement is inserted at the end as instruction 42 in Listing 5.9, which results in an endless loop. This instruction is jumped to from the inserted `goto` statement at instruction 20. In the example the instructions necessary to duplicate the `if` statement are instruction 13 and 14 in Listing 5.9.

```
public dk.aau.cs.test.ClassA();
Code:
 0: aload_0
 1: invokespecial #1 // Object."<init>":()V
 4: aload_0
 5: invokevirtual #2 // getInput:()I
 8: istore_1
 9: iload_1
10: ifge          24
13: getstatic     #3 // PrintStream;
16: ldc           #4 // String Error
18: invokevirtual #5 // println:(String;)V
21: goto          31
24: getstatic     #3 // PrintStream;
27: iload_1
28: invokevirtual #6 // println:(I)V
31: return
```

Listing 5.8: A simple method with a single `if` statement.

```
public dk.aau.cs.test.ClassA();
Code:
 0: aload_0
 1: invokespecial #21 // Object."<init>":()V
 4: aload_0
 5: invokevirtual #8  // getInput:()I
 8: istore_1
 9: iload_1
10: ifge          34
13: aload_0
14: invokevirtual #8  // getInput:()I
17: iflt          23
20: goto          42
23: getstatic     #13 // PrintStream;
26: ldc           #31 // String Error
28: invokevirtual #24 // println:(String;)V
31: goto          38
34: getstatic     #13 // PrintStream;
37: iload_1
38: invokevirtual #7  // println:(I)V
41: return
42: goto          42  // kill
```

Listing 5.9: Implementation of branch duplication in a simple method. The bold font bytecodes are the inserted bytecodes.

5.5.1 Switch statements

`if` statements are not the only conditional branching instruction that needs to be secured. `switch` statements also branch on some condition. The straight forward approach for implementing branch duplication for `switch` statements can be seen in Listing 5.10.

```
Given a switch statement s:
Recursively find the set of statements dup involved in ss condition
For each branch, ignoring the default branch:
  Insert dup after each branch (case)
  Insert if statement right after dup: if switchValue != CaseConstant goto kill
```

Listing 5.10: Pseudo-code for implementing branch duplication for `switch` statements.

Besides the steps in Listing 5.10 there are some post-processing steps to take as well, where the lookup table has to be updated to point to the first of the newly inserted statements instead. The steps in Listing 5.10 does not handle the specific situation when one switch case fall through to another case. In such situation we need to either consider each possible value in the cases fallen through as illustrated in below,

```
if (switchValue != caseConstant1 && switchValue != caseConstant2 ...)
```

or, simply avoid inserting the branch duplication for `switch` cases that can be fallen through to. In Section 5.9.2 we argue why we have chosen not to insert checks in cases which might be reached because of fallthrough.

5.5.2 Branch Dependent Condition

As described in Section 3.1.1 handling `if` statements where the condition involves a variable that obtain different values dependent on a prior control flow, see Listing 3.5, requires an extra effort to implement correctly. There is always the possibility to detect such situation and simply avoid duplicating the `if` statement to ensure a correct result. If the same approach as in Listing 5.7 was taken the result would be wrong as there is no way of determining the value of `x` at compile time and thus the computation of `x` cannot be duplicated. Simply reading the same variable again would not provide security in the case where the attacker has altered the value of `x` permanently rendering a double read useless.

Instead, we have chosen to introduce another variable, `x'`, for each operand of the `if` statement that can be determined to have multiple definitions. Then for every assignment of that variable in different non-overlapping scopes, we perform the calculations and assignment again to `x'`. The duplicated `if` statement then uses the prime variables instead.

An example of such implementation can be seen in Listing 5.11 and Listing 5.12, where on the left the original bytecode is shown, and on the right the branch duplication is implemented.

```
public dk.aau.cs.test.ClassA();
Code:
 0: aload_0
 1: invokespecial #1 // Object."<init>":()V
 4: aload_0
 5: invokespecial #2 // getValue:()I
 8: istore_1
 9: iconst_4
10: istore_3
11: iload_1
12: lookupswitch  { // 2
             20: 40
             50: 45
        default: 50
    }
40: iconst_3
41: istore_2
42: goto          52
45: iconst_4
46: istore_2
47: goto          52
50: iconst_1
51: istore_2
52: iload_2
53: iload_3
54: if_icmplt     68
57: getstatic     #3 // PrintStream;
60: ldc           #4 // String Good
62: invokevirtual #5 // println:(String;)V
65: goto          76
68: getstatic     #3 // PrintStream;
71: ldc           #6 // String Error
73: invokevirtual #5 // println:(String;)V
76: return
```

Listing 5.11: An example of branch dependent condition of `if` statement.

```
public dk.aau.cs.test.ClassA();
Code:
  0: aload_0
  1: invokespecial #19 // Object."<init>":()V
  4: aload_0
  5: invokespecial #18 // getValue:()I
  8: lookupswitch  { // 2
             20: 36
             50: 52
        default: 68
    }
 36: aload_0
 37: invokespecial #18 // getValue:()I
 40: bipush        20
 42: if_icmpne     102
 45: iconst_3
 46: istore_0
 47: iconst_3
 48: istore_1
 49: goto          72
 52: aload_0
 53: invokespecial #18 // getValue:()I
 56: bipush        50
 58: if_icmpne     102
 61: iconst_4
 62: istore_0
 63: iconst_4
 64: istore_1
 65: goto          72
 68: iconst_1
 69: istore_0
 70: iconst_1
 71: istore_1
 72: iload_0
 73: iconst_4
 74: if_icmplt     96
 77: iload_1
 78: iconst_4
 79: if_icmpge     85
 82: goto          105
 85: getstatic     #11 // PrintStream;
 88: ldc           #21 // String Good
 90: invokevirtual #25 // println:(String;)V
 93: goto          101
 96: getstatic     #11 // PrintStream;
 99: ldc           #30 // String Error
101: invokevirtual #25 // println:(String;)V
104: return
105: goto          102 // kill
```

Listing 5.12: Implementation of additional variable for branch dependent conditions in `if` statement.

In Listing 5.12 the bold font bytecodes are the extra inserted variable assignments and the duplicated `if` statement using that extra variable. It can be seen that the `store` instructions at line 48, 64, and 71 stores the value in another local than the original `store` instructions at line 46, 62, and 69, and that the new local is being loaded for the duplicated `if` instruction at line 77.

5.5.3 Duplicator versus Call Graph Integrity

As the *Call Graph Integrity* phase is run before the *Duplicator* phase, the inserted check after each method call should not be duplicated by the *Duplicator* phase. The normal behaviour of the *Duplicator* is to duplicate the instructions needed to calculate the operands of an `if` statement, but in the case of the call graph integrity check this would be wrong. Consider the simple example in Listing 5.13, where `someCall()` assigns the static variable `CGII.identifier = 2` right before returning. The *Duplicator* does not recognize the method call as a dependency of the operands of the `if` statement, and if it were to duplicate the `if` statement it would only include the assignment of `CGII.identifier` right before the method call and thus in effect do the comparison `1 != 2` as can be seen Listing 5.14.

```
CGII.identifier = 1;
someCall();
if (CGII.identifier == 2) {
  // normal execution
} else {
  // error handling
}
```

Listing 5.13: Simple call graph integrity implementation.

```
CGII.identifier = 1;
someCall();
if (CGII.identifier == 2) {
  CGII.identifier = 1;
  if (CGII.identifier != 2) {
    // error handling
  } else {
    // normal execution
  }
} else {
  // error handling
}
```

Listing 5.14: Straight-forward duplication of `if` statement. Bear in mind that when decompiling bytecode the conditions are inverted.

The situation is handled by looking for a tag, set by the *Call Graph Integrity* phase, indicating that the `if` statement should not be duplicated.

Another similar situation is when a method call is part of the condition and the *Duplicator* can duplicate the call, which is determined by the purity analysis. In such situation the necessary call graph integrity checks normally inserted by the *Call Graph Integrity* phase would not be inserted as the *Duplicator* does not know the identifiers of the invoked method. This is handled by letting the *Call Graph Integrity* phase decorate the invoke statement with the two identifiers, such that the *Duplicator* phase is able to insert the necessary checks.

5.6 Java Card Class Initializer

We encountered a problem in the way Soot initialize static arrays. According to the specification [1, JCVM Section 2.2.4.6], the `<clinit>` method is limited to the following instructions: `iconst_[m1,0-5]`, `[b|s]ipush`, `ldc[_w]`, `aconst_null`, `newarray`, `dup`, `[b|i|s]astore`, `putstatic`, and `return`. Since the use of locals is allowed for normal Java applications, Soot uses these instructions, e.g., `aload` and `astore`. Our phase fixes this issue by removing `astore` instructions and replacing `aload` instructions with `dup` instructions.

An example of how Soot writes the method can be seen in Listing 5.15 and the result after this phase in Listing 5.16.

```
static {};
  flags: ACC_STATIC
    Code:
      stack=3, locals=1, args_size=0
         0: bipush        11
         2: newarray       byte
         4: astore_0
         5: aload_0
         6: iconst_0
         7: bipush        -96
         9: bastore
        10: aload_0
        11: iconst_1
        12: iconst_0
        13: bastore
        ...
        53: aload_0
        54: bipush        10
        56: iconst_1
        57: bastore
        58: aload_0
        59: putstatic     #23
        62: return
```

Listing 5.15: An example of a <clinit> method written by Soot.

```
static {};
    flags: ACC_STATIC

    Code:
      stack=4, locals=1, args_size=0
         0: bipush        11
         2: newarray       byte
         4: dup
         5: iconst_0
         6: bipush        -96
         8: bastore
         9: dup
        10: iconst_1
        11: iconst_0
        12: bastore
        ...
        52: dup
        53: bipush        10
        55: iconst_1
        56: bastore
        57: putstatic     #23
        60: return
```

Listing 5.16: An example of a <clinit> method written by Soot and corrected by this phase.

We implement this by finding `astore` instructions and all following `aload` instructions associated with the `astore` instruction. Then we remove the `astore` instructions and replaces all the `aload` instructions by a `dup` instruction, except for the last load, which is instead removed. In this way we avoid using both `astore` and `aload` instructions while still having the same result.

5.7 Java Card Integer Support

In this phase we ensure that no local variables are stored as integers. The problem is that Soot sometimes stores a local `integer`. Given the fact that the `integer` type on Java Card is optional confer [1, JCVM Section 2.2.3.1] the developer can pass a commandline option to disable this conversion if the use of integer types is deliberate. The commandline option is:

```
-p bb.JavaCardIntegerSupport enabled:false
```

We eliminate `integers` by iterating through each Baf instruction, looking for `store` and `load` instructions. For each of these instructions we check if the type of the instruction is `integer`, and if that is the case, we convert it to a `short` type instead.

An example of such a case can be seen in Listing 5.17.

```
...
// Value of someVar depend on non-duplicable calculation
if (someVar % 2 == 0) {
  ..
}
```

Listing 5.17: Faulty code when duplicated.

```
...
// Value of someVar depend on non-duplicable calculation
short someVar2 = (short)(someVar % 2);
if (someVar2 == 0) {
  ..
}
```

Listing 5.18: Correct code when duplicated.

The problem in Listing 5.17 is that when the calculation of `someVar` cannot be duplicated, but the value should still be used in the duplicated `if` statement, the value will be stored in a local variable. Since modulo instruction, `irem`, leaves an `integer` on the operand stack, Soot will store the value as integer and load the stored value for the duplicated `if` statement. This may result in problems on Java Card since it does not generally support `integers`. Even though this could be solved by the programmer by storing the calculation in a `short` variable, as seen in Listing 5.18, we decided that this way of programming should not be considered wrong in combination with our tool.

5.8 Applied Optimizations

In order to produce efficient bytecode from Jimple code, we enable some of the built-in optimization phases. The enabled phases in the `jop` pack can be seen in Figure F5-1 as Copy Propagator, Constant Propagator and Folder, Dead Assignment Eliminator, and Unused Local Eliminator. The phases called Unconditional Branch Folder and Unreachable Code Eliminator has been disabled as these phases removes the inserted kill statement at the end of the method body. Removing unused variables and dead assignments does not harm the implemented countermeasure. For the *Duplicator* the duplicated statements are not used in any other way than the original code, and thus if the code did not contain unused local variables or dead assignments before, the rewritten code does not contain any of them. For call graph integrity the only added local is `checkIdentifier` which cannot be unused. The Copy Propagator phase does cascading copy propagation which replaces copies of a variable with the direct access to that variable where possible, e.g. `x = y; z = 42 + x;` would be replaced with `z = 42 + y;`. The Constant Propagator and Folder evaluates expressions that on compile-time can be determined to be constants, e.g. `x = 3 * 4;` would be written as `x = 12;`.

Without applying the Copy Propagator phase we encountered unnecessary use of load- and store-bytecodes when generating bytecode from Jimple. A pattern consisting of three consecutive bytecodes: `load_1`, `store_1`, `load_1`, makes no sense because it would have the same effect as a single `load_1`. Applying the above optimizing phases on Jimple eliminated such bytecode pattern and further reduced the size of the program.

In the `bb` pack we have disabled the *Local Packer* phase. This is because Java Card does not allow for locals to change types during a method. The *Local Packer* will attempt to minimize the number of locals used in a method by reusing locals when they are no longer used, potentially changing the type of the local.

5.9 Shortcuts

While making this tool a number of shortcuts have been taken. In this section we will describe these shortcuts as well as what problems it might cause.

5.9.1 Side Effect on Fields

When we encounter an `if` statement whose condition is depending on an instance or class variable, we do not create a new object for the duplication. We rather use the same field and attempt to redo the calculations needed for the field. In certain situations we are, however, unable to effectively calculate the correct result. Take the example in Listing 5.19.

```
public void someMethod() {
  instanceVar1 = 5;
  someOtherMethod();
  if (instanceVar1 > 5) {
    ...
  }
}

public void someOtherMethod() {
  instanceVar1 = 10;
}
```

Listing 5.19: Example where duplication will calculate the wrong result.

```
public void someMethod() {
  instanceVar1 = 5;
  someOtherMethod();
  if (instanceVar1 > 5) {
    instanceVar1 = 5;
    if (instanceVar1 <= 5) {
      // Error handling
    } else {
      ...
    }
  }
}

public void someOtherMethod() {
  instanceVar1 = 10;
}
```

Listing 5.20: Incorrectly duplicated method.

The problem in this case is that the side effect of the method `someOtherMethod()` is to change the value of `instanceVar1` which will not be reflected when we perform the duplication as seen in Listing 5.20, since we do not know that the call to `someOtherMethod()` changes the value of `instanceVar1`. We have identified two potential solutions for this problem. The first is to look at all the instructions between the declaration of the variable and the `if` statement, whose condition uses the variable, to see if any impure method call takes place. If such a call is found the recalculation of the variable will be skipped and the existing variable will instead be loaded onto the stack, and used by the second `if` statement. This solution will work, but some safe recalculations will probably be skipped if the method that is called does not result in a change in the used fields. A better solution is to recursively check the method calls between the declaration and `if` statement, to see which fields are changed. In this way it can be determined if it is safe to recalculate the value.

5.9.2 Switch Fallthrough

Currently we do not create any checks in a case of a `switch` statement if we detect that it is possible that execution can fall through from the previous case. If we were to do so, we would need to know which cases it can fall through, as well as which values the condition should have for those cases. We should then create a check for each possible value, for each case in the `switch`. A simple `switch` with 5 cases and a default case, with fallthrough from case 1 to 3 can be seen in Listing 5.21 and a secured version in Listing 5.22.

```
public void someMethod(short var1) {
  switch(var1) {
    case 1:
      ...
    case 2:
      ...
    case 3:
      ...
      break;
    case 4:
      ...
      break;
    case 5:
      ...
      break;
    default:
      ...
  }
}
```

Listing 5.21: Example of `switch` statement with 5 cases and default case.

```
public void someMethod(short var1) {
  switch(var1) {
    case 1:
      if (var1 != 1) {
        // Error handling
      }
      ...
    case 2:
      if (var1 != 1 && var1 != 2) {
        // Error handling
      }
      ...
    case 3:
      if (var1 != 1 && var1 != 2 &&
          var1 != 3) {
        // Error handling
      }
      ...
      break;
    case 4:
      if (var1 != 4) {
        // Error handling
      }
      ...
      break;
    case 5:
      if (var1 != 5) {
        // Error handling
      }
      ...
      break;
    default:
      ...
  }
}
```

Listing 5.22: Condition check for fallthrough.

On bytecode level the secured version will have an additional 8 `if` statements, one for each check. Furthermore, according to [24] is it bad practice to omit the `break` or `return` statement at the end of each case, and thereby allowing the flow to fall through to the next case. Because of the potentially large code size overhead, as well as the recommendation to end all cases with a `break` or `return` statement, we have decided not to implement these checks. This means that cases which can be reached because of fallthrough are not secured.

CHAPTER 6

Testing

In this chapter we describe how testing of the rewriting tool is conducted as well as which errors were found and corrected because of the test. Furthermore this chapter describes which experiments we have done.

6.1 Test

In order to test that our tool produces the correct output a number of tests are conducted. We test that for a certain input, we get the same output from the original method and the rewritten method. We do this by creating a number of test methods, each testing different cases in order to cover different constructions of bytecode, e.g., `if` statements, `while` loops, and `switch` statements. Each method takes a `short` as input and returns a `short` as output. We then run each test method 32768 times where the input for each call is the numbers from 0 to 32767. The same number is used as input for both the original method and the rewritten method, this means that for a correctly rewritten program, we will see the same output from both methods. We refer to this as our *output comparing test*.

We also test that our tool produces the same output each time. We do this by manually checking the rewritten class file for errors in the program on bytecode-level. If the class is deemed correct we create a copy of it and uses it as the reference class. Whenever we rewrite a new class we run the tests, which then compares the newly created class with the reference class. Because of the call graph integrity the hashes of both classes cannot be compared. This is because the IDs assigned to each method during the call graph integrity phase may change from rewrite to rewrite. We therefore parse the reference class into Jimple. Then for each method we compare the number of instructions. If there are the same number of instructions we compare the `toString()` of each instruction. For this to work we replace all class names and numbers by `x` in the `toString()` output. This is necessary since the IDs will be different, and the class name of the reference class is different from the class being tested. By replacing all numbers we lose accuracy in the test, but we deem this to be acceptable since the overall structure is still tested. If we do not change the way we rewrite the class or the structure of the class we are rewriting, we expect to get the same structure for each rewrite. We refer to this as our *rewrite-monitor* as it monitors the rewritten program and fails when a difference in the rewriting occurs. This process is illustrated in Figure F6-1. When the test fails a diff-tool can be used to see the difference between the *Current Accepted Class* and the *Newly Rewritten Class* (after `javap` command).

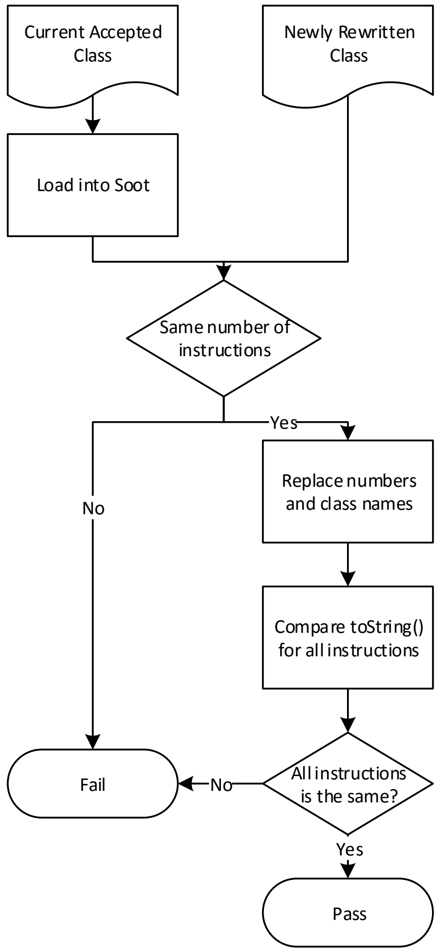

Figure F6-1: Flowchart describing the process of the *rewrite-monitor* test.

6.1.1 Test Results

As a result of *output comparing test* an error was revealed showing that we did not handle `switch` statements with fallthrough correctly. The tool inserted duplicated branches for each `switch` statement case, but in the case of fallthrough every subsequent duplicated branch would fail, as the original case value would be wrong. When the duplicated branch condition is not met, the program end in an error state, which in this case is an endless loop. Therefore our tests would never finish when containing `switch` statements with fallthrough.

We also discovered that, because of restructuring the implementation of the rewriting tool, the look-up tables of a `switch` statement was not updated correctly in the rewritten code. This error revealed it self through the *output comparing test* as the execution of the rewritten would have a different control flow.

Another issue we discovered using our *rewrite-monitor* was that the purity analysis that is implemented as part of the Soot library did not render methods creating objects or persistent arrays on the heap as impure. This is not incorrect by the Java language specification because the JVM implements a garbage collector to free such memory again. This is not the case for the JCVM which specification does not state that at garbage collector should be implemented and as such we are not guaranteed that objects and persistent arrays are collected as garbage.

6.2 Experiments & Metrics

Besides testing that our tool rewrites correctly and that our test run without errors, we are able to test our tool against a Java Card applet by generating and verifying a CAP file. The Java Card SDK provide tools for converting class files into a CAP file and for verifying that the CAP file is consistent w.r.t. the Java Card Virtual Machine specification and that it is consistent with a context of a Java Card enabled device[4].

Our experiment in this regard is to take as a basis a sample Java Card Applet, convert it into a CAP file and then verify that CAP file. Next step is to use the tool to implement countermeasures in the same applet and do the process over again to verify that it is still consistent.

Furthermore we gather metrics about the applet in that we count the number of certain instructions before and after implementing the countermeasures. The metrics gathered are the number of writes to EEPROM (i.e. the number of `putstatic` and `putfield` instructions), the size of the class file in bytes, the number of invoke instructions, that is `invokespecial`, `invokevirtual`, `invokeinterface`, and `invokestatic`, and lastly the number of `load` and `store` instructions.

The difference in each of these metrics tell us indirectly something about the performance of the applet whether that being memory footprint or running time.

Load and store instructions operate on memory located in RAM and only poses minor impact on the running time of the applet. For every `invokevirtual`, `invokespecial`, and `invokeinterface` instruction a method is resolved at runtime by the JCVM by looking an index up in the constant pool while, among other checks, also performing a firewall check that the method is allowed invocation from the current context. This indicates that the instructions are expensive w.r.t. running time because there are runtime lookups in tables depending on the type of reference.

The instruction `invokestatic` is less expensive w.r.t. running time because there are less checks at runtime[1, JCVM Section 7.5.56]. For instance there is no firewall checking for static invocation, only a lookup in the constant pool.

Counting the number of writes to EEPROM both tells us something about the lifespan of the Java Card, because EEPROM has a limited number of writes, but also something about the running time as writing to EEPROM is around 10,000 times slower than writes to RAM[29].

Lastly, the size of the `.class` file before and after tells us something about the storage usage and may indicate whether there is space for the applet.

6.2.1 Results

Taking an excerpt of the sample applets provided with the Java Card SDK version 2.2.2, we have done our experiments on `sigMsgRecApplet`, `transitApplet`, `AccountAccessor`, `ConnectionManager`, and `SamplePasswdBioApppet`. Each of the applets has been converted to a `.cap` file with the provided `converter` tool, after which the `.cap` files has been verified by the provided `verifycap` tool.

	sigMsg-RecApplet		transit-Applet		Account-Accessor		Connection-Manager		SamplePasswd-BioApplet	
	orig.	rewrit.	orig.	rewrit.	orig.	rewrit.	orig.	rewrit.	orig.	rewrit.
CAP verified	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EEPROM writes	11	14	21	45	8	10	2	8	2	4
Size in bytes	4716	4844	8042	9723	3650	3745	3796	4076	2444	2483
Invokevirtuals	15	15	26	30	9	10	8	9	12	13
Invokestatics	10	10	52	52	13	13	19	21	4	4
Invokeinterfaces	12	14	3	3	0	0	0	0	2	3
Invokespecials	5	5	18	18	4	4	12	12	2	2
Loads	100	118	263	362	85	112	67	94	38	48
Stores	150	150	51	60	31	32	19	19	19	18

Table T6-1: Table of metrics gathered for Java Card applets both before and after implementing countermeasures.

Table T6-1 show metrics gathered for each of the sample applets before (orig.) and after (rewrit.) applying the rewriting tool. The `transitApplet` is of particular interest since it requires almost one hundred more loads and more than doubles the writes to EEPROM. Furthermore the size has increased by approximately 20 %. Examining further the produced bytecode of the rewriting tool, see Listing A.3, we see that a write to the static CGI identifier variable is inserted 24 times, which explains the extra writes to EEPROM. Furthermore, a total of 68 branching instructions have been duplicated, which explains the high number of loads because every duplicated branching instruction implies two or more instructions to either perform the calculation of the condition variables again or simply load them.

Whether the implied overhead of the implemented countermeasures are acceptable can only be the judgement of the applet developer. In the example of `transitApplet`, if the running time of the applet is an issue the developer might decide not to implement the call graph integrity countermeasure to reduce the time consuming EEPROM writes.

CHAPTER 7

Discussion

While developing this product a number of problems have been encountered which is worth discussing. Early in the development we encountered a problem with the purity analysis when working on `ArrayList`. If this was used in a program, the purity analysis would take a very long time to finish. On our developing computers it took more than 3 hours, before finally crashing due to `OutOfMemoryError`. At first this was considered a problem, but after a bit of research it was concluded that `ArrayList` is not part of Java Card, and therefore not a problem relevant for our tool.

During testing of the tool we encountered a serious problem. We were unable to convert a lot of applets after rewriting them, because of type errors. We learned that the converter requires that the class files contain debug information, specifically the `LocalVariableTable` attribute is needed which tells the type of the local variables in each method. We concluded that the version of Soot we used, version 2.5.0, was unable to add this information to the rewritten class files, which meant that we were unable to convert a lot of applets after rewriting the class files. We decided to use the newer version, version 2.6.0, which unfortunately, as of writing, is still under development. This version uses ASM[7] to create the bytecode, instead of Jasmin[17], which is able to write the `LocalVariableTable` attribute. It does, however, also mean that it was not completely bug-free. For a few applets the definition-use analysis ended up in an endless loop, which meant that we were unable to rewrite these applets. We decided that this problem did not outweigh the problem with the missing `LocalVariableTable`. Both because we only encountered the endless loop in a single applet, while the missing table caused problems for more applets, but also because of a hope that the problem with the analysis will be resolved when Soot 2.6.0 is released in a final version.

Another problem we encountered when converting rewritten applets was that we got type errors for local variables. We localized this to be because of the "Local Packer" optimization in the Baf body creation pack. This optimization intends to reduce the number of local variables used by a method by reusing locals when they are no longer used. This means that the type of a local memory slot may change throughout a method. Soot is intended for Java where this is not a problem, but for Java Card this is a problem. Therefore we disabled this optimization which meant that the applets after rewriting may have used a few more locals than before rewriting. This was not an optimal solution, but necessary to make the rewritten applets work on Java Card.

We generally had a few problems because of Soot's focus on the Java specification, rather than the Java Card specification. Some of these problems could be solved by enabling or disabling phases in the framework. Other problems required that we added extra phases to the Baf body creation pack, as mentioned in Section 5.6 and Section 5.7.

Some Java Card applets used Remote Method Invocation (RMI), which gave problems for our call graph integrity countermeasure. Soot was unable to detect which methods were called when RMI was used. This meant that we were unable to implement the call graph integrity countermeasure for those applets. We decided that in the time frame of this project, our time would be better spent elsewhere.

We decided for our call graph integrity to use a static field in a separate class to store the expected ID when calling methods. This solution could be improved by using a transient array instead. In this way the read and write speed would be improved, because the value would be stored in RAM instead of the EEPROM. In order for this to work, we would have to call the `makeTransientShortArray()` method from the Java Card API. This should happen in the `install` method of the applet in order to only create the array once.

For this project we focused on Java Card 2, since this version was widely used. There exists a new version, version 3, which has some new capabilities, e.g. a volatile heap, while still being compatible with version 2[2]. This decision also influenced how we handled object creation, since Java Card does not necessarily use garbage collection[1, JCVM Section 3.3]. We therefore did not duplicate objects, even if the object was part of a condition. If our focus was on Java Card 3, the extra volatile heap would allow us to duplicate some objects, since they would be removed when the power source was removed.

7.1 Conclusion

In Section 1.1 we list a number of questions we would like to be answered throughout this project.

The first question is:

> *What is required to automatically insert countermeasures in a Java Card applet?*

Because of our choice to work on class files a way of reading, manipulating, and writing bytecode is necessary. Furthermore, a number of analyses are needed to get the necessary information to properly insert the countermeasures. For our project we needed a call graph, definition-use and use-definition chains, and a purity analysis. The call graph is needed for our call graph integrity in order to know the intra-procedural control flow of the applet. The chains are used by our branch duplication to know which variables we should duplicate to perform the needed calculations. Finally the purity analysis is used to decide whether we can safely duplicate calls to a method.

The second question is:

> *What has to be considered when implementing branch duplication and call graph integrity?*

We learned during the project that for especially branch duplication there are a lot of cases to take into account. Generally it is important to think about when you may end up overwriting a value in the duplication that will cause the program to break. This is for example the case for loops, if the loop condition is part of a nested branch. Then the needed calculations may not be feasible since it may cause an endless loop. The considerations we have made for branch duplication can be read in Section 3.1.1. For call graph integrity there is an important consideration to make, namely how to handle polymorphism. Another consideration is that since Java Card is very limited on resources, the solution should not use too much memory or be too computationally expensive. Our consideration for call graph integrity can be found in Section 3.2.1.

The last question is:

> *How much of the process can be automated?*

For branch duplication the whole process can be automated. Without any intervention from the developer the process can be applied to all methods in a class. If, however, only some methods should be processed, or even only some of the statements in a method should be processed, it might be necessary for the developer to help the tool in deciding which statements should be duplicated. This can for example be done using a special naming convention or annotation for the methods that should be processed. Generally it can be hard to automatically determine the sensitivity of a piece of code.

Bibliography

[1] Mar 2006. URL http://www.oracle.com/technetwork/java/javacard/specs-138637.html.

[2] Sep 2011. URL http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html.

[3] Jul 2015. URL http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.5.1-doc/api/index.html.

[4] May 2016. URL https://docs.oracle.com/javacard/3.0.5/guide/verifying_cap_files.htm.

[5] Olivier Benot. *Encyclopedia of Cryptography and Security*, chapter Fault Attack, pages 452–453. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_505. URL http://dx.doi.org/10.1007/978-1-4419-5906-5_505.

[6] Apache Commons. Bcel. https://commons.apache.org/proper/commons-bcel/, Marts 2016. visited: 23. Marts 2016.

[7] OW2 Consortium. Asm. http://asm.ow2.org/, Marts 2016. visited: 23. Marts 2016.

[8] Julian Dolby and Manu Sridharan. *Static and Dynamic Program Analysis Using WALA*. 2010. URL http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf.

[9] Julian Dolby, Manu Sridharan, and Stephen Fink. T.j. watson libraries for analysis (wala). URL http://wala.sourceforge.net/.

[10] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. *Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. Proceedings*, chapter Efficient Inference of Static Types for Java Bytecode, pages 199–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45099-3. doi: 10.1007/978-3-540-45099-3_11. URL http://dx.doi.org/10.1007/978-3-540-45099-3_11.

[11] Sable Research Group. Soot - a framework for analyzing and transforming java and android applications. http://sable.github.io/soot/, Marts 2016. visited: 16. Marts 2016.

[12] Sable Research Group. Packs and phases in soot. https://github.com/

Sable/soot/wiki/Packs-and-phases-in-Soot, Marts 2016. visited: 17. Marts 2016.

[13] Sable Research Group. Soot command-line options. https://ssebuild.cased.de/nightly/soot/doc/soot_options.htm, Marts 2016. visited: 23. Marts 2016.

[14] René Rydhof Hansen. personal communication.

[15] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas P. Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for java. *CoRR*, abs/1007.3353, 2010. URL http://arxiv.org/abs/1007.3353.

[16] Chien-Hung Liu, D. C. Kung, and Pei Hsia. Object-based data flow testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 7–16, 2000. doi: 10.1109/APAQ.2000.883773.

[17] Jonathan Meyer. Jasmin home page, Oct 2004. URL http://jasmin.sourceforge.net/.

[18] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and pin is broken. In *2010 IEEE Symposium on Security and Privacy*, pages 433–446, May 2010. doi: 10.1109/SP.2010.33.

[19] Ed Ort. Developing a java card applet, Aug 2001. URL http://www.oracle.com/technetwork/java/embedded/javacard/documentation/applet-136808.html.

[20] Eric Poll. Embedded software security, July 2015. URL https://www.cs.ru.nl/E.Poll/talks/ISSIPS_erik_poll.pdf.

[21] Ando Saabas and Tarmo Uustalu. Type systems for optimizing stack-based code. *Electronic Notes in Theoretical Computer Science*, 2007. URL http://set.ee/publications/bytecode07.pdf.

[22] Alexandru Salcianu and Martin Rinard. A combined pointer and purity analysis for java programs. *CSAIL Technical Reports July 1, 2003*, 2004. URL http://hdl.handle.net/1721.1/30470.

[23] Darlene Storm. Hack to steal cars with keyless ignition: Volkswagen spent 2 years hiding flaw, Aug 2015. URL http://www.computerworld.com/article/2971826/cybercrime-hacking/hack-to-steal-cars-with-keyless-ignition-volkswagen-spent-2-years-hiding-flaw.html.

[24] David Svoboda. Msc17-c. finish every set of statements associated with a case label with a break statement, Feb 2016. URL https://www.securecoding.cert.org/confluence/display/c/MSC17-C.+Finish+every+set+of+statements+associated+with+a+case+label+with+a+break+statement.

[25] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and

Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL `http://dl.acm.org/citation.cfm?id=781995.782008`.

[26] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. *Compiler Construction: 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25 – April 2, 2000 Proceedings*, chapter Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?, pages 18–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-46423-5. doi: 10.1007/3-540-46423-9_2. URL `http://dx.doi.org/10.1007/3-540-46423-9_2`.

[27] Raja Vallée-Rai. Soot: A java bytecode optimization framework, October 2000.

[28] Abe White. Serp - overview. `http://serp.sourceforge.net/`, Marts 2016. visited: 23. Marts 2016.

[29] Kenneth Cox Wolfgang Rankl. *Smart Card Applications: Design models for using and programming smart cards*, chapter Implementation Patterns, page 124. Wiley, 2007. ISBN 978-0-470-05882-4.

[30] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger. A high performance java card virtual machine interpreter based on an application specific instruction-set processor. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 270–278, Aug 2014. doi: 10.1109/DSD.2014.47.

APPENDIX A

Transit Applet

In the following a complete sample applet is given first in source code, then in bytecode as produced by the `javap` tool provided with the Java SDK excluding the constant pool, and lastly a bytecode output of the applet with both call graph integrity and branch duplication implemented.

```java
/*
 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package com.sun.javacard.samples.transit;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.JCSystem;
import javacard.framework.OwnerPIN;
import javacard.framework.Util;
import javacard.security.DESKey;
import javacard.security.KeyBuilder;
import javacard.security.RandomData;
import javacard.security.Signature;
import javacardx.crypto.Cipher;

/**
 * This applet implements the on-card part of a transit system solution. The
 * on-card applet and the off-card applications (transit terminal and POS
 * terminal) use a mutual authentication scheme based on a dynamically generated
 * DES session key to ensure data integrity and origin authentication during a
 * session.
 *
 * When interacting with a POS terminal, the account maintained on the card can
 * be credited or queried for the current balance.
 *
 * When interacting with a transit terminal, the transit system entry and the
 * exit events are checked for consistency and processed - the account
 * maintained on the card is debited upon proper exit from the transit system.
 *
 * Design notes:
 * - This sample transit applet does not account for any admin or self-admin use cases such as
 * resetting the card of a transit system user when it is in an inconsistent transit
 * state. Such an inconsistent state can, for example, result from the user jumping the gates when
```

```
 * the turnstile is out of order...
 * - This sample transit applet does not account for any system-wide transactional
 * operations. For example, during a credit operation, if the user removes his card
 * just after the balance has been updated but before the APDU response gets to
 * the terminal, the account on the card will remain credited but the terminal will
 * only be able to detect an IO error b/w the card and the card reader.
 * - The constants defined for this class should have been shared through
 * an additional class or interface with the terminal code
 * (see com.sun.javacard.clientsamples.transit.Constants).
 * - This applet could be refactored so that the mutual authentication code
 * be moved in a base abstract class and the transit system specific behavior be
 * implemented in a subclass of this base class. This refactoring would facilitate
 * the reuse of the mutual authentication scheme in other application domain.
 */
public class TransitApplet extends Applet {

    // Codes of INS byte in the command APDU header

    /**
     * INS value for ISO 7816-4 VERIFY command
     */
    final static byte VERIFY = (byte) 0x20;

    /**
     * INS value for INITIALIZE_SESSION command
     */
    final static byte INITIALIZE_SESSION = (byte) 0x30;

    /**
     * INS value for PROCESS_REQUEST command
     */
    final static byte PROCESS_REQUEST = (byte) 0x40;

    // Tags for TLV records in PROCESS_REQUEST C-APDU

    /**
     * TLV Tag for PROCESS_ENTRY request
     */
    final static byte PROCESS_ENTRY = (byte) 0xC1;

    /**
     * TLV Tag for PROCESS_EXIT request
     */
    final static byte PROCESS_EXIT = (byte) 0xC2;

    /**
     * TLV Tag for CREDIT request
     */
    final static byte CREDIT = (byte) 0xC3;

    /**
     * TLV Tag for GET_BALANCE request
     */
    final static byte GET_BALANCE = (byte) 0xC4;

    // Offsets of TLV components in PROCESS_REQUEST C-APDU [CLA, INS, P1, P2, LC
    // T L V...]

    /**
     * TLV tag offset
     */
    final static short TLV_TAG_OFFSET = ISO7816.OFFSET_CDATA;

    /**
     * TLV length offset
```

```
     */
    final static short TLV_LENGTH_OFFSET = TLV_TAG_OFFSET + 1;

    /**
     * TLV value offset
     */
    final static short TLV_VALUE_OFFSET = TLV_LENGTH_OFFSET + 1;

    /**
     * Maximum allowed balance
     */
    final static short MAX_BALANCE = (short) 500;

    /**
     * Minimum balance to start transit
     */
    final static short MIN_TRANSIT_BALANCE = (short) 10;

    /**
     * Maximum amount to be credited
     */
    final static short MAX_CREDIT_AMOUNT = (short) 100;

    /**
     * Maximum number of incorrect tries before the PIN is blocked
     */
    final static byte MAX_PIN_TRIES = (byte) 0x03;

    /**
     * Maximum PIN size
     */
    final static byte MAX_PIN_SIZE = (byte) 0x08;

    /**
     * SW bytes for PIN verification failure
     */
    final static short SW_VERIFICATION_FAILED = 0x6300;

    /**
     * SW bytes for PIN validation required
     */
    final static short SW_PIN_VERIFICATION_REQUIRED = 0x6301;

    /**
     * SW bytes for invalid credit amount (amount > MAX_CREDIT_AMOUNT or amount <
     * 0)
     */
    final static short SW_INVALID_TRANSACTION_AMOUNT = 0x6A83;

    /**
     * SW bytes for maximum balance exceeded
     */
    final static short SW_EXCEED_MAXIMUM_BALANCE = 0x6A84;

    /**
     * SW bytes for negative balance reached
     */
    final static short SW_NEGATIVE_BALANCE = 0x6A85;

    /**
     * SW bytes for wrong signature condition
     */
    final static short SW_WRONG_SIGNATURE = (short) 0x9105;

    /**
```

```
     * SW bytes for minimum transit balance not met
     */
    final static short SW_MIN_TRANSIT_BALANCE = (short) 0x9106;

    /**
     * SW bytes for invalid transit state
     */
    final static short SW_INVALID_TRANSIT_STATE = (short) 0x9107;

    /**
     * SW bytes for success, used in MAC
     */
    final static short SW_SUCCESS = (short) 0x9000;

    /**
     * Unique ID length
     */
    final static short UID_LENGTH = (short) 8;

    /**
     * DES key length in bytes
     */
    final static short LENGTH_DES_BYTE = (short) (KeyBuilder.LENGTH_DES / 8);

    /**
     * Host and card challenge length (note: (2 * CHALLENGE_LENGTH) * 8 ==
     * KeyBuilder.LENGTH_DES
     */
    final static short CHALLENGE_LENGTH = (short) 4;

    /**
     * MAC length as generated by Signature.ALG_DES_MAC8_ISO9797_M2
     */
    final static short MAC_LENGTH = (short) 8;

    /**
     * Unique ID
     */
    private byte[] uid;

    // Signature/key objects

    /**
     * Cipher used to encrypt - using the static DES key - the derivation data
     * to form the session key
     */
    private Cipher cipher;

    /**
     * DES static key, shared b/w host and card
     */
    private DESKey staticKey;

    /**
     * 4-bytes Card challenge
     */
    private byte[] cardChallenge; // Transient

    /**
     * 8-bytes key derivation data, generated from the host challenge and the
     * card challenge
     */
    private byte[] keyDerivationData; // Transient

    /**
```

```java
     * 8-bytes session key data, generated from the derivation data
     */
    private byte[] sessionKeyData; // Transient

    /**
     * DES session key, generated from the derivation data
     */
    private DESKey sessionKey; // Transient key

    /**
     * Indicates whether or not to use transient session key - for performance
     * measurement only
     */
    private boolean useTransientKey = true;

    /**
     * Signature initialized with the DES key and used to verify incoming
     * messages and to sign outgoing messages
     */
    private Signature signature;

    /**
     * Random data generator, used to generate the card challenge
     */
    private RandomData random;

    /**
     * The user PIN
     */
    private OwnerPIN pin;

    /**
     * The balance
     */
    private short balance = (short) 0;

    /**
     * The entry ststion id, set to (-1) when not in transit
     */
    private short entryStationId = (short) -1;

    /**
     * A correlation id that may be used by the backend system to correlate
     * entry and exit events
     */
    private byte correlationId = (byte) 0;

    /**
     * Creates a new Transit applet instance.
     *
     * @param bArray
     *            The array containing installation parameters
     * @param bOffset
     *            The starting offset in bArray
     * @param bLength
     *            The length in bytes of the parameter data in bArray
     */
    protected TransitApplet(byte[] bArray, short bOffset, byte bLength) {

        // Create static DES key
        staticKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
                KeyBuilder.LENGTH_DES, false);

        // Create cipher
        cipher = Cipher.getInstance(Cipher.ALG_DES_CBC_ISO9797_M2, false);
```

```
        // Create card challenge transient buffer
        cardChallenge = JCSystem.makeTransientByteArray(CHALLENGE_LENGTH,
                JCSystem.CLEAR_ON_DESELECT);

        // Create key derivation data transient buffer
        keyDerivationData = JCSystem.makeTransientByteArray(
                (short) (2 * CHALLENGE_LENGTH), JCSystem.CLEAR_ON_DESELECT);

        // Create session key data transient buffer
        sessionKeyData = JCSystem.makeTransientByteArray(
            (short) (2 * keyDerivationData.length),
            JCSystem.CLEAR_ON_DESELECT);
        // XXX: Allocates more than actual key to contain the complete
        // encrypted key derivation data

        // Create signature
        signature = Signature.getInstance(Signature.ALG_DES_MAC8_ISO9797_M2,
                false);

        byte aidLen = bArray[bOffset]; // aid length
        if (aidLen == (byte) 0) {
            register();
        } else {
            register(bArray, (short) (bOffset + 1), aidLen);
        }

        // Ignore control info
        bOffset = (short) (bOffset + aidLen + 1);
        byte infoLen = bArray[bOffset]; // control info length
        bOffset = (short) (bOffset + infoLen + 1);

        byte paramLen = bArray[bOffset++]; // applet parameters length

        // Retrieve UID, static key data and the PIN initialization values from
        // installation parameters

        if (paramLen <= (LENGTH_DES_BYTE + UID_LENGTH)
                || paramLen > (LENGTH_DES_BYTE + UID_LENGTH + MAX_PIN_SIZE)) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Retrieve the UID
        uid = new byte[UID_LENGTH];
        Util.arrayCopy(bArray, bOffset, uid, (short) 0, UID_LENGTH);
        bOffset += UID_LENGTH;

        // Retrieve the static key data
        staticKey.setKey(fixParity(bArray, bOffset, LENGTH_DES_BYTE), bOffset);
        bOffset += LENGTH_DES_BYTE;

        // Retrieve the flag indicating whether or not to use a transient key
        useTransientKey = (bArray[bOffset] != (byte) 0);
        bOffset++;

        // Retrieve the PIN
        pin = new OwnerPIN(MAX_PIN_TRIES, MAX_PIN_SIZE);
        pin.update(bArray, bOffset,
                (byte) (paramLen - UID_LENGTH - LENGTH_DES_BYTE - 1));

        // Create transient DES session key
        if (useTransientKey) {
            sessionKey = (DESKey) KeyBuilder.buildKey(
                KeyBuilder.TYPE_DES_TRANSIENT_DESELECT, KeyBuilder.LENGTH_DES,
                false);
```

```
        } else {
            sessionKey = (DESKey) KeyBuilder.buildKey(
                KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES,
                false);
        }

        // Create and initialize the ramdom data generator with the UID (seed)
        random = RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
        random.setSeed(uid, (short) 0, UID_LENGTH);

        // Initialize the cipher with the static key
        cipher.init(staticKey, Cipher.MODE_ENCRYPT);

    }

    public static void install(byte[] bArray, short bOffset, byte bLength) {
        // Create a Transit applet instance
        new TransitApplet(bArray, bOffset, bLength);
    }

    public boolean select() {
        // The applet declines to be selected
        // if the PIN is blocked.
        if (pin.getTriesRemaining() == 0) {
            return false;
        }
        return true;
    }

    public void deselect() {
        // Reset the PIN value
        pin.reset();
        if (!useTransientKey) {
            sessionKey.clearKey();
        }
    }

    public void process(APDU apdu) {

        // C-APDU: [CLA, INS, P1, P2, LC, ...]

        byte[] buffer = apdu.getBuffer();

        // Dispatch C-APDU for processing
        if (!apdu.isISOInterindustryCLA()) {
            switch (buffer[ISO7816.OFFSET_INS]) {
            case INITIALIZE_SESSION:
                initializeSession(apdu);
                return;
            case PROCESS_REQUEST:
                processRequest(apdu);
                return;
            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
            }
        } else {
            if (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)) {
                return;
            } else if (buffer[ISO7816.OFFSET_INS] == VERIFY) {
                verify(apdu);
            } else {
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
            }
        }
    }
```

```java
    /**
     * Initializes a CAD/card interaction session. This is the first step of
     * mutual authentication. A new card challenge is generated and used along
     * with the passed-in host challenge to generate the derivation data from
     * which a new session key is derived. The card challenge is appended to the
     * response message. The response message is signed using the newly
     * generated session key then sent back. Note that mutual authentication is
     * subsequently completed upon succesful verification of the signature of
     * the first request received.
     *
     * @param apdu
     *            The APDU
     */
    private void initializeSession(APDU apdu) {

        // C-APDU: [CLA, INS, P1, P2, LC, [4-bytes Host Challenge]]

        byte[] buffer = apdu.getBuffer();

        if ((buffer[ISO7816.OFFSET_P1] != 0)
                || (buffer[ISO7816.OFFSET_P2] != 0)) {
            ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
        }

        byte numBytes = buffer[ISO7816.OFFSET_LC];

        byte count = (byte) apdu.setIncomingAndReceive();

        if (numBytes != CHALLENGE_LENGTH || count != CHALLENGE_LENGTH) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Generate card challenge
        generateCardChallenge();

        // Generate key derivation data from host challenge and card challenge
        generateKeyDerivationData(buffer);

        // Generate session key from derivation data
        generateSessionKey();

        // R-APDU: [[4-bytes Card Challenge], [2-bytes Status Word], [8-bytes
        // MAC]]

        short offset = 0;

        // Append card challenge to response message
        offset = Util.arrayCopyNonAtomic(cardChallenge, (short) 0, buffer,
                offset, CHALLENGE_LENGTH);

        // Append status word to response message
        offset = Util.setShort(buffer, offset, SW_SUCCESS);

        // Sign response message and append MAC to response message
        offset = generateMAC(buffer, offset);

        // Send R-APDU
        apdu.setOutgoingAndSend((short) 0, offset);
    }

    /**
     * Processes an incoming request. The request message signature is verified,
     * then it is dispatched to the relevant handling method. The response
     * message is then signed and sent back.
```

```
     *
     * @param apdu
     *            The APDU
     */
    private void processRequest(APDU apdu) {

        // C-APDU: [CLA, INS, P1, P2, LC, [Request Message], [8-bytes MAC]]
        // Request Message: [T, L, [V...]]

        byte[] buffer = apdu.getBuffer();

        if ((buffer[ISO7816.OFFSET_P1] != 0)
                || (buffer[ISO7816.OFFSET_P2] != 0)) {
            ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
        }

        byte numBytes = buffer[ISO7816.OFFSET_LC];

        byte count = (byte) apdu.setIncomingAndReceive();

        if (numBytes != count) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Check request message signature
        if (!checkMAC(buffer)) {
            ISOException.throwIt(SW_WRONG_SIGNATURE);
        }

        if ((numBytes - MAC_LENGTH) != (buffer[TLV_LENGTH_OFFSET] + 2)) {
            ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }

        // R-APDU: [[Response Message], [2-bytes Status Word], [8-bytes MAC]]

        short offset = 0;

        // Dispatch request message for processing
        switch (buffer[TLV_TAG_OFFSET]) {
        case PROCESS_ENTRY:
            offset = processEntry(buffer, TLV_VALUE_OFFSET,
                    buffer[TLV_LENGTH_OFFSET]);
            break;
        case PROCESS_EXIT:
            offset = processExit(buffer, TLV_VALUE_OFFSET,
                    buffer[TLV_LENGTH_OFFSET]);
            break;
        case CREDIT:
            offset = credit(buffer, TLV_VALUE_OFFSET, buffer[TLV_LENGTH_OFFSET]);
            break;
        case GET_BALANCE:
            offset = getBalance(buffer, TLV_VALUE_OFFSET,
                    buffer[TLV_LENGTH_OFFSET]);
            break;
        default:
            ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
        }

        // Append status word to response message
        offset = Util.setShort(buffer, offset, SW_SUCCESS);

        // Sign response message and append MAC to response message
        offset = generateMAC(buffer, offset);

        // Send R-APDU
```

```
        apdu.setOutgoingAndSend((short) 0, offset);
    }

    /**
     * Verifies the PIN.
     *
     * @param apdu
     *            The APDU
     */
    private void verify(APDU apdu) {

        byte[] buffer = apdu.getBuffer();

        byte numBytes = buffer[ISO7816.OFFSET_LC];

        byte count = (byte) apdu.setIncomingAndReceive();

        if (numBytes != count) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Verify PIN
        if (pin.check(buffer, ISO7816.OFFSET_CDATA, numBytes) == false) {
            ISOException.throwIt(SW_VERIFICATION_FAILED);
        }
    }

    /**
     * Generates a new random card challenge.
     *
     */
    private void generateCardChallenge() {
        // Generate random card challenge
        random.generateData(cardChallenge, (short) 0, CHALLENGE_LENGTH);
    }

    /**
     * Generates the session key derivation data from the passed-in host
     * challenge and the card challenge.
     *
     * @param buffer
     *            The APDU buffer
     */
    private void generateKeyDerivationData(byte[] buffer) {
        byte numBytes = buffer[ISO7816.OFFSET_LC];

        if (numBytes < CHALLENGE_LENGTH) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Derivation data: [[8-bytes host challenge], [8-bytes card challenge]]

        // Append host challenge (from buffer) to derivation data
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, keyDerivationData,
                (short) 0, CHALLENGE_LENGTH);
        // Append card challenge to derivation data
        Util.arrayCopy(cardChallenge, (short) 0, keyDerivationData,
                CHALLENGE_LENGTH, CHALLENGE_LENGTH);
    }

    /**
     * Generates a new DES session key from the derivation data.
     *
     */
    private void generateSessionKey() {
```

```java
        cipher.doFinal(keyDerivationData, (short) 0, (short) keyDerivationData.length,
                sessionKeyData, (short) 0);
        // Generate new session key from encrypted derivation data
        sessionKey.setKey(fixParity(sessionKeyData, (short) 0, (short) sessionKeyData.length /*LENGTH_DES_BYTE*/), (short) 0);
    }

    /**
     * Checks the request message signature.
     *
     * @param buffer
     *            The APDU buffer
     * @return true if the message signature is correct; false otherwise
     */
    private boolean checkMAC(byte[] buffer) {
        byte numBytes = buffer[ISO7816.OFFSET_LC];

        if (numBytes <= MAC_LENGTH) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Initialize signature with current session key for verification
        signature.init(sessionKey, Signature.MODE_VERIFY);
        // Verify request message signature
        return signature.verify(buffer, ISO7816.OFFSET_CDATA,
                (short) (numBytes - MAC_LENGTH), buffer,
                (short) (ISO7816.OFFSET_CDATA + numBytes - MAC_LENGTH),
                MAC_LENGTH);
    }

    /**
     * Generates the response message MAC: generates the MAC and appends the MAC
     * to the response message.
     *
     * @param buffer
     *            The APDU buffer
     * @param offset
     *            The offset of the MAC in the buffer
     * @return The resulting length of the response message
     */
    private short generateMAC(byte[] buffer, short offset) {
        // Initialize signature with current session key for signing
        signature.init(sessionKey, Signature.MODE_SIGN);
        // Sign response message and append the MAC to the response message
        short sigLength = signature.sign(buffer, (short) 0, offset, buffer,
                offset);
        return (short) (offset + sigLength);
    }

    /**
     * Processes a transit entry event. The passed-in entry station ID is
     * recorded and the correlation ID is incremented. The UID and the
     * correlation ID are returned in the response message.
     *
     * Request Message: [2-bytes Entry Station ID]
     *
     * Response Message: [[2-bytes UID], [2-bytes Correlation ID]]
     *
     * @param buffer
     *            The APDU buffer
     * @param messageOffset
     *            The offset of the request message content in the APDU buffer
     * @param messageLength
     *            The length of the request message content.
```

```
     * @return The offset at which content can be appended to the response
     *         message
     */
    private short processEntry(byte[] buffer, short messageOffset,
            short messageLength) {

        // Request Message: [2-bytes Entry Station ID]

        if (messageLength != 2) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Check minimum balance
        if (balance < MIN_TRANSIT_BALANCE) {
            ISOException.throwIt(SW_MIN_TRANSIT_BALANCE);
        }

        // Check consistent transit state: should not currently be in transit
        if (entryStationId >= 0) {
            ISOException.throwIt(SW_INVALID_TRANSIT_STATE);
        }

        JCSystem.beginTransaction();

        // Get/assign entry station ID from request message
        entryStationId = Util.getShort(buffer, messageOffset);

        // Increment correlation ID
        correlationId++;

        JCSystem.commitTransaction();

        // Response Message: [[8-bytes UID], [2-bytes Correlation ID]]

        short offset = 0;

        // Append UID to response message
        offset = Util.arrayCopy(uid, (short) 0, buffer, offset, UID_LENGTH);

        // Append correlation ID to response message
        offset = Util.setShort(buffer, offset, correlationId);

        return offset;
    }

    /**
     * Processes a transit exit event. The passed-in transit fee is debited from
     * the account. The UID and the correlation ID are returned in the response
     * message.
     *
     * Request Message: [1-byte Transit Fee]
     *
     * Response Message: [[2-bytes UID], [2-bytes Correlation ID]]
     *
     * @param buffer
     *            The APDU buffer
     * @param messageOffset
     *            The offset of the request message content in the APDU buffer
     * @param messageLength
     *            The length of the request message content.
     * @return The offset at which content can be appended to the response
     *         message
     */
    private short processExit(byte[] buffer, short messageOffset,
            short messageLength) {
```

```
        // Request Message: [1-byte Transit Fee]

        if (messageLength != 1) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Check minimum balance
        if (balance < MIN_TRANSIT_BALANCE) {
            ISOException.throwIt(SW_MIN_TRANSIT_BALANCE);
        }

        // Check consistent transit state: should be currently in transit
        if (entryStationId < 0) {
            ISOException.throwIt(SW_INVALID_TRANSIT_STATE);
        }

        // Get transit fee from request message
        byte transitFee = buffer[messageOffset];

        // Check potential negative balance
        if (balance < transitFee) {
            ISOException.throwIt(SW_NEGATIVE_BALANCE);
        }

        JCSystem.beginTransaction();

        // Debit transit fee
        balance -= transitFee;

        // Reset entry station ID
        entryStationId = -1;

        JCSystem.commitTransaction();

        // Response Message: [[8-bytes UID], [2-bytes Correlation ID]]

        short offset = 0;

        // Append UID to response message
        offset = Util.arrayCopy(uid, (short) 0, buffer, offset, UID_LENGTH);

        // Append correlation ID to response message
        offset = Util.setShort(buffer, offset, correlationId);

        return offset;
    }

    /**
     * Credits the account of the passed-in amount.
     *
     * Request Message: [1-byte Credit Amount]
     *
     * Response Message: []
     *
     * @param buffer
     *            The APDU buffer
     * @param messageOffset
     *            The offset of the request message content in the APDU buffer
     * @param messageLength
     *            The length of the request message content.
     * @return The offset at which content can be appended to the response
     *         message
     */
    private short credit(byte[] buffer, short messageOffset, short messageLength) {
```

```
        // Check access authorization
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
        }

        // Request Message: [1-byte Credit Amount]

        if (messageLength != 1) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Get credit amount from request message
        byte creditAmount = buffer[messageOffset];

        // Check credit amount
        if ((creditAmount > MAX_CREDIT_AMOUNT) || (creditAmount < 0)) {
            ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
        }

        // Check the new balance
        if ((short) (balance + creditAmount) > MAX_BALANCE) {
            ISOException.throwIt(SW_EXCEED_MAXIMUM_BALANCE);
        }

        // Credit the amount
        balance += creditAmount;

        // Response Message: []

        return 0;
    }

    /**
     * Gets/returns the balance.
     *
     * Request Message: []
     *
     * Response Message: [2-bytes Balance]
     *
     * @param buffer
     *            The APDU buffer
     * @param messageOffset
     *            The offset of the request message content in the APDU buffer
     * @param messageLength
     *            The length of the request message content.
     * @return The offset at which content can be appended to the response
     *         message
     */
    private short getBalance(byte[] buffer, short messageOffset,
            short messageLength) {

        // Check access authorization
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
        }

        // Request Message: []

        if (messageLength != 0) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Response Message: [2-bytes Balance]
```

```java
        short offset = 0;

        // Append balance to response message
        offset = Util.setShort(buffer, offset, balance);

        return offset;
    }

    /**
     * Fixes the parity on DES key data.
     *
     * @param buffer
     *            The buffer containing the DES key data
     * @param offset
     *            The offset of the DES key data in the buffer
     * @param length
     *            The length of the DES key data
     * @return The passed-in buffer with the DES key data parity fixed
     */
    private byte[] fixParity(byte[] buffer, short offset, short length) {
        for (byte i = 0; i < length; i++) {
            short parity = 0;
            buffer[(short) (offset + i)] &= 0xFE;
            for (byte j = 1; j < 8; j++) {
                if ((buffer[(short) (offset + i)] & (byte) (1 << j)) != 0) {
                    parity++;
                }
            }
            if ((parity % 2) == 0) {
                buffer[(short) (offset + i)] |= 1;
            }
        }
        return buffer;
    }
}
```

Listing A.1: Source code of the TransitApplet.

```
Compiled from "TransitApplet.java"
public class com.sun.javacard.samples.transit.TransitApplet extends javacard.
    framework.Applet {
  static final byte VERIFY;

  static final byte INITIALIZE_SESSION;

  static final byte PROCESS_REQUEST;

  static final byte PROCESS_ENTRY;

  static final byte PROCESS_EXIT;

  static final byte CREDIT;

  static final byte GET_BALANCE;

  static final short TLV_TAG_OFFSET;

  static final short TLV_LENGTH_OFFSET;

  static final short TLV_VALUE_OFFSET;

  static final short MAX_BALANCE;

```

```
static final short MIN_TRANSIT_BALANCE;

static final short MAX_CREDIT_AMOUNT;

static final byte MAX_PIN_TRIES;

static final byte MAX_PIN_SIZE;

static final short SW_VERIFICATION_FAILED;

static final short SW_PIN_VERIFICATION_REQUIRED;

static final short SW_INVALID_TRANSACTION_AMOUNT;

static final short SW_EXCEED_MAXIMUM_BALANCE;

static final short SW_NEGATIVE_BALANCE;

static final short SW_WRONG_SIGNATURE;

static final short SW_MIN_TRANSIT_BALANCE;

static final short SW_INVALID_TRANSIT_STATE;

static final short SW_SUCCESS;

static final short UID_LENGTH;

static final short LENGTH_DES_BYTE;

static final short CHALLENGE_LENGTH;

static final short MAC_LENGTH;

private byte[] uid;

private javacardx.crypto.Cipher cipher;

private javacard.security.DESKey staticKey;

private byte[] cardChallenge;

private byte[] keyDerivationData;

private byte[] sessionKeyData;

private javacard.security.DESKey sessionKey;

private boolean useTransientKey;

private javacard.security.Signature signature;

private javacard.security.RandomData random;

private javacard.framework.OwnerPIN pin;

private short balance;

private short entryStationId;

private byte correlationId;

protected com.sun.javacard.samples.transit.TransitApplet(byte[], short, byte);
  Code:
     0: aload_0
```

```
         1: invokespecial #1                  // Method javacard/framework/Applet."<init>":()V
         4: aload_0
         5: iconst_1
         6: putfield      #2                  // Field useTransientKey:Z
         9: aload_0
        10: iconst_0
        11: putfield      #3                  // Field balance:S
        14: aload_0
        15: iconst_m1
        16: putfield      #4                  // Field entryStationId:S
        19: aload_0
        20: iconst_0
        21: putfield      #5                  // Field correlationId:B
        24: aload_0
        25: iconst_3
        26: bipush        64
        28: iconst_0
        29: invokestatic  #6                  // Method javacard/security/KeyBuilder.buildKey:(BSZ)Ljavacard/security/Key;
        32: checkcast     #7                  // class javacard/security/DESKey
        35: putfield      #8                  // Field staticKey:Ljavacard/security/DESKey;
        38: aload_0
        39: iconst_3
        40: iconst_0
        41: invokestatic  #9                  // Method javacardx/crypto/Cipher.getInstance:(BZ)Ljavacardx/crypto/Cipher;
        44: putfield      #10                 // Field cipher:Ljavacardx/crypto/Cipher;
        47: aload_0
        48: iconst_4
        49: iconst_2
        50: invokestatic  #11                 // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
        53: putfield      #12                 // Field cardChallenge:[B
        56: aload_0
        57: bipush        8
        59: iconst_2
        60: invokestatic  #11                 // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
        63: putfield      #13                 // Field keyDerivationData:[B
        66: aload_0
        67: iconst_2
        68: aload_0
        69: getfield      #13                 // Field keyDerivationData:[B
        72: arraylength
        73: imul
        74: i2s
        75: iconst_2
        76: invokestatic  #11                 // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
        79: putfield      #14                 // Field sessionKeyData:[B
        82: aload_0
        83: bipush        6
        85: iconst_0
        86: invokestatic  #15                 // Method javacard/security/Signature.getInstance:(BZ)Ljavacard/security/Signature;
        89: putfield      #16                 // Field signature:Ljavacard/security/Signature;
        92: aload_1
        93: iload_2
        94: baload
        95: istore        4
        97: iload         4
        99: ifne          109
```

```
102: aload_0
103: invokevirtual #17                  // Method register:()V
106: goto          120
109: aload_0
110: aload_1
111: iload_2
112: iconst_1
113: iadd
114: i2s
115: iload         4
117: invokevirtual #18                  // Method register:([BSB)V
120: iload_2
121: iload         4
123: iadd
124: iconst_1
125: iadd
126: i2s
127: istore_2
128: aload_1
129: iload_2
130: baload
131: istore        5
133: iload_2
134: iload         5
136: iadd
137: iconst_1
138: iadd
139: i2s
140: istore_2
141: aload_1
142: iload_2
143: iload_2
144: iconst_1
145: iadd
146: i2s
147: istore_2
148: baload
149: istore        6
151: iload         6
153: bipush        16
155: if_icmple     165
158: iload         6
160: bipush        24
162: if_icmple     171
165: sipush        26368
168: invokestatic  #19                  // Method javacard/framework/ISOException
    .throwIt:(S)V
171: aload_0
172: bipush        8
174: newarray       byte
176: putfield      #20                  // Field uid:[B
179: aload_1
180: iload_2
181: aload_0
182: getfield      #20                  // Field uid:[B
185: iconst_0
186: bipush        8
188: invokestatic  #21                  // Method javacard/framework/Util.
    arrayCopy:([BS[BSS)S
191: pop
192: iload_2
193: bipush        8
195: iadd
196: i2s
197: istore_2
```

```
    198: aload_0
    199: getfield      #8                  // Field staticKey:Ljavacard/security/DESKey;
    202: aload_0
    203: aload_1
    204: iload_2
    205: bipush        8
    207: invokespecial #22                 // Method fixParity:([BSS)[B
    210: iload_2
    211: invokeinterface #23,  3           // InterfaceMethod javacard/security/DESKey.setKey:([BS)V
    216: iload_2
    217: bipush        8
    219: iadd
    220: i2s
    221: istore_2
    222: aload_0
    223: aload_1
    224: iload_2
    225: baload
    226: ifeq          233
    229: iconst_1
    230: goto          234
    233: iconst_0
    234: putfield      #2                  // Field useTransientKey:Z
    237: iload_2
    238: iconst_1
    239: iadd
    240: i2s
    241: istore_2
    242: aload_0
    243: new           #24                 // class javacard/framework/OwnerPIN
    246: dup
    247: iconst_3
    248: bipush        8
    250: invokespecial #25                 // Method javacard/framework/OwnerPIN."<init>":(BB)V
    253: putfield      #26                 // Field pin:Ljavacard/framework/OwnerPIN;
    256: aload_0
    257: getfield      #26                 // Field pin:Ljavacard/framework/OwnerPIN;
    260: aload_1
    261: iload_2
    262: iload         6
    264: bipush        8
    266: isub
    267: bipush        8
    269: isub
    270: iconst_1
    271: isub
    272: i2b
    273: invokevirtual #27                 // Method javacard/framework/OwnerPIN.update:([BSB)V
    276: aload_0
    277: getfield      #2                  // Field useTransientKey:Z
    280: ifeq          300
    283: aload_0
    284: iconst_2
    285: bipush        64
    287: iconst_0
    288: invokestatic  #6                  // Method javacard/security/KeyBuilder.buildKey:(BSZ)Ljavacard/security/Key;
    291: checkcast     #7                  // class javacard/security/DESKey
```

```
     294: putfield      #28                 // Field sessionKey:Ljavacard/security/
         DESKey;
     297: goto          314
     300: aload_0
     301: iconst_3
     302: bipush        64
     304: iconst_0
     305: invokestatic  #6                  // Method javacard/security/KeyBuilder.
         buildKey:(BSZ)Ljavacard/security/Key;
     308: checkcast     #7                  // class javacard/security/DESKey
     311: putfield      #28                 // Field sessionKey:Ljavacard/security/
         DESKey;
     314: aload_0
     315: iconst_1
     316: invokestatic  #29                 // Method javacard/security/RandomData.
         getInstance:(B)Ljavacard/security/RandomData;
     319: putfield      #30                 // Field random:Ljavacard/security/
         RandomData;
     322: aload_0
     323: getfield      #30                 // Field random:Ljavacard/security/
         RandomData;
     326: aload_0
     327: getfield      #20                 // Field uid:[B
     330: iconst_0
     331: bipush        8
     333: invokevirtual #31                 // Method javacard/security/RandomData.
         setSeed:([BSS)V
     336: aload_0
     337: getfield      #10                 // Field cipher:Ljavacardx/crypto/Cipher;
     340: aload_0
     341: getfield      #8                  // Field staticKey:Ljavacard/security/
         DESKey;
     344: iconst_2
     345: invokevirtual #32                 // Method javacardx/crypto/Cipher.init:(
         Ljavacard/security/Key;B)V
     348: return

  public static void install(byte[], short, byte);
    Code:
       0: new           #33                 // class com/sun/javacard/samples/transit
           /TransitApplet
       3: dup
       4: aload_0
       5: iload_1
       6: iload_2
       7: invokespecial #34                 // Method "<init>":([BSB)V
      10: pop
      11: return

  public boolean select();
    Code:
       0: aload_0
       1: getfield      #26                 // Field pin:Ljavacard/framework/
           OwnerPIN;
       4: invokevirtual #35                 // Method javacard/framework/OwnerPIN.
           getTriesRemaining:()B
       7: ifne          12
      10: iconst_0
      11: ireturn
      12: iconst_1
      13: ireturn

  public void deselect();
    Code:
       0: aload_0
```

```
       1: getfield      #26                 // Field pin:Ljavacard/framework/
           OwnerPIN;
       4: invokevirtual #36                 // Method javacard/framework/OwnerPIN.
           reset:()V
       7: aload_0
       8: getfield      #2                  // Field useTransientKey:Z
      11: ifne          23
      14: aload_0
      15: getfield      #28                 // Field sessionKey:Ljavacard/security/
          DESKey;
      18: invokeinterface #37,  1           // InterfaceMethod javacard/security/
          DESKey.clearKey:()V
      23: return

  public void process(javacard.framework.APDU);
    Code:
       0: aload_1
       1: invokevirtual #38                 // Method javacard/framework/APDU.
           getBuffer:()[B
       4: astore_2
       5: aload_1
       6: invokevirtual #39                 // Method javacard/framework/APDU.
           isISOInterindustryCLA:()Z
       9: ifne          61
      12: aload_2
      13: iconst_1
      14: baload
      15: lookupswitch  { // 2

                    48: 40

                    64: 46
               default: 52
          }
      40: aload_0
      41: aload_1
      42: invokespecial #40                 // Method initializeSession:(Ljavacard/
          framework/APDU;)V
      45: return
      46: aload_0
      47: aload_1
      48: invokespecial #41                 // Method processRequest:(Ljavacard/
          framework/APDU;)V
      51: return
      52: sipush        27904
      55: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      58: goto          92
      61: aload_2
      62: iconst_1
      63: baload
      64: bipush        -92
      66: if_icmpne     70
      69: return
      70: aload_2
      71: iconst_1
      72: baload
      73: bipush        32
      75: if_icmpne     86
      78: aload_0
      79: aload_1
      80: invokespecial #42                 // Method verify:(Ljavacard/framework/
          APDU;)V
      83: goto          92
      86: sipush        27904
```

```
      89: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      92: return

  private void initializeSession(javacard.framework.APDU);
    Code:
       0: aload_1
       1: invokevirtual #38                 // Method javacard/framework/APDU.
           getBuffer:()[B
       4: astore_2
       5: aload_2
       6: iconst_2
       7: baload
       8: ifne          17
      11: aload_2
      12: iconst_3
      13: baload
      14: ifeq          23
      17: sipush        27270
      20: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      23: aload_2
      24: iconst_4
      25: baload
      26: istore_3
      27: aload_1
      28: invokevirtual #43                 // Method javacard/framework/APDU.
          setIncomingAndReceive:()S
      31: i2b
      32: istore        4
      34: iload_3
      35: iconst_4
      36: if_icmpne     45
      39: iload         4
      41: iconst_4
      42: if_icmpeq     51
      45: sipush        26368
      48: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      51: aload_0
      52: invokespecial #44                 // Method generateCardChallenge:()V
      55: aload_0
      56: aload_2
      57: invokespecial #45                 // Method generateKeyDerivationData:([B)V
      60: aload_0
      61: invokespecial #46                 // Method generateSessionKey:()V
      64: iconst_0
      65: istore        5
      67: aload_0
      68: getfield      #12                 // Field cardChallenge:[B
      71: iconst_0
      72: aload_2
      73: iload         5
      75: iconst_4
      76: invokestatic  #47                 // Method javacard/framework/Util.
          arrayCopyNonAtomic:([BS[BSS)S
      79: istore        5
      81: aload_2
      82: iload         5
      84: sipush        -28672
      87: invokestatic  #48                 // Method javacard/framework/Util.
          setShort:([BSS)S
      90: istore        5
      92: aload_0
      93: aload_2
```

```
      94: iload         5
      96: invokespecial #49                 // Method generateMAC:([BS)S
      99: istore        5
     101: aload_1
     102: iconst_0
     103: iload         5
     105: invokevirtual #50                 // Method javacard/framework/APDU.
         setOutgoingAndSend:(SS)V
     108: return

  private void processRequest(javacard.framework.APDU);
    Code:
       0: aload_1
       1: invokevirtual #38                 // Method javacard/framework/APDU.
           getBuffer:()[B
       4: astore_2
       5: aload_2
       6: iconst_2
       7: baload
       8: ifne          17
      11: aload_2
      12: iconst_3
      13: baload
      14: ifeq          23
      17: sipush        27270
      20: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      23: aload_2
      24: iconst_4
      25: baload
      26: istore_3
      27: aload_1
      28: invokevirtual #43                 // Method javacard/framework/APDU.
          setIncomingAndReceive:()S
      31: i2b
      32: istore        4
      34: iload_3
      35: iload         4
      37: if_icmpeq     46
      40: sipush        26368
      43: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      46: aload_0
      47: aload_2
      48: invokespecial #51                 // Method checkMAC:([B)Z
      51: ifne          60
      54: sipush        -28411
      57: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      60: iload_3
      61: bipush        8
      63: isub
      64: aload_2
      65: bipush        6
      67: baload
      68: iconst_2
      69: iadd
      70: if_icmpeq     79
      73: sipush        27264
      76: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      79: iconst_0
      80: istore        5
      82: aload_2
      83: iconst_5
```

```
      84: baload
      85: tableswitch   { // -63 to -60

                   -63: 116

                   -62: 133

                   -61: 150

                   -60: 167
               default: 184
          }
     116: aload_0
     117: aload_2
     118: bipush        7
     120: aload_2
     121: bipush        6
     123: baload
     124: i2s
     125: invokespecial #52                 // Method processEntry:([BSS)S
     128: istore        5
     130: goto          190
     133: aload_0
     134: aload_2
     135: bipush        7
     137: aload_2
     138: bipush        6
     140: baload
     141: i2s
     142: invokespecial #53                 // Method processExit:([BSS)S
     145: istore        5
     147: goto          190
     150: aload_0
     151: aload_2
     152: bipush        7
     154: aload_2
     155: bipush        6
     157: baload
     158: i2s
     159: invokespecial #54                 // Method credit:([BSS)S
     162: istore        5
     164: goto          190
     167: aload_0
     168: aload_2
     169: bipush        7
     171: aload_2
     172: bipush        6
     174: baload
     175: i2s
     176: invokespecial #55                 // Method getBalance:([BSS)S
     179: istore        5
     181: goto          190
     184: sipush        27265
     187: invokestatic  #19                 // Method javacard/framework/ISOException
         .throwIt:(S)V
     190: aload_2
     191: iload         5
     193: sipush        -28672
     196: invokestatic  #48                 // Method javacard/framework/Util.
         setShort:([BSS)S
     199: istore        5
     201: aload_0
     202: aload_2
     203: iload         5
     205: invokespecial #49                 // Method generateMAC:([BS)S
```

```
    208: istore        5
    210: aload_1
    211: iconst_0
    212: iload         5
    214: invokevirtual #50                 // Method javacard/framework/APDU.
        setOutgoingAndSend:(SS)V
    217: return

  private void verify(javacard.framework.APDU);
    Code:
       0: aload_1
       1: invokevirtual #38                 // Method javacard/framework/APDU.
          getBuffer:()[B
       4: astore_2
       5: aload_2
       6: iconst_4
       7: baload
       8: istore_3
       9: aload_1
      10: invokevirtual #43                 // Method javacard/framework/APDU.
         setIncomingAndReceive:()S
      13: i2b
      14: istore        4
      16: iload_3
      17: iload         4
      19: if_icmpeq     28
      22: sipush        26368
      25: invokestatic  #19                 // Method javacard/framework/ISOException
         .throwIt:(S)V
      28: aload_0
      29: getfield      #26                 // Field pin:Ljavacard/framework/
         OwnerPIN;
      32: aload_2
      33: iconst_5
      34: iload_3
      35: invokevirtual #56                 // Method javacard/framework/OwnerPIN.
         check:([BSB)Z
      38: ifne          47
      41: sipush        25344
      44: invokestatic  #19                 // Method javacard/framework/ISOException
         .throwIt:(S)V
      47: return

  private void generateCardChallenge();
    Code:
       0: aload_0
       1: getfield      #30                 // Field random:Ljavacard/security/
          RandomData;
       4: aload_0
       5: getfield      #12                 // Field cardChallenge:[B
       8: iconst_0
       9: iconst_4
      10: invokevirtual #57                 // Method javacard/security/RandomData.
         generateData:([BSS)V
      13: return

  private void generateKeyDerivationData(byte[]);
    Code:
       0: aload_1
       1: iconst_4
       2: baload
       3: istore_2
       4: iload_2
       5: iconst_4
       6: if_icmpge     15
```

```
       9: sipush        26368
      12: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      15: aload_1
      16: iconst_5
      17: aload_0
      18: getfield      #13                 // Field keyDerivationData:[B
      21: iconst_0
      22: iconst_4
      23: invokestatic  #21                 // Method javacard/framework/Util.
          arrayCopy:([BS[BSS)S
      26: pop
      27: aload_0
      28: getfield      #12                 // Field cardChallenge:[B
      31: iconst_0
      32: aload_0
      33: getfield      #13                 // Field keyDerivationData:[B
      36: iconst_4
      37: iconst_4
      38: invokestatic  #21                 // Method javacard/framework/Util.
          arrayCopy:([BS[BSS)S
      41: pop
      42: return

  private void generateSessionKey();
    Code:
       0: aload_0
       1: getfield      #10                 // Field cipher:Ljavacardx/crypto/Cipher;
       4: aload_0
       5: getfield      #13                 // Field keyDerivationData:[B
       8: iconst_0
       9: aload_0
      10: getfield      #13                 // Field keyDerivationData:[B
      13: arraylength
      14: i2s
      15: aload_0
      16: getfield      #14                 // Field sessionKeyData:[B
      19: iconst_0
      20: invokevirtual #58                 // Method javacardx/crypto/Cipher.doFinal
          :([BSS[BS)S
      23: pop
      24: aload_0
      25: getfield      #28                 // Field sessionKey:Ljavacard/security/
          DESKey;
      28: aload_0
      29: aload_0
      30: getfield      #14                 // Field sessionKeyData:[B
      33: iconst_0
      34: aload_0
      35: getfield      #14                 // Field sessionKeyData:[B
      38: arraylength
      39: i2s
      40: invokespecial #22                 // Method fixParity:([BSS)[B
      43: iconst_0
      44: invokeinterface #23,  3           // InterfaceMethod javacard/security/
          DESKey.setKey:([BS)V
      49: return

  private boolean checkMAC(byte[]);
    Code:
       0: aload_1
       1: iconst_4
       2: baload
       3: istore_2
       4: iload_2
```

```
       5: bipush        8
       7: if_icmpgt     16
      10: sipush        26368
      13: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      16: aload_0
      17: getfield      #16                 // Field signature:Ljavacard/security/
          Signature;
      20: aload_0
      21: getfield      #28                 // Field sessionKey:Ljavacard/security/
          DESKey;
      24: iconst_2
      25: invokevirtual #59                 // Method javacard/security/Signature.
          init:(Ljavacard/security/Key;B)V
      28: aload_0
      29: getfield      #16                 // Field signature:Ljavacard/security/
          Signature;
      32: aload_1
      33: iconst_5
      34: iload_2
      35: bipush        8
      37: isub
      38: i2s
      39: aload_1
      40: iconst_5
      41: iload_2
      42: iadd
      43: bipush        8
      45: isub
      46: i2s
      47: bipush        8
      49: invokevirtual #60                 // Method javacard/security/Signature.
          verify:([BSS[BSS)Z
      52: ireturn

  private short generateMAC(byte[], short);
    Code:
       0: aload_0
       1: getfield      #16                 // Field signature:Ljavacard/security/
           Signature;
       4: aload_0
       5: getfield      #28                 // Field sessionKey:Ljavacard/security/
           DESKey;
       8: iconst_1
       9: invokevirtual #59                 // Method javacard/security/Signature.
           init:(Ljavacard/security/Key;B)V
      12: aload_0
      13: getfield      #16                 // Field signature:Ljavacard/security/
          Signature;
      16: aload_1
      17: iconst_0
      18: iload_2
      19: aload_1
      20: iload_2
      21: invokevirtual #61                 // Method javacard/security/Signature.
          sign:([BSS[BS)S
      24: istore_3
      25: iload_2
      26: iload_3
      27: iadd
      28: i2s
      29: ireturn

  private short processEntry(byte[], short, short);
    Code:
```

```
        0: iload_3
        1: iconst_2
        2: if_icmpeq     11
        5: sipush        26368
        8: invokestatic  #19                // Method javacard/framework/ISOException
         .throwIt:(S)V
       11: aload_0
       12: getfield      #3                 // Field balance:S
       15: bipush        10
       17: if_icmpge     26
       20: sipush        -28410
       23: invokestatic  #19                // Method javacard/framework/ISOException
         .throwIt:(S)V
       26: aload_0
       27: getfield      #4                 // Field entryStationId:S
       30: iflt          39
       33: sipush        -28409
       36: invokestatic  #19                // Method javacard/framework/ISOException
         .throwIt:(S)V
       39: invokestatic  #62                // Method javacard/framework/JCSystem.
         beginTransaction:()V
       42: aload_0
       43: aload_1
       44: iload_2
       45: invokestatic  #63                // Method javacard/framework/Util.
         getShort:([BS)S
       48: putfield      #4                 // Field entryStationId:S
       51: aload_0
       52: dup
       53: getfield      #5                 // Field correlationId:B
       56: iconst_1
       57: iadd
       58: i2b
       59: putfield      #5                 // Field correlationId:B
       62: invokestatic  #64                // Method javacard/framework/JCSystem.
         commitTransaction:()V
       65: iconst_0
       66: istore        4
       68: aload_0
       69: getfield      #20                // Field uid:[B
       72: iconst_0
       73: aload_1
       74: iload         4
       76: bipush        8
       78: invokestatic  #21                // Method javacard/framework/Util.
         arrayCopy:([BS[BSS)S
       81: istore        4
       83: aload_1
       84: iload         4
       86: aload_0
       87: getfield      #5                 // Field correlationId:B
       90: i2s
       91: invokestatic  #48                // Method javacard/framework/Util.
         setShort:([BSS)S
       94: istore        4
       96: iload         4
       98: ireturn

  private short processExit(byte[], short, short);
    Code:
        0: iload_3
        1: iconst_1
        2: if_icmpeq     11
        5: sipush        26368
```

```
       8: invokestatic  #19                 // Method javacard/framework/ISOException
           .throwIt:(S)V
      11: aload_0
      12: getfield      #3                  // Field balance:S
      15: bipush        10
      17: if_icmpge     26
      20: sipush        -28410
      23: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      26: aload_0
      27: getfield      #4                  // Field entryStationId:S
      30: ifge          39
      33: sipush        -28409
      36: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      39: aload_1
      40: iload_2
      41: baload
      42: istore        4
      44: aload_0
      45: getfield      #3                  // Field balance:S
      48: iload         4
      50: if_icmpge     59
      53: sipush        27269
      56: invokestatic  #19                 // Method javacard/framework/ISOException
          .throwIt:(S)V
      59: invokestatic  #62                 // Method javacard/framework/JCSystem.
          beginTransaction:()V
      62: aload_0
      63: dup
      64: getfield      #3                  // Field balance:S
      67: iload         4
      69: isub
      70: i2s
      71: putfield      #3                  // Field balance:S
      74: aload_0
      75: iconst_m1
      76: putfield      #4                  // Field entryStationId:S
      79: invokestatic  #64                 // Method javacard/framework/JCSystem.
          commitTransaction:()V
      82: iconst_0
      83: istore        5
      85: aload_0
      86: getfield      #20                 // Field uid:[B
      89: iconst_0
      90: aload_1
      91: iload         5
      93: bipush        8
      95: invokestatic  #21                 // Method javacard/framework/Util.
          arrayCopy:([BS[BSS)S
      98: istore        5
     100: aload_1
     101: iload         5
     103: aload_0
     104: getfield      #5                  // Field correlationId:B
     107: i2s
     108: invokestatic  #48                 // Method javacard/framework/Util.
         setShort:([BSS)S
     111: istore        5
     113: iload         5
     115: ireturn

  private short credit(byte[], short, short);
    Code:
       0: aload_0
```

```
       1: getfield      #26                 // Field pin:Ljavacard/framework/OwnerPIN;
       4: invokevirtual #65                 // Method javacard/framework/OwnerPIN.isValidated:()Z
       7: ifne          16
      10: sipush        25345
      13: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      16: iload_3
      17: iconst_1
      18: if_icmpeq     27
      21: sipush        26368
      24: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      27: aload_1
      28: iload_2
      29: baload
      30: istore        4
      32: iload         4
      34: bipush        100
      36: if_icmpgt     44
      39: iload         4
      41: ifge          50
      44: sipush        27267
      47: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      50: aload_0
      51: getfield      #3                  // Field balance:S
      54: iload         4
      56: iadd
      57: i2s
      58: sipush        500
      61: if_icmple     70
      64: sipush        27268
      67: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      70: aload_0
      71: dup
      72: getfield      #3                  // Field balance:S
      75: iload         4
      77: iadd
      78: i2s
      79: putfield      #3                  // Field balance:S
      82: iconst_0
      83: ireturn

  private short getBalance(byte[], short, short);
    Code:
       0: aload_0
       1: getfield      #26                 // Field pin:Ljavacard/framework/OwnerPIN;
       4: invokevirtual #65                 // Method javacard/framework/OwnerPIN.isValidated:()Z
       7: ifne          16
      10: sipush        25345
      13: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      16: iload_3
      17: ifeq          26
      20: sipush        26368
      23: invokestatic  #19                 // Method javacard/framework/ISOException.throwIt:(S)V
      26: iconst_0
      27: istore        4
      29: aload_1
```

```
      30: iload         4
      32: aload_0
      33: getfield      #3                  // Field balance:S
      36: invokestatic  #48                 // Method javacard/framework/Util.
          setShort:([BSS)S
      39: istore        4
      41: iload         4
      43: ireturn

  private byte[] fixParity(byte[], short, short);
    Code:
       0: iconst_0
       1: istore        4
       3: iload         4
       5: iload_3
       6: if_icmpge     98
       9: iconst_0
      10: istore        5
      12: aload_1
      13: iload_2
      14: iload         4
      16: iadd
      17: i2s
      18: dup2
      19: baload
      20: sipush        254
      23: iand
      24: i2b
      25: bastore
      26: iconst_1
      27: istore        6
      29: iload         6
      31: bipush        8
      33: if_icmpge     69
      36: aload_1
      37: iload_2
      38: iload         4
      40: iadd
      41: i2s
      42: baload
      43: iconst_1
      44: iload         6
      46: ishl
      47: i2b
      48: iand
      49: ifeq          59
      52: iload         5
      54: iconst_1
      55: iadd
      56: i2s
      57: istore        5
      59: iload         6
      61: iconst_1
      62: iadd
      63: i2b
      64: istore        6
      66: goto          29
      69: iload         5
      71: iconst_2
      72: irem
      73: ifne          88
      76: aload_1
      77: iload_2
      78: iload         4
      80: iadd
```

```
       81: i2s
       82: dup2
       83: baload
       84: iconst_1
       85: ior
       86: i2b
       87: bastore
       88: iload        4
       90: iconst_1
       91: iadd
       92: i2b
       93: istore       4
       95: goto         3
       98: aload_1
       99: areturn
}
```

Listing A.2: Bytecode of the TransitApplet without implemented countermeasures.

```
Compiled from "TransitApplet.java"
public class com.sun.javacard.samples.transit.TransitApplet extends javacard.
    framework.Applet {
  static final byte VERIFY;

  static final byte INITIALIZE_SESSION;

  static final byte PROCESS_REQUEST;

  static final byte PROCESS_ENTRY;

  static final byte PROCESS_EXIT;

  static final byte CREDIT;

  static final byte GET_BALANCE;

  static final short TLV_TAG_OFFSET;

  static final short TLV_LENGTH_OFFSET;

  static final short TLV_VALUE_OFFSET;

  static final short MAX_BALANCE;

  static final short MIN_TRANSIT_BALANCE;

  static final short MAX_CREDIT_AMOUNT;

  static final byte MAX_PIN_TRIES;

  static final byte MAX_PIN_SIZE;

  static final short SW_VERIFICATION_FAILED;

  static final short SW_PIN_VERIFICATION_REQUIRED;

  static final short SW_INVALID_TRANSACTION_AMOUNT;

  static final short SW_EXCEED_MAXIMUM_BALANCE;

  static final short SW_NEGATIVE_BALANCE;

  static final short SW_WRONG_SIGNATURE;
```

```
static final short SW_MIN_TRANSIT_BALANCE;

static final short SW_INVALID_TRANSIT_STATE;

static final short SW_SUCCESS;

static final short UID_LENGTH;

static final short LENGTH_DES_BYTE;

static final short CHALLENGE_LENGTH;

static final short MAC_LENGTH;

private byte[] uid;

private javacardx.crypto.Cipher cipher;

private javacard.security.DESKey staticKey;

private byte[] cardChallenge;

private byte[] keyDerivationData;

private byte[] sessionKeyData;

private javacard.security.DESKey sessionKey;

private boolean useTransientKey;

private javacard.security.Signature signature;

private javacard.security.RandomData random;

private javacard.framework.OwnerPIN pin;

private short balance;

private short entryStationId;

private byte correlationId;

protected com.sun.javacard.samples.transit.TransitApplet(byte[], short, byte);
  Code:
     0: aload_0
     1: invokespecial #86                 // Method javacard/framework/Applet."<
         init>":()V
     4: aload_0
     5: iconst_1
     6: putfield      #88                 // Field useTransientKey:Z
     9: aload_0
    10: iconst_0
    11: putfield      #90                 // Field balance:S
    14: aload_0
    15: iconst_m1
    16: putfield      #92                 // Field entryStationId:S
    19: aload_0
    20: iconst_0
    21: putfield      #94                 // Field correlationId:B
    24: aload_0
    25: iconst_3
    26: bipush        64
    28: iconst_0
```

```
      29: invokestatic  #100                // Method javacard/security/KeyBuilder.buildKey:(BSZ)Ljavacard/security/Key;
      32: checkcast     #102                // class javacard/security/DESKey
      35: putfield      #104                // Field staticKey:Ljavacard/security/DESKey;
      38: aload_0
      39: iconst_3
      40: iconst_0
      41: invokestatic  #110                // Method javacardx/crypto/Cipher.getInstance:(BZ)Ljavacardx/crypto/Cipher;
      44: putfield      #112                // Field cipher:Ljavacardx/crypto/Cipher;
      47: aload_0
      48: iconst_4
      49: iconst_2
      50: invokestatic  #118                // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
      53: putfield      #120                // Field cardChallenge:[B
      56: aload_0
      57: bipush        8
      59: iconst_2
      60: invokestatic  #118                // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
      63: putfield      #122                // Field keyDerivationData:[B
      66: aload_0
      67: iconst_2
      68: aload_0
      69: getfield      #122                // Field keyDerivationData:[B
      72: arraylength
      73: imul
      74: i2s
      75: iconst_2
      76: invokestatic  #118                // Method javacard/framework/JCSystem.makeTransientByteArray:(SB)[B
      79: putfield      #124                // Field sessionKeyData:[B
      82: aload_0
      83: bipush        6
      85: iconst_0
      86: invokestatic  #129                // Method javacard/security/Signature.getInstance:(BZ)Ljavacard/security/Signature;
      89: putfield      #131                // Field signature:Ljavacard/security/Signature;
      92: aload_1
      93: iload_2
      94: baload
      95: istore        4
      97: iload         4
      99: ifne          122
     102: aload_1
     103: iload_2
     104: baload
     105: istore        4
     107: iload         4
     109: ifeq          115
     112: goto          432
     115: aload_0
     116: invokevirtual #134                // Method register:()V
     119: goto          133
     122: aload_0
     123: aload_1
     124: iload_2
     125: iconst_1
     126: iadd
     127: i2s
     128: iload         4
     130: invokevirtual #136                // Method register:([BSB)V
```

```
    133: aload_1
    134: iload_2
    135: iload        4
    137: iadd
    138: iconst_1
    139: iadd
    140: i2s
    141: istore       7
    143: iload        7
    145: baload
    146: iload        7
    148: iadd
    149: iconst_1
    150: iadd
    151: i2s
    152: istore       8
    154: iload        8
    156: iconst_1
    157: iadd
    158: i2s
    159: istore       9
    161: aload_1
    162: iload        8
    164: baload
    165: istore       5
    167: iload        5
    169: bipush       16
    171: if_icmple    213
    174: aload_1
    175: iload        8
    177: baload
    178: istore       5
    180: iload        5
    182: bipush       16
    184: if_icmpgt    190
    187: goto         432
    190: iload        5
    192: bipush       24
    194: if_icmple    219
    197: aload_1
    198: iload        8
    200: baload
    201: istore       5
    203: iload        5
    205: bipush       24
    207: if_icmpgt    213
    210: goto         432
    213: sipush       26368
    216: invokestatic #142                // Method javacard/framework/ISOException
        .throwIt:(S)V
    219: aload_0
    220: bipush       8
    222: newarray      byte
    224: putfield     #144                // Field uid:[B
    227: aload_1
    228: iload        9
    230: aload_0
    231: getfield     #144                // Field uid:[B
    234: iconst_0
    235: bipush       8
    237: invokestatic #150                // Method javacard/framework/Util.
        arrayCopy:([BS[BSS)S
    240: pop
    241: iload        9
    243: bipush       8
```

```
     245: iadd
     246: i2s
     247: istore        10
     249: aload_0
     250: getfield      #104                // Field staticKey:Ljavacard/security/
         DESKey;
     253: aload_0
     254: aload_1
     255: iload         10
     257: bipush        8
     259: invokespecial #154                // Method fixParity:([BSS)[B
     262: iload         10
     264: invokeinterface #158,  3          // InterfaceMethod javacard/security/
         DESKey.setKey:([BS)V
     269: iload         10
     271: bipush        8
     273: iadd
     274: i2s
     275: istore        11
     277: aload_1
     278: iload         11
     280: baload
     281: ifeq          300
     284: aload_1
     285: iload         11
     287: baload
     288: ifne          294
     291: goto          432
     294: iconst_1
     295: istore        12
     297: goto          303
     300: iconst_0
     301: istore        12
     303: aload_0
     304: iload         12
     306: putfield      #88                 // Field useTransientKey:Z
     309: aload_0
     310: new           #160                // class javacard/framework/OwnerPIN
     313: dup
     314: iconst_3
     315: bipush        8
     317: invokespecial #163                // Method javacard/framework/OwnerPIN."<
         init>":(BB)V
     320: putfield      #165                // Field pin:Ljavacard/framework/
         OwnerPIN;
     323: aload_0
     324: getfield      #165                // Field pin:Ljavacard/framework/
         OwnerPIN;
     327: aload_1
     328: iload         11
     330: iconst_1
     331: iadd
     332: i2s
     333: iload         5
     335: bipush        8
     337: isub
     338: bipush        8
     340: isub
     341: iconst_1
     342: isub
     343: i2b
     344: invokevirtual #168                // Method javacard/framework/OwnerPIN.
         update:([BSB)V
     347: aload_0
     348: getfield      #88                 // Field useTransientKey:Z
```

```
      351: istore        6
      353: iload         6
      355: ifeq          383
      358: iload         6
      360: ifne          366
      363: goto          432
      366: aload_0
      367: iconst_2
      368: bipush        64
      370: iconst_0
      371: invokestatic  #100                // Method javacard/security/KeyBuilder.
          buildKey:(BSZ)Ljavacard/security/Key;
      374: checkcast     #102                // class javacard/security/DESKey
      377: putfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      380: goto          397
      383: aload_0
      384: iconst_3
      385: bipush        64
      387: iconst_0
      388: invokestatic  #100                // Method javacard/security/KeyBuilder.
          buildKey:(BSZ)Ljavacard/security/Key;
      391: checkcast     #102                // class javacard/security/DESKey
      394: putfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      397: aload_0
      398: iconst_1
      399: invokestatic  #175                // Method javacard/security/RandomData.
          getInstance:(B)Ljavacard/security/RandomData;
      402: putfield      #177                // Field random:Ljavacard/security/
          RandomData;
      405: aload_0
      406: getfield      #177                // Field random:Ljavacard/security/
          RandomData;
      409: aload_0
      410: getfield      #144                // Field uid:[B
      413: iconst_0
      414: bipush        8
      416: invokevirtual #181                // Method javacard/security/RandomData.
          setSeed:([BSS)V
      419: aload_0
      420: getfield      #112                // Field cipher:Ljavacardx/crypto/Cipher;
      423: aload_0
      424: getfield      #104                // Field staticKey:Ljavacard/security/
          DESKey;
      427: iconst_2
      428: invokevirtual #185                // Method javacardx/crypto/Cipher.init:(
          Ljavacard/security/Key;B)V
      431: return
      432: goto          432

  private boolean checkMAC(byte[]);
    Code:
        0: bipush        11
        2: getstatic     #207                // Field tool/generated/CGII.identifier:S
        5: if_icmpne     79
        8: aload_1
        9: iconst_4
       10: baload
       11: istore_2
       12: iload_2
       13: bipush        8
       15: if_icmpgt     37
       18: aload_1
       19: iconst_4
```

```
      20: baload
      21: istore_2
      22: iload_2
      23: bipush        8
      25: if_icmple     31
      28: goto          79
      31: sipush        26368
      34: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      37: aload_0
      38: getfield      #131                // Field signature:Ljavacard/security/
          Signature;
      41: aload_0
      42: getfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      45: iconst_2
      46: invokevirtual #208                // Method javacard/security/Signature.
          init:(Ljavacard/security/Key;B)V
      49: aload_0
      50: getfield      #131                // Field signature:Ljavacard/security/
          Signature;
      53: aload_1
      54: iconst_5
      55: iload_2
      56: bipush        8
      58: isub
      59: i2s
      60: aload_1
      61: iconst_5
      62: iload_2
      63: iadd
      64: bipush        8
      66: isub
      67: i2s
      68: bipush        8
      70: invokevirtual #212                // Method javacard/security/Signature.
          verify:([BSS[BSS)Z
      73: bipush        12
      75: putstatic     #207                // Field tool/generated/CGII.identifier:S
      78: ireturn
      79: goto          79

  private short credit(byte[], short, short);
    Code:
       0: bipush        19
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     170
       8: aload_0
       9: getfield      #165                // Field pin:Ljavacard/framework/
           OwnerPIN;
      12: astore        5
      14: aload         5
      16: invokevirtual #218                // Method javacard/framework/OwnerPIN.
          isValidated:()Z
      19: ifne          39
      22: aload         5
      24: invokevirtual #218                // Method javacard/framework/OwnerPIN.
          isValidated:()Z
      27: ifeq          33
      30: goto          170
      33: sipush        25345
      36: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      39: iload_3
      40: iconst_1
```

```
      41: if_icmpeq     58
      44: iload_3
      45: iconst_1
      46: if_icmpne     52
      49: goto          170
      52: sipush        26368
      55: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      58: aload_1
      59: iload_2
      60: baload
      61: istore        4
      63: iload         4
      65: bipush        100
      67: if_icmpgt     103
      70: aload_1
      71: iload_2
      72: baload
      73: istore        4
      75: iload         4
      77: bipush        100
      79: if_icmple     85
      82: goto          170
      85: iload         4
      87: ifge          109
      90: aload_1
      91: iload_2
      92: baload
      93: istore        4
      95: iload         4
      97: iflt          103
     100: goto          170
     103: sipush        27267
     106: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     109: aload_0
     110: getfield      #90                 // Field balance:S
     113: iload         4
     115: iadd
     116: i2s
     117: sipush        500
     120: if_icmple     151
     123: aload_1
     124: iload_2
     125: baload
     126: istore        4
     128: aload_0
     129: getfield      #90                 // Field balance:S
     132: iload         4
     134: iadd
     135: i2s
     136: sipush        500
     139: if_icmpgt     145
     142: goto          170
     145: sipush        27268
     148: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     151: aload_0
     152: aload_0
     153: getfield      #90                 // Field balance:S
     156: iload         4
     158: iadd
     159: i2s
     160: putfield      #90                 // Field balance:S
     163: bipush        20
```

```
      165: putstatic     #207                // Field tool/generated/CGII.identifier:S
      168: iconst_0
      169: ireturn
      170: goto          170

  public void deselect();
    Code:
       0: aload_0
       1: getfield      #165                // Field pin:Ljavacard/framework/
           OwnerPIN;
       4: invokevirtual #226                // Method javacard/framework/OwnerPIN.
           reset:()V
       7: aload_0
       8: getfield      #88                 // Field useTransientKey:Z
      11: ifne          33
      14: aload_0
      15: getfield      #88                 // Field useTransientKey:Z
      18: ifeq          24
      21: goto          34
      24: aload_0
      25: getfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      28: invokeinterface #229,  1          // InterfaceMethod javacard/security/
          DESKey.clearKey:()V
      33: return
      34: goto          34

  private byte[] fixParity(byte[], short, short);
    Code:
       0: bipush        9
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     180
       8: iconst_0
       9: istore        10
      11: iload         10
      13: iload_3
      14: if_icmpge     173
      17: iload         10
      19: iload_3
      20: if_icmplt     26
      23: goto          180
      26: iconst_0
      27: istore        11
      29: aload_1
      30: aload_1
      31: iload_2
      32: iload         10
      34: iadd
      35: i2s
      36: istore        4
      38: iload         4
      40: baload
      41: sipush        254
      44: iand
      45: i2b
      46: istore        5
      48: iload         4
      50: iload         5
      52: bastore
      53: iconst_1
      54: istore        12
      56: iload         12
      58: bipush        8
      60: if_icmpge     122
      63: iload         12
```

```
     65: bipush        8
     67: if_icmplt     73
     70: goto          180
     73: aload_1
     74: iload_2
     75: iload         10
     77: iadd
     78: i2s
     79: baload
     80: iconst_1
     81: iload         12
     83: ishl
     84: i2b
     85: iand
     86: istore        9
     88: iload         9
     90: ifeq          112
     93: iload         9
     95: ifne          101
     98: goto          180
    101: iload         11
    103: iconst_1
    104: iadd
    105: dup
    106: i2s
    107: istore        11
    109: i2s
    110: istore        13
    112: iload         12
    114: iconst_1
    115: iadd
    116: i2b
    117: istore        12
    119: goto          56
    122: iload         11
    124: iconst_2
    125: irem
    126: istore        6
    128: iload         6
    130: ifne          163
    133: iload         6
    135: ifeq          141
    138: goto          180
    141: aload_1
    142: aload_1
    143: iload_2
    144: iload         10
    146: iadd
    147: i2s
    148: istore        7
    150: iload         7
    152: baload
    153: iconst_1
    154: ior
    155: nop
    156: istore        8
    158: iload         7
    160: iload         8
    162: bastore
    163: iload         10
    165: iconst_1
    166: iadd
    167: i2b
    168: istore        10
    170: goto          11
```

```
     173: bipush        10
     175: putstatic     #207                // Field tool/generated/CGII.identifier:S
     178: aload_1
     179: areturn
     180: goto          180

  private void generateCardChallenge();
    Code:
       0: iconst_3
       1: getstatic     #207                // Field tool/generated/CGII.identifier:S
       4: if_icmpne     25
       7: aload_0
       8: getfield      #177                // Field random:Ljavacard/security/
         RandomData;
      11: aload_0
      12: getfield      #120                // Field cardChallenge:[B
      15: iconst_0
      16: iconst_4
      17: invokevirtual #241                // Method javacard/security/RandomData.
         generateData:([BSS)V
      20: iconst_4
      21: putstatic     #207                // Field tool/generated/CGII.identifier:S
      24: return
      25: goto          25

  private void generateKeyDerivationData(byte[]);
    Code:
       0: bipush        17
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     64
       8: aload_1
       9: iconst_4
      10: baload
      11: iconst_4
      12: if_icmpge     31
      15: aload_1
      16: iconst_4
      17: baload
      18: iconst_4
      19: if_icmplt     25
      22: goto          64
      25: sipush        26368
      28: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
      31: aload_1
      32: iconst_5
      33: aload_0
      34: getfield      #122                // Field keyDerivationData:[B
      37: iconst_0
      38: iconst_4
      39: invokestatic  #150                // Method javacard/framework/Util.
         arrayCopy:([BS[BSS)S
      42: pop
      43: aload_0
      44: getfield      #120                // Field cardChallenge:[B
      47: iconst_0
      48: aload_0
      49: getfield      #122                // Field keyDerivationData:[B
      52: iconst_4
      53: iconst_4
      54: invokestatic  #150                // Method javacard/framework/Util.
         arrayCopy:([BS[BSS)S
      57: pop
      58: bipush        18
      60: putstatic     #207                // Field tool/generated/CGII.identifier:S
```

```
      63: return
      64: goto          64

  private short generateMAC(byte[], short);
    Code:
       0: bipush        23
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     41
       8: aload_0
       9: getfield      #131                // Field signature:Ljavacard/security/
           Signature;
      12: aload_0
      13: getfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      16: iconst_1
      17: invokevirtual #208                // Method javacard/security/Signature.
          init:(Ljavacard/security/Key;B)V
      20: iload_2
      21: aload_0
      22: getfield      #131                // Field signature:Ljavacard/security/
          Signature;
      25: aload_1
      26: iconst_0
      27: iload_2
      28: aload_1
      29: iload_2
      30: invokevirtual #249                // Method javacard/security/Signature.
          sign:([BSS[BS)S
      33: iadd
      34: i2s
      35: bipush        24
      37: putstatic     #207                // Field tool/generated/CGII.identifier:S
      40: ireturn
      41: goto          41

  private void generateSessionKey();
    Code:
       0: bipush        7
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     80
       8: aload_0
       9: getfield      #112                // Field cipher:Ljavacardx/crypto/Cipher;
      12: aload_0
      13: getfield      #122                // Field keyDerivationData:[B
      16: iconst_0
      17: aload_0
      18: getfield      #122                // Field keyDerivationData:[B
      21: arraylength
      22: i2s
      23: aload_0
      24: getfield      #124                // Field sessionKeyData:[B
      27: iconst_0
      28: invokevirtual #253                // Method javacardx/crypto/Cipher.doFinal
          :([BSS[BS)S
      31: pop
      32: aload_0
      33: getfield      #170                // Field sessionKey:Ljavacard/security/
          DESKey;
      36: astore_1
      37: aload_0
      38: aload_0
      39: getfield      #124                // Field sessionKeyData:[B
      42: iconst_0
      43: aload_0
      44: getfield      #124                // Field sessionKeyData:[B
```

```
      47: arraylength
      48: i2s
      49: bipush        9
      51: putstatic     #207                // Field tool/generated/CGII.identifier:S
      54: invokespecial #154                // Method fixParity:([BSS)[B
      57: astore_2
      58: getstatic     #207                // Field tool/generated/CGII.identifier:S
      61: bipush        10
      63: if_icmpne     80
      66: aload_1
      67: aload_2
      68: iconst_0
      69: invokeinterface #158,  3          // InterfaceMethod javacard/security/
          DESKey.setKey:([BS)V
      74: bipush        8
      76: putstatic     #207                // Field tool/generated/CGII.identifier:S
      79: return
      80: goto          80

  private short getBalance(byte[], short, short);
    Code:
       0: iconst_1
       1: getstatic     #207                // Field tool/generated/CGII.identifier:S
       4: if_icmpne     69
       7: aload_0
       8: getfield      #165                // Field pin:Ljavacard/framework/
           OwnerPIN;
      11: astore        4
      13: aload         4
      15: invokevirtual #218                // Method javacard/framework/OwnerPIN.
          isValidated:()Z
      18: ifne          38
      21: aload         4
      23: invokevirtual #218                // Method javacard/framework/OwnerPIN.
          isValidated:()Z
      26: ifeq          32
      29: goto          69
      32: sipush        25345
      35: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      38: iload_3
      39: ifeq          55
      42: iload_3
      43: ifne          49
      46: goto          69
      49: sipush        26368
      52: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      55: aload_1
      56: iconst_0
      57: aload_0
      58: getfield      #90                 // Field balance:S
      61: invokestatic  #259                // Method javacard/framework/Util.
          setShort:([BSS)S
      64: iconst_2
      65: putstatic     #207                // Field tool/generated/CGII.identifier:S
      68: ireturn
      69: goto          69

  private void initializeSession(javacard.framework.APDU);
    Code:
       0: bipush        25
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     184
       8: aload_1
```

```
         9: invokevirtual #267                  // Method javacard/framework/APDU.
            getBuffer:()[B
        12: astore_2
        13: aload_2
        14: iconst_2
        15: baload
        16: ifne          43
        19: aload_2
        20: iconst_2
        21: baload
        22: ifeq          28
        25: goto          184
        28: aload_2
        29: iconst_3
        30: baload
        31: ifeq          49
        34: aload_2
        35: iconst_3
        36: baload
        37: ifne          43
        40: goto          184
        43: sipush        27270
        46: invokestatic  #142                  // Method javacard/framework/ISOException
            .throwIt:(S)V
        49: aload_2
        50: iconst_4
        51: baload
        52: aload_1
        53: invokevirtual #271                  // Method javacard/framework/APDU.
            setIncomingAndReceive:()S
        56: i2b
        57: istore_3
        58: iconst_4
        59: if_icmpne     77
        62: aload_2
        63: iconst_4
        64: baload
        65: iconst_4
        66: if_icmpeq     72
        69: goto          184
        72: iload_3
        73: iconst_4
        74: if_icmpeq     83
        77: sipush        26368
        80: invokestatic  #142                  // Method javacard/framework/ISOException
            .throwIt:(S)V
        83: iconst_3
        84: putstatic     #207                  // Field tool/generated/CGII.identifier:S
        87: aload_0
        88: invokespecial #273                  // Method generateCardChallenge:()V
        91: getstatic     #207                  // Field tool/generated/CGII.identifier:S
        94: iconst_4
        95: if_icmpne     184
        98: bipush        17
       100: putstatic     #207                  // Field tool/generated/CGII.identifier:S
       103: aload_0
       104: aload_2
       105: invokespecial #275                  // Method generateKeyDerivationData:([B)V
       108: getstatic     #207                  // Field tool/generated/CGII.identifier:S
       111: bipush        18
       113: if_icmpne     184
       116: bipush        7
       118: putstatic     #207                  // Field tool/generated/CGII.identifier:S
       121: aload_0
       122: invokespecial #277                  // Method generateSessionKey:()V
```

```
      125: getstatic     #207                // Field tool/generated/CGII.identifier:S
      128: bipush        8
      130: if_icmpne     184
      133: aload_0
      134: aload_2
      135: aload_2
      136: aload_0
      137: getfield      #120                // Field cardChallenge:[B
      140: iconst_0
      141: aload_2
      142: iconst_0
      143: iconst_4
      144: invokestatic  #280                // Method javacard/framework/Util.
          arrayCopyNonAtomic:([BS[BSS)S
      147: sipush        -28672
      150: invokestatic  #259                // Method javacard/framework/Util.
          setShort:([BSS)S
      153: bipush        23
      155: putstatic     #207                // Field tool/generated/CGII.identifier:S
      158: invokespecial #282                // Method generateMAC:([BS)S
      161: istore        4
      163: getstatic     #207                // Field tool/generated/CGII.identifier:S
      166: bipush        24
      168: if_icmpne     184
      171: aload_1
      172: iconst_0
      173: iload         4
      175: invokevirtual #286                // Method javacard/framework/APDU.
          setOutgoingAndSend:(SS)V
      178: bipush        26
      180: putstatic     #207                // Field tool/generated/CGII.identifier:S
      183: return
      184: goto          184

  public static void install(byte[], short, byte);
    Code:
        0: new           #2                  // class com/sun/javacard/samples/transit
            /TransitApplet
        3: aload_0
        4: iload_1
        5: iload_2
        6: invokespecial #292                // Method "<init>":([BSB)V
        9: return

  public void process(javacard.framework.APDU);
    Code:
        0: aload_1
        1: invokevirtual #267                // Method javacard/framework/APDU.
            getBuffer:()[B
        4: astore_2
        5: aload_1
        6: invokevirtual #296                // Method javacard/framework/APDU.
            isISOInterindustryCLA:()Z
        9: ifne          89
       12: aload_1
       13: invokevirtual #296                // Method javacard/framework/APDU.
           isISOInterindustryCLA:()Z
       16: ifeq          22
       19: goto          143
       22: aload_2
       23: iconst_1
       24: baload
       25: lookupswitch  { // 2

                    48: 52
```

```
                    64: 66
               default: 80
          }
      52: aload_2
      53: iconst_1
      54: baload
      55: bipush        48
      57: if_icmpne     143
      60: aload_0
      61: aload_1
      62: invokespecial #298                // Method initializeSession:(Ljavacard/
          framework/APDU;)V
      65: return
      66: aload_2
      67: iconst_1
      68: baload
      69: bipush        64
      71: if_icmpne     143
      74: aload_0
      75: aload_1
      76: invokespecial #301                // Method processRequest:(Ljavacard/
          framework/APDU;)V
      79: return
      80: sipush        27904
      83: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      86: goto          142
      89: aload_2
      90: iconst_1
      91: baload
      92: bipush        -92
      94: if_icmpne     109
      97: aload_2
      98: iconst_1
      99: baload
     100: bipush        -92
     102: if_icmpeq     108
     105: goto          143
     108: return
     109: aload_2
     110: iconst_1
     111: baload
     112: bipush        32
     114: if_icmpne     136
     117: aload_2
     118: iconst_1
     119: baload
     120: bipush        32
     122: if_icmpeq     128
     125: goto          143
     128: aload_0
     129: aload_1
     130: invokespecial #303                // Method verify:(Ljavacard/framework/
         APDU;)V
     133: goto          142
     136: sipush        27904
     139: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     142: return
     143: goto          143

  private short processEntry(byte[], short, short);
    Code:
       0: iconst_5
```

```
         1: getstatic     #207                // Field tool/generated/CGII.identifier:S
         4: if_icmpne     129
         7: iload_3
         8: iconst_2
         9: if_icmpeq     26
        12: iload_3
        13: iconst_2
        14: if_icmpne     20
        17: goto          129
        20: sipush        26368
        23: invokestatic  #142                // Method javacard/framework/ISOException
            .throwIt:(S)V
        26: aload_0
        27: getfield      #90                 // Field balance:S
        30: bipush        10
        32: if_icmpge     53
        35: aload_0
        36: getfield      #90                 // Field balance:S
        39: bipush        10
        41: if_icmplt     47
        44: goto          129
        47: sipush        -28410
        50: invokestatic  #142                // Method javacard/framework/ISOException
            .throwIt:(S)V
        53: aload_0
        54: getfield      #92                 // Field entryStationId:S
        57: iflt          76
        60: aload_0
        61: getfield      #92                 // Field entryStationId:S
        64: ifge          70
        67: goto          129
        70: sipush        -28409
        73: invokestatic  #142                // Method javacard/framework/ISOException
            .throwIt:(S)V
        76: invokestatic  #307                // Method javacard/framework/JCSystem.
            beginTransaction:()V
        79: aload_0
        80: aload_1
        81: iload_2
        82: invokestatic  #310                // Method javacard/framework/Util.
            getShort:([BS)S
        85: putfield      #92                 // Field entryStationId:S
        88: aload_0
        89: aload_0
        90: getfield      #94                 // Field correlationId:B
        93: iconst_1
        94: iadd
        95: i2b
        96: putfield      #94                 // Field correlationId:B
        99: invokestatic  #313                // Method javacard/framework/JCSystem.
            commitTransaction:()V
       102: aload_1
       103: aload_0
       104: getfield      #144                // Field uid:[B
       107: iconst_0
       108: aload_1
       109: iconst_0
       110: bipush        8
       112: invokestatic  #150                // Method javacard/framework/Util.
           arrayCopy:([BS[BSS)S
       115: aload_0
       116: getfield      #94                 // Field correlationId:B
       119: i2s
       120: invokestatic  #259                // Method javacard/framework/Util.
           setShort:([BSS)S
```

```
     123: bipush        6
     125: putstatic     #207                // Field tool/generated/CGII.identifier:S
     128: ireturn
     129: goto          129

  private short processExit(byte[], short, short);
    Code:
       0: bipush        13
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     164
       8: iload_3
       9: iconst_1
      10: if_icmpeq     27
      13: iload_3
      14: iconst_1
      15: if_icmpne     21
      18: goto          164
      21: sipush        26368
      24: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      27: aload_0
      28: getfield      #90                 // Field balance:S
      31: bipush        10
      33: if_icmpge     54
      36: aload_0
      37: getfield      #90                 // Field balance:S
      40: bipush        10
      42: if_icmplt     48
      45: goto          164
      48: sipush        -28410
      51: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      54: aload_0
      55: getfield      #92                 // Field entryStationId:S
      58: ifge          77
      61: aload_0
      62: getfield      #92                 // Field entryStationId:S
      65: iflt          71
      68: goto          164
      71: sipush        -28409
      74: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      77: aload_1
      78: iload_2
      79: baload
      80: istore        4
      82: aload_0
      83: getfield      #90                 // Field balance:S
      86: iload         4
      88: if_icmpge     114
      91: aload_0
      92: getfield      #90                 // Field balance:S
      95: aload_1
      96: iload_2
      97: baload
      98: istore        4
     100: iload         4
     102: if_icmplt     108
     105: goto          164
     108: sipush        27269
     111: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     114: invokestatic  #307                // Method javacard/framework/JCSystem.
         beginTransaction:()V
     117: aload_0
```

```
     118: aload_0
     119: getfield      #90                 // Field balance:S
     122: iload         4
     124: isub
     125: i2s
     126: putfield      #90                 // Field balance:S
     129: aload_0
     130: iconst_m1
     131: putfield      #92                 // Field entryStationId:S
     134: invokestatic  #313                // Method javacard/framework/JCSystem.
         commitTransaction:()V
     137: aload_1
     138: aload_0
     139: getfield      #144                // Field uid:[B
     142: iconst_0
     143: aload_1
     144: iconst_0
     145: bipush        8
     147: invokestatic  #150                // Method javacard/framework/Util.
         arrayCopy:([BS[BSS)S
     150: aload_0
     151: getfield      #94                 // Field correlationId:B
     154: i2s
     155: invokestatic  #259                // Method javacard/framework/Util.
         setShort:([BSS)S
     158: bipush        14
     160: putstatic     #207                // Field tool/generated/CGII.identifier:S
     163: ireturn
     164: goto          164

  private void processRequest(javacard.framework.APDU);
    Code:
       0: bipush        21
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     385
       8: aload_1
       9: invokevirtual #267                // Method javacard/framework/APDU.
           getBuffer:()[B
      12: astore_2
      13: aload_2
      14: iconst_2
      15: baload
      16: ifne          43
      19: aload_2
      20: iconst_2
      21: baload
      22: ifeq          28
      25: goto          385
      28: aload_2
      29: iconst_3
      30: baload
      31: ifeq          49
      34: aload_2
      35: iconst_3
      36: baload
      37: ifne          43
      40: goto          385
      43: sipush        27270
      46: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      49: aload_2
      50: iconst_4
      51: baload
      52: istore_3
      53: aload_1
```

```
      54: invokevirtual #271                // Method javacard/framework/APDU.
          setIncomingAndReceive:()S
      57: i2b
      58: istore        4
      60: iload_3
      61: iload         4
      63: if_icmpeq     85
      66: aload_2
      67: iconst_4
      68: baload
      69: istore_3
      70: iload_3
      71: iload         4
      73: if_icmpne     79
      76: goto          385
      79: sipush        26368
      82: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      85: bipush        11
      87: putstatic     #207                // Field tool/generated/CGII.identifier:S
      90: aload_0
      91: aload_2
      92: invokespecial #316                // Method checkMAC:([B)Z
      95: istore        5
      97: getstatic     #207                // Field tool/generated/CGII.identifier:S
     100: bipush        12
     102: if_icmpne     385
     105: iload         5
     107: ifne          116
     110: sipush        -28411
     113: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     116: iload_3
     117: bipush        8
     119: isub
     120: aload_2
     121: bipush        6
     123: baload
     124: iconst_2
     125: iadd
     126: if_icmpeq     153
     129: aload_2
     130: iconst_4
     131: baload
     132: bipush        8
     134: isub
     135: aload_2
     136: bipush        6
     138: baload
     139: iconst_2
     140: iadd
     141: if_icmpne     147
     144: goto          385
     147: sipush        27264
     150: invokestatic  #142                // Method javacard/framework/ISOException
         .throwIt:(S)V
     153: iconst_0
     154: istore        6
     156: aload_2
     157: iconst_5
     158: baload
     159: tableswitch   { // -63 to -60

                  -63: 188
```

```
                 -62: 225

                 -61: 263

                 -60: 301
             default: 337
        }
     188: aload_2
     189: iconst_5
     190: baload
     191: bipush        -63
     193: if_icmpne     385
     196: aload_0
     197: aload_2
     198: bipush        7
     200: aload_2
     201: bipush        6
     203: baload
     204: i2s
     205: iconst_5
     206: putstatic     #207                    // Field tool/generated/CGII.identifier:S
     209: invokespecial #318                    // Method processEntry:([BSS)S
     212: istore        6
     214: getstatic     #207                    // Field tool/generated/CGII.identifier:S
     217: bipush        6
     219: if_icmpne     385
     222: goto          343
     225: aload_2
     226: iconst_5
     227: baload
     228: bipush        -62
     230: if_icmpne     385
     233: aload_0
     234: aload_2
     235: bipush        7
     237: aload_2
     238: bipush        6
     240: baload
     241: i2s
     242: bipush        13
     244: putstatic     #207                    // Field tool/generated/CGII.identifier:S
     247: invokespecial #320                    // Method processExit:([BSS)S
     250: istore        6
     252: getstatic     #207                    // Field tool/generated/CGII.identifier:S
     255: bipush        14
     257: if_icmpne     385
     260: goto          343
     263: aload_2
     264: iconst_5
     265: baload
     266: bipush        -61
     268: if_icmpne     385
     271: aload_0
     272: aload_2
     273: bipush        7
     275: aload_2
     276: bipush        6
     278: baload
     279: i2s
     280: bipush        19
     282: putstatic     #207                    // Field tool/generated/CGII.identifier:S
     285: invokespecial #322                    // Method credit:([BSS)S
     288: istore        6
     290: getstatic     #207                    // Field tool/generated/CGII.identifier:S
     293: bipush        20
```

```
      295: if_icmpne     385
      298: goto          343
      301: aload_2
      302: iconst_5
      303: baload
      304: bipush        -60
      306: if_icmpne     385
      309: aload_0
      310: aload_2
      311: bipush        7
      313: aload_2
      314: bipush        6
      316: baload
      317: i2s
      318: iconst_1
      319: putstatic     #207                // Field tool/generated/CGII.identifier:S
      322: invokespecial #324                // Method getBalance:([BSS)S
      325: istore        6
      327: getstatic     #207                // Field tool/generated/CGII.identifier:S
      330: iconst_2
      331: if_icmpne     385
      334: goto          343
      337: sipush        27265
      340: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      343: aload_0
      344: aload_2
      345: aload_2
      346: iload         6
      348: sipush        -28672
      351: invokestatic  #259                // Method javacard/framework/Util.
          setShort:([BSS)S
      354: bipush        23
      356: putstatic     #207                // Field tool/generated/CGII.identifier:S
      359: invokespecial #282                // Method generateMAC:([BS)S
      362: istore        7
      364: getstatic     #207                // Field tool/generated/CGII.identifier:S
      367: bipush        24
      369: if_icmpne     385
      372: aload_1
      373: iconst_0
      374: iload         7
      376: invokevirtual #286                // Method javacard/framework/APDU.
          setOutgoingAndSend:(SS)V
      379: bipush        22
      381: putstatic     #207                // Field tool/generated/CGII.identifier:S
      384: return
      385: goto          385

  public boolean select();
    Code:
       0: aload_0
       1: getfield      #165                // Field pin:Ljavacard/framework/
            OwnerPIN;
       4: astore_1
       5: aload_1
       6: invokevirtual #331                // Method javacard/framework/OwnerPIN.
            getTriesRemaining:()B
       9: ifne          24
      12: aload_1
      13: invokevirtual #331                // Method javacard/framework/OwnerPIN.
          getTriesRemaining:()B
      16: ifeq          22
      19: goto          26
      22: iconst_0
```

```
      23: ireturn
      24: iconst_1
      25: ireturn
      26: goto          26

  private void verify(javacard.framework.APDU);
    Code:
       0: bipush        15
       2: getstatic     #207                // Field tool/generated/CGII.identifier:S
       5: if_icmpne     74
       8: aload_1
       9: invokevirtual #267                // Method javacard/framework/APDU.
          getBuffer:()[B
      12: astore_2
      13: aload_2
      14: iconst_4
      15: baload
      16: istore_3
      17: aload_1
      18: invokevirtual #271                // Method javacard/framework/APDU.
          setIncomingAndReceive:()S
      21: i2b
      22: istore        4
      24: iload_3
      25: iload         4
      27: if_icmpeq     49
      30: aload_2
      31: iconst_4
      32: baload
      33: istore_3
      34: iload_3
      35: iload         4
      37: if_icmpne     43
      40: goto          74
      43: sipush        26368
      46: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      49: aload_0
      50: getfield      #165                // Field pin:Ljavacard/framework/
          OwnerPIN;
      53: aload_2
      54: iconst_5
      55: iload_3
      56: invokevirtual #336                // Method javacard/framework/OwnerPIN.
          check:([BSB)Z
      59: ifne          68
      62: sipush        25344
      65: invokestatic  #142                // Method javacard/framework/ISOException
          .throwIt:(S)V
      68: bipush        16
      70: putstatic     #207                // Field tool/generated/CGII.identifier:S
      73: return
      74: goto          74
}
```

Listing A.3: Bytecode of the TransitApplet with implemented countermeasures.